Proceedings of the
Linux Symposium

Volume One

July 20nd–23th, 2005
Ottawa, Ontario
Canada

Conference Organizers

Andrew J. Hutton, *Steamballoon, Inc.*
C. Craig Ross, *Linux Symposium*
Stephanie Donovan, *Linux Symposium*

Review Committee

Gerrit Huizenga, *IBM*
Matthew Wilcox, *HP*
Dirk Hohndel, *Intel*
Val Henson, *Sun Microsystems*
Jamal Hadi Salimi, *Znyx*
Matt Domsch, *Dell*
Andrew Hutton, *Steamballoon, Inc.*

Proceedings Formatting Team

John W. Lockhart, *Red Hat, Inc.*

eCryptfs: An Enterprise-class Encrypted Filesystem for Linux

Michael Austin Halcrow
International Business Machines
mhalcrow@us.ibm.com

Abstract

eCryptfs is a cryptographic filesystem for Linux that stacks on top of existing filesystems. It provides functionality similar to that of GnuPG, except the process of encrypting and decrypting the data is done transparently from the perspective of the application. eCryptfs leverages the recently introduced Linux kernel keyring service, the kernel cryptographic API, the Linux Pluggable Authentication Modules (PAM) framework, OpenSSL/GPGME, the Trusted Platform Module (TPM), and the GnuPG keyring in order to make the process of key and authentication token management seamless to the end user.

1 Enterprise Requirements

Any cryptographic application is hard to implement correctly and hard to effectively deploy. When key management and interaction with the cryptographic processes are cumbersome and unwieldy, people will tend to ignore, disable, or circumvent the security measures. They will select insecure passphrases, mishandle their secret keys, or fail to encrypt their sensitive data altogether. This places the confidentiality and the integrity of the data in jeopardy of compromise in the event of unauthorized access to the media on which the data is stored.

While users and administrators take great pains to configure access control mechanisms, including measures such as user account and privilege separation, Mandatory Access Control[13], and biometric identification, they often fail to fully consider the circumstances where none of these technologies can have any effect – for example, when the media itself is separated from the control of its host environment. In these cases, access control must be enforced via cryptography.

When a business process incorporates a cryptographic solution, it must take several issues into account. How will this affect incremental backups? What sort of mitigation is in place to address key loss? What sort of education is required on the part of the employees? What should the policies be? Who should decide them, and how are they expressed? How disruptive or costly will this technology be? What class of cryptography is appropriate, given the risks? Just what are the risks, anyway? Whenever sensitive data is involved, it is incumbent upon those responsible for the information to reflect on these sorts of questions and to take action accordingly.

We see today that far too many businesses neglect to effectively utilize on-disk encryption. We often see news reports of computer equipment that is stolen in trivial cases of burglary[5] or of backup tapes with sensitive customer data that people lose track of.[10] While the physical security measures in place in these business establishments are usually sufficient given the dollar value of the actual equipment, businesses often underrate the value of the data contained on the media in that equipment. Encryption can effectively protect the data, but there there exist a variety of practical barriers to using it effectively. eCryptfs directly addresses these issues.

1.1 Integration of File Encryption into the Filesystem

Cryptography extends access control beyond the trusted domain. Within the trusted domain, physical control, authentication mechanisms, DAC/MAC[14][13], and other technologies regulate what sort of behaviors users can take with respect to data. Through various mathematical operations, cryptographic applications can enforce the confidentiality and the integrity of the data when it is not under these forms of protection. The mathematics, however, is not enough. The cryptographic solution must take human behavior into account and compensate for tendencies to take actions that compromise the security afforded by the cryptographic application.

Several solutions exist that solve separate pieces of the data encryption problem. In one example highlighting transparency, employees within an organization that uses IBM^{TM} Lotus Notes^{TM} [11] for its email will not even notice the complex PKI or the encryption process that is integrated into the product. Encryption and decryption of sensitive email messages is seamless to the end user; it involves checking an "Encrypt" box, specifying a recipient, and sending the message. This effectively addresses a significant file in-transit confidentiality problem. If the local replicated mailbox database is also encrypted, then this addresses confidentiality (to some extent) on the local storage device, but the protection is lost once the data leaves the domain of Notes (for example, if an attached file is saved to disk). The process must be seamlessly integrated into *all* relevant aspects of the user's operating environment.

We learn from this particular application that environments that embody strong hierarchical structures can more easily provide the infrastructure necessary to facilitate an easy-to-use and effective organization-wide cryptographic solution. Wherever possible, systems should leverage this infrastructure to protect sensitive information. Furthermore, when organizations with differing key management infrastructures exchange data, the cryptographic application should be flexible enough to support alternate forms of key management.

Current cryptographic solutions that ship with Linux distributions do not fully leverage existing Linux security technologies to make the process seamless and transparent. Surprisingly few filesystem-level solutions utilize public key cryptography. eCryptfs brings together the kernel cryptographic API, the kernel keyring, PAM, the TPM, and GnuPG in such a way so as to fill many of the gaps[3] that exist with current popular cryptographic technologies.

1.2 Universal Applicability

Although eCryptfs is geared toward securing data in enterprise environments, we explored how eCryptfs can be flexible for use in a wide variety of circumstances. The basic passphrase mode of operation provides equivalent functionality to that of EncFS[23] or CFS[20], with the added advantage of the ability to copy an encrypted file, as an autonomic unit, between hosts while preserving the associated cryptographic contexts. eCryptfs includes a pluggable Public Key Infrastructure API through which it can utilize arbitrary sources for public key management. One such plugin interfaces with GnuPG (see Section 5.7) in order to leverage the web-of-trust mechanism already in wide use among participants on the Internet.

1.3 Enterprise-class

We designed and implemented eCryptfs with the enterprise environment in mind. These environments entail a host of unique opportunities and requirements.

1.3.1 Ease of Deployment

eCryptfs does not require any modifications to the Linux kernel itself.[1] It is deployable as a stand-alone kernel module that utilizes a set of userspace tools to perform key management functions.

Many other cryptographic filesystem solutions, such as dm-crypt, require that a fixed partition (or image) be established upon which to write the encrypted data. This provides the flexibility of block-layer encryption; any application, such as swap, a database application, or a filesystem, can use it without any modification to the application itself. However, it is limited in that the amount of space allocated for the encrypted data is fixed. It is an inconvenient task to increase or decrease the amount of space available on the encrypted partition.

Cryptographic filesystems like EncFS[23] and CFS[20] are more easily deployable, as they operate at the VFS layer and can mount on top of any previously existing directory. These filesystems store cryptographic metadata in special files stored in the location mounted. Thus, the files themselves cannot be decrypted unless the user copies that metadata along with the encrypted files.

eCryptfs goes one step beyond other filesystems by storing cryptographic metadata directly in the files. This information is associated on a per-file basis, in a manner dictated by policies that are contained in special files on the target. These policies specify the behavior of eCryptfs as it works with individual files at the target. These policies are not required in order for the user to work with the files, but the policies can provide enhanced transparency of operation. Planned enhancements include utilities to aid in policy generation (see Section 7).

1.3.2 PKI Integration

Through its pluggable PKI interface (see Section 5.7), eCryptfs aims to be integrable with existing Public Key Infrastructures.

1.3.3 TPM Utilization

The Trusted Computing Group has published an architecture standard for hardware support for various secure operations.[7] Several vendors, including IBM, implement this standard in their products today. As an example, more recent IBM Thinkpad and workstation products ship with an integrated Trusted Computing Platform (TPM) chip.

The TPM can be configured to generate a public/private keypair in which the private exponent cannot be obtained from the chip. The session key to be encrypted or decrypted with this

[1] Note that the *key_type_user* symbol must be exported by the kernel keyring module, which may require a one-line patch for older versions of the module.

key must be passed to the chip itself, which will then use the protected private key to perform the operation. This hardware support provides a strong level of protection for the key that is beyond that which can be provided by a software implementation alone.

Using a TPM, eCryptfs can essentially "bind" a set of files to a particular host. Should the media ever be separated from the host which contains the TPM chip, the session keys (see Section 5.1) of the file will be irretrievable. The user can even configure the TPM in such a manner so that the TPM will refuse to decrypt data unless the machine is booted in a certain configuration; this helps to address attacks that involve booting the machine from untrusted media.

1.3.4 Key Escrow

Employees often forget or otherwise lose their credentials, and it is subsequently necessary for the administrator to reset or restore those credentials. Organizations expect this to happen and have processes in place to rectify the situations with a minimal amount of overhead. When strong cryptographic processes are in place to enforce data integrity and confidentiality, however, the administrator is no more capable of retrieving the keys than anyone else is, unless some steps are taken to store the key in a trustworthy escrow.

1.3.5 Incremental Backups

Cryptographic filesystem solutions that operate at the block layer do not provide adequate security when interoperating with incremental backup utilities. Solutions that store cryptographic contexts separately from the files to which they apply, as EncFS or CFS do, allow for incremental backup utilities to operate while maintaining the security of the data, but the administrator must take caution to assure that the backup tools are also recording the cryptographic metadata. Since eCryptfs stores this data in the body of the files themselves, the backup utilities do not need to take any additional measures to make a functional backup of the encrypted files.

2 Related Work

eCryptfs extends cryptfs, which is one of the filesystems instantiated by the stackable filesystem framework FiST.[9] Erez Zadok heads a research lab at Stony Brook University, where FiST development takes place. Cryptfs is an in-kernel implementation; another option would be to extend EncFS, a userspace cryptographic filesystem that utilizes FUSE to interact with the kernel VFS, to behave in a similar manner. Much of the functionality of eCryptfs revolves around key management, which can be integrated, without significant modification, into a filesystem like EncFS.

Other cryptographic filesystem solutions available under Linux include dm-crypt[18] (preceded by Cryptoloop and Loop-AES), CFS[20], BestCrypt[21], PPDD[19], TCFS[22], and CryptoFS[24]. Reiser4[25] provides a plugin framework whereby cryptographic operations can be implemented.

3 Design Structure

eCryptfs is unique from most other cryptographic filesystem solutions in that it stores a complete set of cryptographic metadata together with each individual file, much like

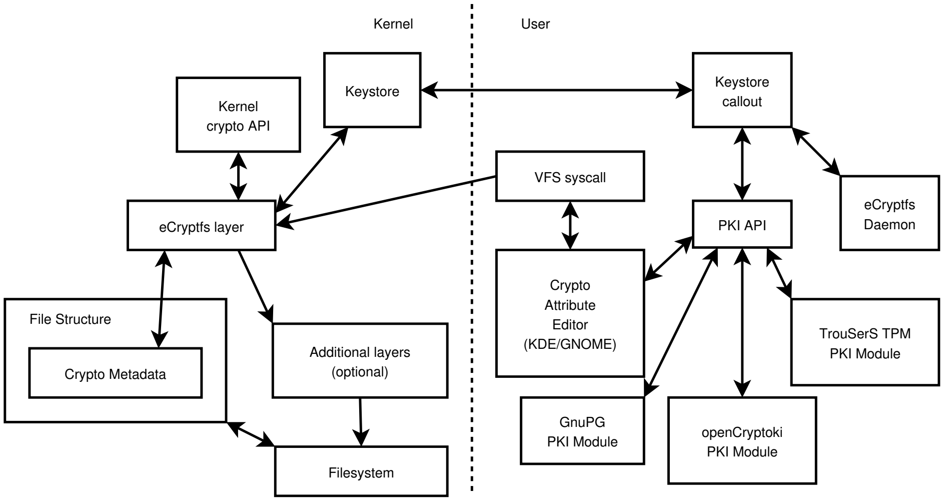

Figure 1: Overview of eCryptfs architecture

PGP-encrypted files are formatted. This allows for encrypted files to be transferred across trusted domains while maintaining the ability for those with the proper credentials to gain access to those files. Because the encryption and decryption takes place at the VFS layer, the process is made transparent from the application's perspective.

eCryptfs is implemented as a kernel module augmented with various userspace utilities for performing key management functions. The kernel module performs the bulk encryption of the file contents via the kernel cryptographic API. A keystore component extracts the header information from individual files[2] and forwards this data to a callout application. The callout application evaluates the header information against the target policy and performs various operations, such as prompting the user for a passphrase or decrypting a session key with a private key.

eCryptfs performs key management operations at the time that an application either opens or closes a file (see Figure 2). Since these events occur relatively infrequently in comparison to page reads and writes, the overhead involved in transferring data and control flow between the kernel and userspace is relatively insignificant. Furthermore, pushing key management functions out into userspace reduces the amount and the complexity of code that must run in kernel space.

4 Cryptographic Operations

eCryptfs performs the bulk symmetric encryption of the file contents in the kernel module portion itself. It utilizes the kernel cryptographic API.

4.1 File Format

The underlying file format for eCryptfs is based on the OpenPGP format described in RFC

[2] Note that the initial prototype of eCryptfs, demonstrated at OLS 2004, utilized Extended Attributes (EA) to store the cryptographic context. Due to the fact that EA's are not ubiquitously and consistently supported, this information was moved directly into the file contents. eCryptfs now uses EA's to cache cryptographic contexts, but EA support is not required for correct operation.

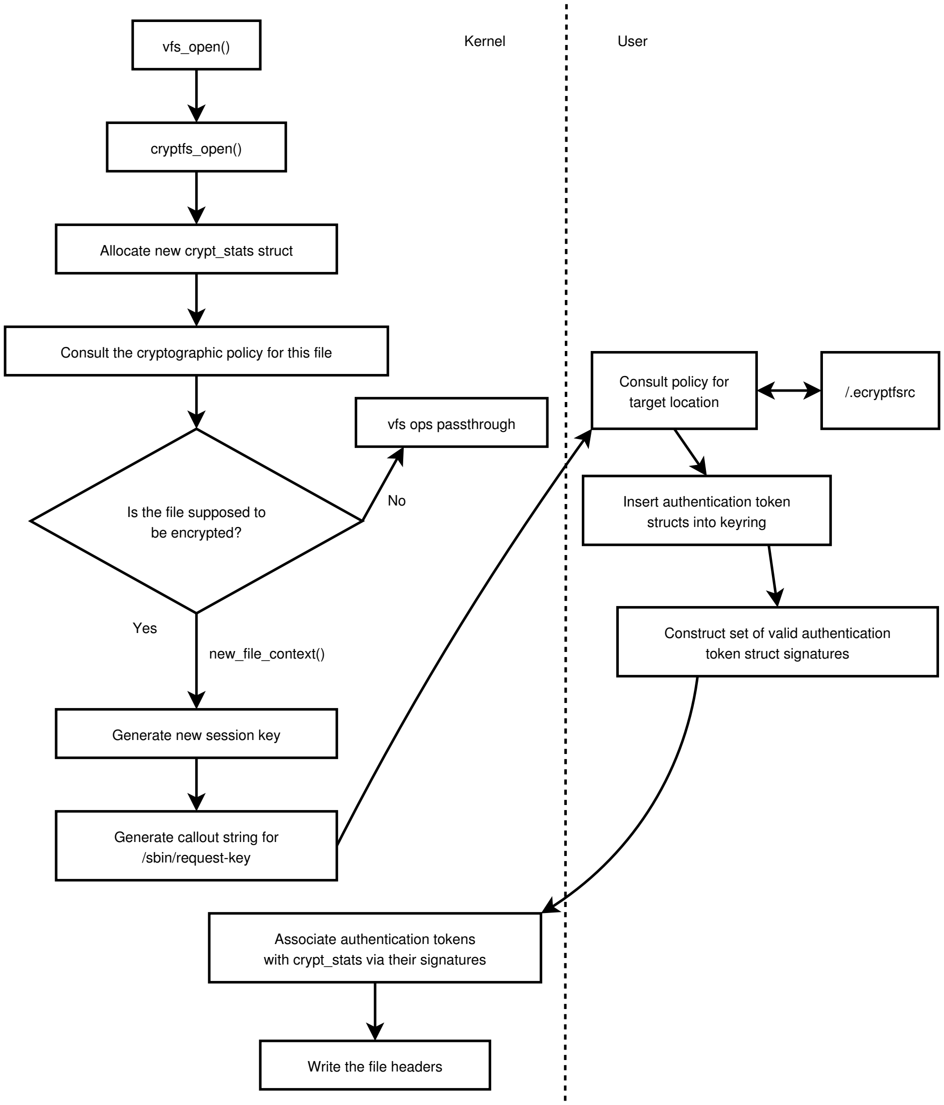
Kernel
User
vfs_open()
cryptfs_open()
Allocate new crypt_stats struct
Consult the cryptographic policy for this file
vfs ops passthrough
No
Is the file supposed to be encrypted?
Yes
new_file_context()
Generate new session key
Generate callout string for /sbin/request-key
Consult policy for target location
/.ecryptfsrc
Insert authentication token structs into keyring
Construct set of valid authentication token struct signatures
Associate authentication tokens with crypt_stats via their signatures
Write the file headers

Figure 2: New file process

2440[2] (see Figure 3). In order to accommodate random access, eCryptfs necessarily deviates from that standard to some extent. The OpenPGP standard assumes that the encryption and decryption is done as an atomic operation over the entire data contents of the file; there is no concept of a partially encrypted or decrypted file. Since the data is encrypted using a chained block cipher, it would be impossible to read the very last byte of a file without first decrypting the entire contents of the file up to that point. Likewise, writing the very first byte of the file would require re-encrypting the entire contents of the file from that point.

To compensate for this particular issue while maintaining the security afforded by a cipher operating in block chaining mode[6], eCryptfs breaks the data into extents. These extents, by default, span the page size (as specified for each kernel build). Data is dealt with on a per-extent basis; any data read from the middle of an extent causes that entire extent to be decrypted, and any data written to that extent causes that

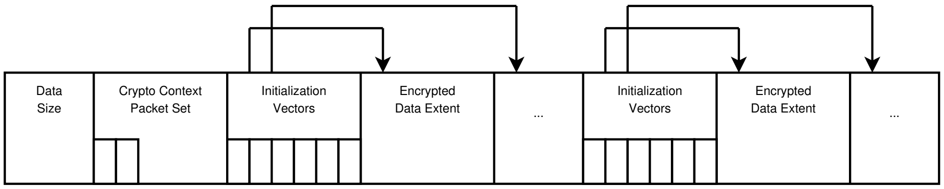

Figure 3: Underlying file format

entire extent to be encrypted.

Each extent has a unique initialization vector (IV) associated with it. One extent containing IV's precedes a group of extents to which those IV's apply. Whenever data is written to an extent, its associated IV is rotated and rewritten to the IV extent before the associated data extent is encrypted. The extents are encrypted with the block cipher selected by policy for that file and employ CBC mode to chain the blocks.

4.1.1 Sparse Files

Sparse files present a challenge for eCryptfs. Under UNIX semantics, a file becomes sparse when an application seeks past the end of a file. The regions of the file where no data is written represent *holes*. No data is actually written to the disk for these regions; the filesystem "fakes it" by specially marking the regions and setting the reported filesize accordingly. The space occupied on the disk winds up being less than the size of the file as reported by the file's inodes. When sparse regions are read, the filesystem simply pretends to be reading the data from the disk by filling in zero's for the data.

The underlying file structure for eCryptfs is amenable to accommodating this behavior; IV's consisting of all zero's can indicate that the underlying region that corresponds is sparse. The obvious problem with this approach is that it is readily apparent to an attacker which regions of the file consist of holes, and this may constitute an unacceptable breach of confidentiality. It makes sense to relegate eCryptfs's behavior with respect to sparse files as something that policy decides.

4.2 Kernel Crypto API

eCryptfs performs the bulk data encryption in the kernel module, and hence it takes advantage of the kernel cryptographic API to perform the encryption and the decryption. One of the primary motivators in implementing eCryptfs in the kernel is to avoid the overhead of context switches between userspace and kernel space, which is frequent when dealing with pages in file I/O. Any symmetric ciphers supported by the Linux kernel are candidates for usage as the bulk data ciphers for the eCryptfs files.

4.3 Header Information

eCryptfs stores the cryptographic context for each file as header information contained directly in the underlying file (see Figure 4). Thus, all of the information necessary for users with the appropriate credentials to access the file is readily available. This makes files amenable to transfer across untrusted domains while preserving the information necessary to decrypt and/or verify the contents of the file. In this respect, eCryptfs operates much like an OpenPGP application.

Most encrypted filesystem solutions either operate on the entire block device or operate on

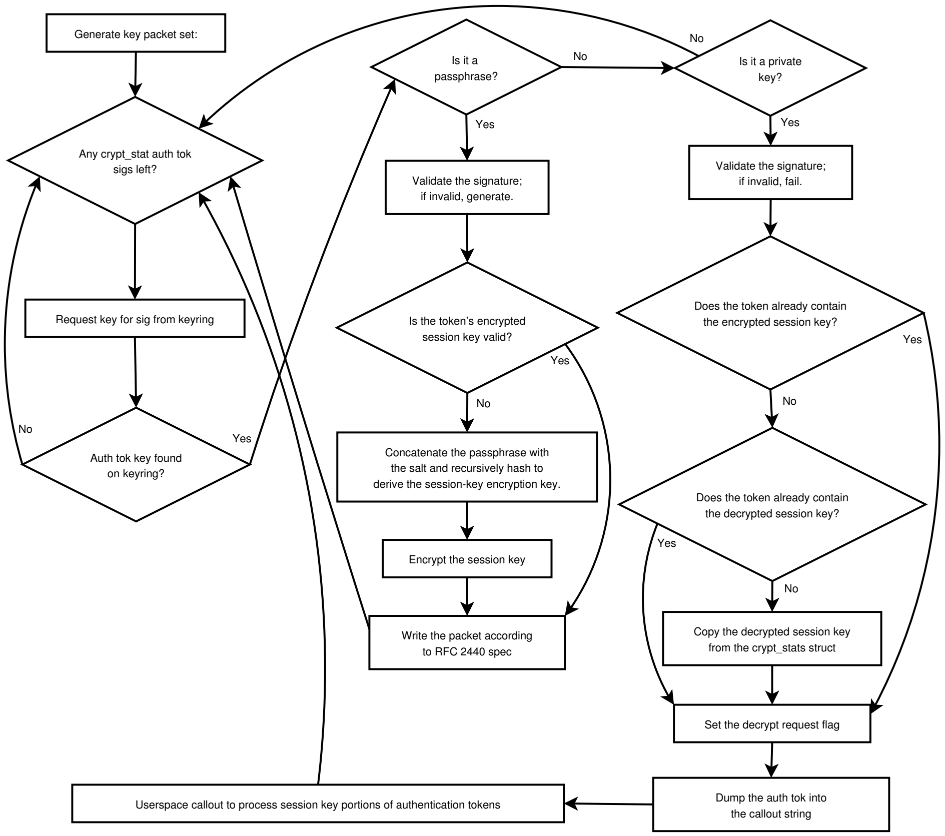
Generate key packet set:
Any crypt_stat auth tok sigs left?
Request key for sig from keyring
Auth tok key found on keyring?
No
Yes
Is it a passphrase?
No
Yes
Is it a private key?
No
Yes
Validate the signature; if invalid, generate.
Is the token's encrypted session key valid?
Yes
No
Concatenate the passphrase with the salt and recursively hash to derive the session-key encryption key.
Encrypt the session key
Write the packet according to RFC 2440 spec
Validate the signature; if invalid, fail.
Does the token already contain the encrypted session key?
Yes
No
Does the token already contain the decrypted session key?
Yes
No
Copy the decrypted session key from the crypt_stats struct
Set the decrypt request flag
Dump the auth tok into the callout string
Userspace callout to process session key portions of authentication tokens

Figure 4: Writing file headers

entire directories. There are several advantages to implementing filesystem encryption at the filesystem level and storing encryption metadata in the headers of each file:

- Granularity: Keys can be mapped to individual files, rather than entire block devices or entire directories.
- Backup Utilities: Incremental backup tools can correctly operate without having to have access to the decrypted content of the files it is backing up.
- Performance: In most cases, only certain files need to be encrypted. System libraries and executables, in general, do not need to be encrypted. By limiting the actual encryption and decryption to only those files that really need it, system resources will not be taxed as much.
- Transparent Operation: Individual encrypted files can be easily transferred off of the block device without any extra transformation, and others with authorization will be able to decrypt those files. The userspace applications and libraries do not need to be modified and recompiled to support this transparency.

4.4 Rotating Initialization Vectors

eCryptfs extents span page lengths. For most architectures, this is 4096 bytes. Subsequent writes within extents may provide information to an attacker who aims to perform linear cryptanalysis against the file. In order to mitigate this risk, eCryptfs associates a unique Initialization Vector with each extent. These IV's are interspersed throughout each file. In order to simplify and streamline the mapping of the underlying file data with the overlying file, IV's are currently grouped on a per-page basis.

4.5 HMAC's Over Extents

Integrity verification can be accomplished via sets of keyed hashes over extents within the file. Keyed hashes are used to prove that whoever modified the data had access to the shared secret, which is, in this case, the session key. Since hashes apply on a per-extent basis, eCryptfs need not generate the hash over the entire file before it can begin reading the file. If, at any time in the process of reading the file, eCryptfs detects a hash mismatch for an extent, it can flag the read operation as failing in the return code for the VFS syscall.

This technique can be applied to generate a built-in digital signature structure for files downloaded over the Internet. Given that an eCryptfs key management module is able to ascertain the trustworthiness of a particular key, then that key can be used to encode a verification packet into the file via HMAC's. This is accomplished by generating hashes over the extents of the files, as eCryptfs normally does when operating in integrity verification mode. When the file is closed, an HMAC is generated by hashing the concatenation of all of the hashes in the file, along with a secret key. This HMAC is then encrypted with the distributor's private key and written to an HMAC-type packet. The recipients of the file can proceed then to retrieve the secret key by decrypting it with the distributor's trusted public key and performing the hash operations to generate the final HMAC, which can be compared then against the HMAC that is stored in the file header in order to verify the file's integrity.

4.6 File Context

Each eCryptfs inode correlates with an inode from the underlying filesystem and has a cryptographic context associated with it. This context contains, but is not limited to, the following information:

- The session key for the file
- Whether the file is encrypted
- A pointer to the kernel crypto API context for that file
- The signatures of the authentication tokens associated with that file
- The size of the extents

eCryptfs can cache each file's cryptographic context in the user's session keyring in order to facilitate faster repeat access by bypassing the process of reading and interpreting of authentication token header information from the file.

4.7 Revocation

Since anyone with the proper credentials can extract a file's session key, revocation of access for any given credential to future versions of the file will necessitate regeneration of a session key and re-encryption of the file data with that key.

5 Key Management

eCryptfs aims to operate in a manner that is as transparent as possible to the applications and the end users of the system. Under most circumstances, when access control over the data

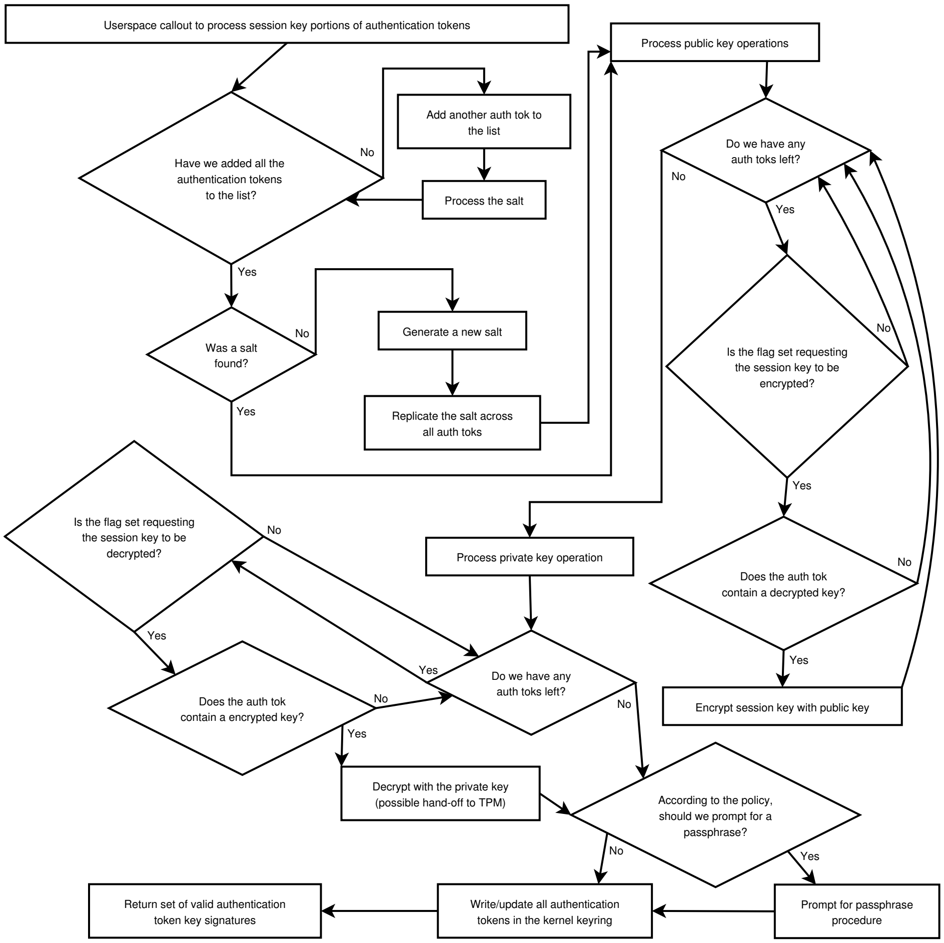
Userspace callout to process session key portions of authentication tokens
Have we added all the authentication tokens to the list?
No
Add another auth tok to the list
Process the salt
Yes
Was a salt found?
No
Generate a new salt
Replicate the salt across all auth toks
Yes
Process public key operations
Do we have any auth toks left?
No
Yes
Is the flag set requesting the session key to be encrypted?
No
Yes
Does the auth tok contain a decrypted key?
No
Yes
Encrypt session key with public key
Is the flag set requesting the session key to be decrypted?
No
Yes
Process private key operation
Do we have any auth toks left?
Yes
No
Does the auth tok contain a encrypted key?
No
Yes
Decrypt with the private key (possible hand-off to TPM)
According to the policy, should we prompt for a passphrase?
No
Yes
Prompt for passphrase procedure
Write/update all authentication tokens in the kernel keyring
Return set of valid authentication token key signatures

Figure 5: Key management

cannot be provided at all times by the host, the fact that the files are being encrypted should not be a concern for the user. Encryption must protect the confidentiality and the integrity of the files in these cases, and the system is configured to do just that, using the user's authentication credentials to generate or access the keys.

5.1 Session Keys

Every file receives a randomly generated session key, which eCryptfs uses in the bulk data encryption of the file contents. eCryptfs stores this session key in the cryptographic metadata for the file, which is in turn cached in the user's session keyring. When an application closes a newly created file, the eCryptfs encrypts the session key once for each authentication token associated with that file, as dictated by policy, then writes these encrypted session keys into packets in the header of the underlying file.

When an application later opens the file, eCryptfs reads in the encrypted session keys and chains them off of the cryptographic metadata for the file. eCryptfs looks through the user's authentication tokens to attempt to find a match with the encrypted session keys; it uses the first one found to decrypt the session key. In the event that no authentication tokens in the user's session keyring can decrypt any of the encrypted session key packets, eCryptfs falls back on policy. This policy can dictate actions such as querying PKI modules for the existence of private keys or prompting the user for a passphrase.

5.2 Passphrase

> *Passwords just don't work anymore.*
> – Bruce Schneier

Many cryptographic applications in Linux rely too heavily on passphrases to protect data. Technology that employs public key cryptography provides stronger protection against brute force attacks, given that the passphrase-protected private keys are not as easily accessible as the encrypted data files themselves.

Passphrase authentication tokens in eCryptfs exist in three forms: non-passphrased, saltless, and salted. In order to address the threat of passphrase dictionary attacks, eCryptfs utilizes the method whereby a salt value is concatenated with a passphrase to generate a passphrase identifier. The concatenated value is iteratively hashed (65,537 times by default) to generate the identifying signature for the salted authentication token.

On the other hand, saltless authentication tokens exist only in the kernel keyring and are not at any time written out to disk. The userspace callout application combines these saltless authentication tokens with non-passphrased authentication tokens to generate candidate salted authentication tokens, whose signatures are compared against those in file headers.

While eCryptfs supports passphrase-based protection of files, we do not recommend using passphrases for relatively high-value data that requires more than casual protection. Most passphrases that people are capable of remembering are becoming increasingly vulnerable to brute force attacks. eCryptfs takes measures to make such attacks more difficult, but these measures can only be so effective against a determined and properly equipped adversary.

Every effort should be made to employ the use of a TPM and public key cryptography to provide strong protection of data. Keep in mind that using a passphrase authentication token in addition to a public key authentication token does not in any way combine the security of both; rather, it combines the *insecurity* of both. This is due to the fact that, given two authentication tokens, eCryptfs will encrypt and store two copies of the session key (see Section 5.1) that can individually be attacked.

5.3 Kernel Keyring

David Howells recently authored the keyring service, which kernel versions 2.6.10 and later now include. This keyring provides a host of features to manage and protect keys and authentication tokens. eCryptfs takes advantage of the kernel keyring, utilizing it to store authentication tokens, inode cryptographic contexts, and keys.

5.4 Callout and Daemon

The primary contact between the eCryptfs kernel module and the userspace key management code is the request-key callout application, which the kernel keyring invokes. This callout application parses policy information from the target, which it interprets in relation to the header information in each file. It may then make calls through the PKI API in order to satisfy pending public key requests, or it may go searching for a salted passphrase with a particular signature.

In order to be able to prompt the user for a passphrase via a dialog box, eCryptfs must have an avenue whereby it can get to the user's X session. The user can provide this means by simply running a daemon. The eCryptfs daemon listens to a socket (for which the location is written to the user's session keyring). Whenever policy calls for the user to be prompted for a passphrase, the callout application can retrieve the socket's location and use it to request the daemon to prompt the user; the daemon then returns the user's passphrase to the callout application.

5.5 Userspace Utilities

To accommodate those who are not running the eCryptfs layer on their systems, userspace utilities to handle the encrypted content comprise part of the eCryptfs package. These utilities act much like scaled-down versions of GnuPG.

5.6 Pluggable Authentication Module

Pluggable Authentication Modules (PAM) provide a Discretionary Access Control (DAC)[14] mechanism whereby administrators can parameterize how a user is authenticated and what happens at the time of authentication. eCryptfs includes a module that captures the user's login passphrase and stores it in the user's session keyring. This passphrase is stored in the user's session keyring as a saltless passphrase authentication token.

Future actions by eCryptfs, based on policy, can then use this passphrase to perform cryptographic operations. For example, the login passphrase can be used to extract the user's private key from his GnuPG keyring. It could be used to derive a key (via a string-to-key operation) that is directly used to protect a session key for a set of files. Furthermore, this derived key could be combined with a key stored in a TPM in order to offer two-factor authentication (i.e., in order to access a file, the user must have (1) logged into a particular host (2) using a particular passphrase).

Due to PAM's flexibility, these operations do not need to be restricted to a passphrase. There is no reason, for example, that a key contained on a SmartCard or USB device could not be used to help authenticate the user, after which point that key is used in the above named cryptographic operations.

5.7 PKI

eCryptfs offers a pluggable Public Key Infrastructure (PKI) interface. PKI modules accept key identifiers and data, and they return encrypted or decrypted data. Whether any particular key associated with an identifier is available, trustworthy, etc., is up to the PKI module to determine.

eCryptfs PKI modules need to implement a set of functions that accept as input the key identifier and a blob of data. The modules have the responsibility to take whatever course of action is necessary to retrieve the requisite key, evaluate the trustworthiness of that key, and perform the public key operation.

eCryptfs includes a PKI module that utilizes the GnuPG Made Easy (GPGME) interface to access and utilize the user's GnuPG keyring. This module can utilize the user's login passphrase credential, which is stored in the user's session keyring by the eCryptfs PAM (see Section 5.6), to decrypt and utilize the user's private key stored on the user's keyring.

The eCryptfs TPM PKI module utilizes the TrouSerS[26] interface to communicate with the Trusted Platform Module. This allows for the use of a private key that is locked in the hardware, binding a file to a particular host.

The eCryptfs openCryptoki PKCS#11[15] framework PKI provides a mechanism for performing public key operations via various hardware devices supported by openCryptoki, including the IBM Cryptographic Accelerator (ICA) Model 2058, the IBM 4758 PCI Cryptographic Coprocessor, the Broadcom Crypto Accelerator, the AEP Crypto Accelerator, and the TPM.

It is easy to write additional PKI modules for eCryptfs. Such modules can interface with existing PKI's that utilize x.509 certificates, with certificate authorities, revocation lists, and other elements that help manage keys within an organization.

5.8 Key Escrow/Secret Sharing

In enterprise environments, it often makes sense for data confidentiality and integrity to be a shared responsibility. Just as prudent business organizations entail backup plans in the event of the sudden loss of any one employee, the data associated with business operations must survive any one individual in the company. In the vast majority of the cases, it is acceptable for all members of the business to have access to a set of data, while it is not acceptable for someone outside the company who steals a machine or a USB pen drive to have access to that data. In such cases, some forms of key escrow within the company are appropriate.

In enterprise environments where corporate and customer data are being protected cryptographically, key management and key recovery is an especially critical issue. Techniques such as secret splitting or (m,n)-threshold schemes[4] can be used within an organization to balance the need for key secrecy with the need for key recovery.

5.9 Target-centric Policies

When an application creates a new file, eCryptfs must make a number of decisions with regard to that file. Should the file be encrypted or unencrypted? If encrypted, which symmetric block cipher should be used to encrypt the data? Should the file contain HMAC's in addition to IV's? What should the session key length be? How should the session key be protected?

Protecting the session key on disk requires even more policy decisions. Should a passphrase be used? Which one, and how should it be retrieved? What should be the string-to-key parameters (i.e., which hash algorithm and the number of hash iterations)? Should any public keys be used? If so, which ones, and how should they be retrieved?

eCryptfs currently supports Apache-like policy definition files[3] that contain the policies that apply to the target in which they exist. For example, if the root directory on a USB pen drive device contains a .ecryptfsrc file, then eCryptfs will parse the policy from that file and apply it to all files under the mount point associated with that USB pen drive device.

Key definitions associate labels with (PKI, Id) tuples (see Figure 6).

Application directive definitions override default policies for the target, dependent upon the application performing the action and the type of action the application is performing (see Figure 7).

The action definitions associate labels with $(Action, Cipher, SessionKeySize)$ tuples. eCryptfs uses these directives to set cryptographic context parameters for files (see Figure 8).

[3] XML formats are currently being worked on.

```
<ApplicationDirectiveDef mutt_prompt_on_read_encrypted>
  Application /usr/bin/mutt
  Scenario OPEN_FOR_READ
  FileState ENCRYPTED
  Action PROMPT
</ApplicationDirectiveDef>

<ApplicationDirectiveDef mutt_decrypt>
  Application /usr/bin/mutt
  Scenario ALL
  FileState ENCRYPTED
  Action DECRYPT
</ApplicationDirectiveDef>

<ApplicationDirectiveDef openoffice_strong_encrypt>
  Application /usr/bin/ooffice
  Scenario OPEN_FOR_CREATE
  Action encrypt_strong
</ApplicationDirective>
```

Figure 7: Example policy: application directives

The directory policy definitions give default actions for files created under the specified directory location, along with application directives that apply to the directory location (see Figure 9).

6 Additional Measures

eCryptfs concerns itself mainly with protecting data that leaves trusted domains. Additional measures are necessary to address various threats outside the scope of eCryptfs's influence. For example, swap space should be encrypted; this can be easily accomplished with dm-crypt.[18]

Strong access control is outside the scope of the eCryptfs project, yet it is absolutely necessary to provide a comprehensive security solution for sensitive data. SE Linux[16] provides a robust Mandatory Access Control framework that can be leveraged with policies to protect the user's data and keys.

Furthermore, the system should judiciously employ timeouts or periods of accessibility/applicability of credentials. The kernel keyring provides a convenient and powerful mechanism for handling key permissions and expirations. These features must be used appropriately in order to address human oversight, such as failing to lock down a terminal or otherwise exit or invalidate a security context when the user is finished with a task.

7 Future Work

eCryptfs is currently in an experimental stage of development. While the majority of the VFS

```
<Directory />
  DefaultAction blowfish_encrypt
  DefaultState PROMPT
  DefaultPublicKeys mhalcrow legal
  DefaultPassphrase LOGIN
  # This directives for files under this location
  # that meet this criteria
  <FilePattern \.mutt_.*>
    ApplicationDirective mutt_decrypt
  </FilePattern>
  ApplicationDirective mutt_prompt_on_read_encrypted
</Directory>

# Overrides the prior set of policies
<Directory /gnucash>
  DefaultAction encrypt_strong
  DefaultPublicKeys host_tpm
</Directory>
```

Figure 9: Example policy: directory policies

functionality is implemented and functioning, eCryptfs requires testing and debugging across a wide range of platforms under a variety of workload conditions.

eCryptfs has the potential to provide weak file size secrecy in that the size of the file would only be determinable to the granularity of one extent size, given that the file size field in the header is encrypted with the session key. Strong file size secrecy is much more easily obtained through block device layer encryption, where everything about the filesystem is encrypted. eCryptfs only encrypts the data contents of the files; additional secrecy measures must address dentry's, filenames, and Extended Attributes, which are all within the realm of what eCryptfs can influence.

At this stage, eCryptfs requires extensive profiling and streamlining in order to optimize its performance. We need to investigate opportunities for caching cryptographic metadata, and variations on such attributes as the size of the extents could have a significant impact on speed.

eCryptfs policy files are equivalent to the Apache configuration files in form and complexity. eCryptfs policy files are amenable to guided generation via user utilities. Another significant area of future development includes the development of such utilities to aid in the generation of these policy files.

Desktop environments such as GNOME or KDE can provide users with a convenient interface through which to work with the cryptographic properties of the files. In one scenario, by right-clicking on an icon representing the file and selecting "Security", the user will be presented with a window that can be used to control the encryption status of the file. Such options will include whether or not the

```
<Key mhalcrow>
  PKI GPG
  Id 3F5C22A9
</Key>

<Key legal>
  PKI GPG
  Id 7AB1FF25
</Key>

<Key host_tpm>
  PKI TPM
  Id DEFAULT
</Key>
```

Figure 6: Example policy: key defs

file is encrypted, which users should be able to encrypt and decrypt the file (identified by their public keys as reported by the PKI plugin module), what cipher is used, what keylength is used, an optional passphrase that is used to encrypt the symmetric key, whether or not to use keyed hashing over extents of the file for integrity, the hash algorithms to use, whether accesses to the file when no key is available should result in an error or in the encrypted blocks being returned (as dictated by target-centric policies; see Section 5.9), and other properties that are interpreted and used by the eCryptfs layer.

8 Recognitions

We would like to express our appreciation for the contributions and input on the part of all those who have laid the groundwork for an effort toward transparent filesystem encryption. This includes contributors to FiST and Cryptfs, GnuPG, PAM, and many others from which we are basing our development efforts, as well as several members of the kernel development community.

```
<ActionDef blowfish_encrypt>
  Action ENCRYPT
  Cipher blowfish
  SessionKeySize 128
</ActionDef>

<ActionDef encrypt_strong>
  Action ENCRYPT
  Cipher aes
  SessionKeySize 256
</ActionDef>
```

Figure 8: Example policy: action defs

9 Conclusion

eCryptfs is an effort to reduce the barriers that stand in the way of the effective and ubiquitous utilization of file encryption. This is especially relevant as physical media remains exposed to theft and unauthorized access. Whenever sensitive data is being handled, it should be the *modus operandi* that the data be encrypted at all times when it is not directly being accessed in an authorized manner by the applications. Through strong and transparent key management that includes public key support, key->file association, and target-centric policies, eCryptfs provides the means whereby a cryptographic filesystem solution can be more easily and effectively deployed.

10 Availability

eCryptfs is licensed under the GNU General Public License (GPL). SourceForge is hosting the eCryptfs code base at `http://sourceforge.net/projects/ecryptfs`. We welcome any interested parties to become involved in the testing and development of eCryptfs.

11 Legal Statement

This work represents the view of the author and does not necessarily represent the view of IBM.

IBM and Lotus Notes are registered trademarks of International Business Machines Corporation in the United States, other countries, or both.

Other company, product, and service names may be trademarks or service marks of others.

References

[1] M. Blaze. "Key Management in an Encrypting File System", Proc. *Summer '94 USENIX Tech. Conference*, Boston, MA, June 1994.

[2] J. Callas, L. Donnerhacke, H. Finney, and R. Thayer. RFC 2440. November 1998. See `ftp://ftp.rfc-editor.org/in-notes/rfc2440.txt`.

[3] M. Halcrow. "Demands, Solutions, and Improvements for Linux Filesystem Security." *Proceedings of the 2004 Ottawa Linux Symposium*, Ottawa, Canada, July 2004.

[4] S.C. Kothari. "Generalized Linear Threshold Scheme." *Advances in Cryptology: Proceedings of CRYPTO 84*, Springer-Verlag, 1985, pp. 231–241.

[5] M. Liedtke. "Stolen UC Berkeley laptop exposes personal data of nearly 100,000." Associated Press. March 29, 2005. See `http://www.sfgate.com/cgi-bin/article.cgi?f=/n/a/2005/03/28/financial/f151143S80.DTL`

[6] B. Schneier. *Applied Cryptography*. New York: John Wiley & Sons, Inc., 1996. Pp. 193–197.

[7] The Trusted Computing Group. "TCG Specification Architecture Overview version 1.0." `https://www.trustedcomputinggroup.org/downloads/TCG_1_0_Architecture_Overview.pdf`

[8] E. Zadok, L. Badulescu, and A. Shender. "Cryptfs: A stackable vnode level encryption file system." *Technical Report CUCS-021-98, Computer Science Department*, Columbia University, 1998.

[9] E. Zadok and J. Nieh. "FiST: A Language for Stackable File Systems." *Proceedings of the Annual USENIX Technical Conference*, pp. 55–70, San Diego, June 2000.

[10] "Ameritrade addresses missing tape." United Press International. April 19, 2005. See `http://washingtontimes.com/upi-breaking/20050419-110638-7335r.htm`

[11] For more information on IBM Lotus Notes, see `http://www-306.ibm.com/software/lotus/`. Information on Notes security can be obtained from `http://www-10.lotus.com/ldd/`

today.nsf/f01245ebfc115aaf 8525661a006b86b9/ 232e604b847d2cad8825 6ab90074e298?OpenDocument

[12] For more information on Pluggable Authentication Modules (PAM), see http://www.kernel.org/pub/ linux/libs/pam/

[13] For more information on Mandatory Access Control (MAC), see http:// csrc.nist.gov/publications/ nistpubs/800-7/node35.html

[14] For more information on Discretionary Access Control (DAC), see http:// csrc.nist.gov/publications/ nistpubs/800-7/node25.html

[15] For more information on openCryptoki, see http://sourceforge.net/ projects/opencryptoki

[16] For more information on Security-Enhanced Linux (SE Linux), see http://www.nsa.gov/selinux/ index.cfm

[17] For more information on Samhain, see http://la-samhna.de/samhain/

[18] For more information on DM-crypt, see http://www.saout.de/misc/ dm-crypt/

[19] For more information on PPDD, see http://linux01.gwdg.de/ ~alatham/ppdd.html

[20] For more information on CFS, see http://sourceforge.net/ projects/cfsnfs/

[21] For more information on BestCrypt, see http://www.jetico.com/index. htm#/products.htm

[22] For more information on TCFS, see http://www.tcfs.it/

[23] For more information on EncFS, see http://arg0.net/users/ vgough/encfs.html

[24] For more information on CryptoFS, see http://reboot.animeirc.de/ cryptofs/

[25] For more information on Reiser4, see http: //www.namesys.com/v4/v4.html

[26] For more information on TrouSerS, see http://sourceforge.net/ projects/trousers/