ISE331: Fundamentals of Computer Security

Spring 2015

Radu Sion

File Systems Security
Encryption File Systems

Encryption File Systems (EFS)

- What is an encryption file system?
- Alternatives
  - Crypt
    - Stores plain files during editing
    - Need to supply the key several times
  - Integrated security in applications
- Goals
  - Security
  - Usability
  - Performance

Goals of EFS

- Security
  - Privacy
    - On disk
    - On wire
  - Integrity
  - Authentication
  - Authorization

Goals of EFS

- Usability
  - Convenience
  - Transparency
    - User
    - Applications
- Performance
  - Encryption
  - Integrity checking
  - Costs with indirection
    - Copying data
    - Context switching (user land vs. kernel)

Challenges in EFS

- Key Management
  - Storage of keys
    - On disk
    - In memory
      - Swapped out pages
  - Sharing of keys
    - Group management
  - Key compromise
    - Re-encrypt files
      - Costly
      - Gives adversary two versions of same file to work with
  - Key revocation

Challenges in EFS

- Utility services
  - Backup – possible after encryption ?
  - File system checker
  - De-fragmentation
- Random access
  - Cannot use stream ciphers
    - Reduces strength of privacy
  - Use block encryption
    - May leak information
      - Frequency analysis

Challenges in EFS

- Forward Secrecy
  - Data is persistent – “sitting duck effect”
    - Strong encryption
      - Long keys
    - File specific keys
    - IV or Block specific encryption
  - Granularity of encryption
    - All or nothing
    - Per file encryption

Agenda

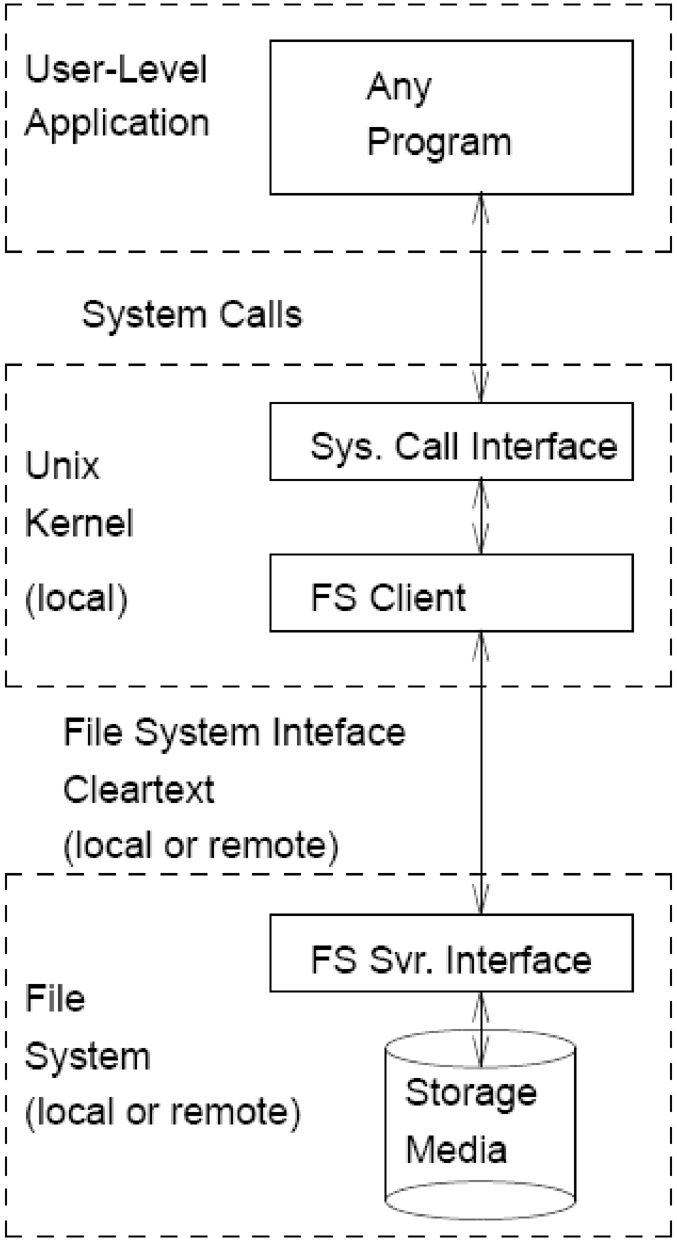
- CFS
- TCFS
- Cryptfs
- NCryptfs
- eCryptfs
- Microsoft EFS

CFS - Cryptographic File System

- First system to push encryption services in the File System layer
- Implemented in the User Layer
  - No kernel recompilation required
  - Portable
- Standard Unix FS API support
- Can use any file systems as its underlying storage
- Transparent encryption
- All or nothing encryption

CFS

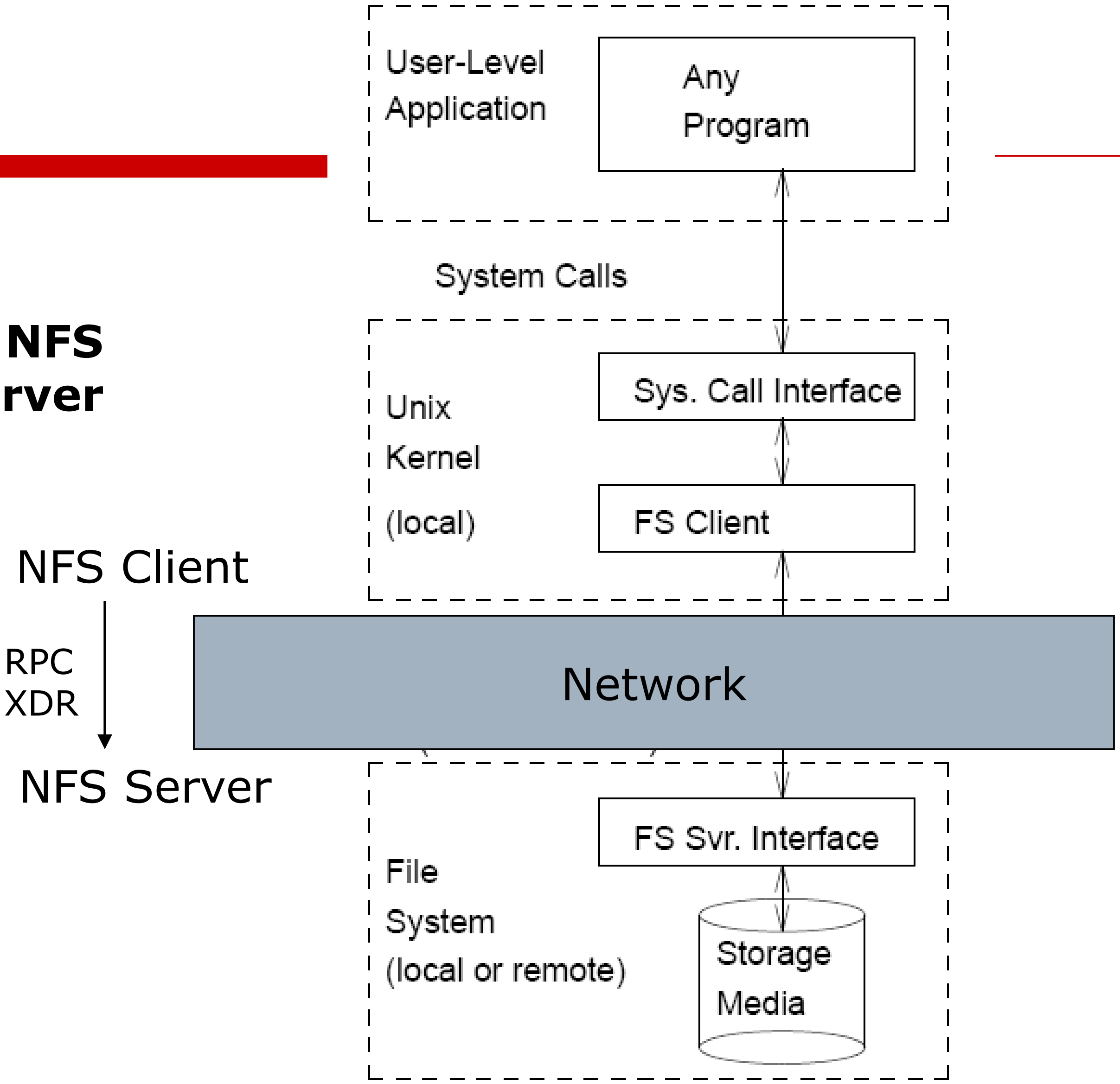
Data Flow in Standard Vnode File System



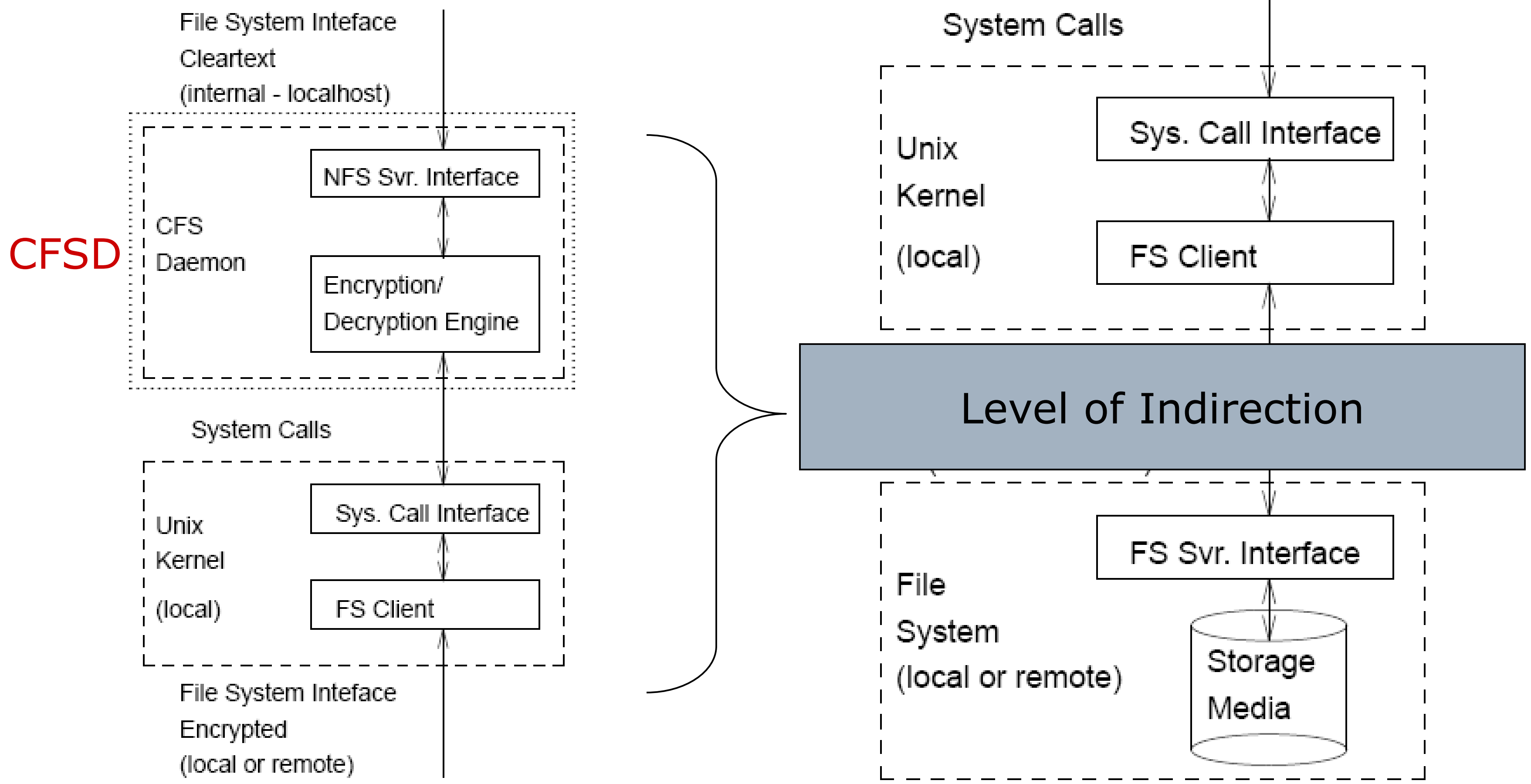
CFS

Data Flow in NFS Client and server

Auth: uid/gid



CFS


File System Inteface
Cleartext
(internal - localhost)
CFSD
CFS
Daemon
NFS Svr. Interface
Encryption/
Decryption Engine
System Calls
Unix
Kernel
(local)
Sys. Call Interface
FS Client
File System Inteface
Encrypted
(local or remote)
User-Level
Application
Any
Program
System Calls
Unix
Kernel
(local)
Sys. Call Interface
FS Client
Level of Indirection
File
System
(local or remote)
FS Svr. Interface
Storage
Media

CFS

- CFSD – a modified NFS server
  - Supports all normal NFS RPCs
  - Provides additional RPCs
  - Accepts RPC from localhost only
- No modification to NFS client
- Start CFSD at boot time
  - Mount /cryptfs
    - A virtual file system

CFS

- Attach a cryptographic key to a directory

```
$ cmkdir /usr/mab/secrets
Key:   (user enters passphrase, which does not echo)
Again:   (same phrase entered again to prevent errors)
$
```

- Directory can be local or remote

CFS

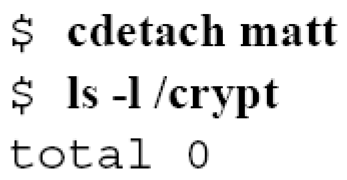
- Attach an encrypted directory

```
$ cattach /usr/mab/secrets matt
Key: (same key used in the cmkdir command)
$ ls -l /crypt
total 1
drwx------ 2 mab 512 Apr 1 15:56 matt
$ echo "murder" > /crypt/matt/crimes
$ ls -l /crypt/matt
total 1
-rw-rw-r-- 1 mab   7 Apr 1 15:57 crimes
```

- Key verified by using a special file in directory encrypted by the hash of the key

CFS

- Detach an encrypted directory



```
$  cdetach matt
$  ls -l /crypt
total 0
```

- Additional commands
  - cname
  - ccat

CFS - Security

- Uses DES in ECB – why ?
- Uses pass phrases
  - Key 1
    - Long Bit Mask (Prevent structural analysis)
  - Key 2
    - Encrypt blocks in ECB mode
- IV
  - Prevent structural analysis across files
  - XORed with each block
  - No Chaining
  - Stored in GID (High security mode)

CFS - Security

- Filenames are encrypted and encoded in ASCII
  - increases size of file names
- An attach can be marked "obscure"
  - security through obscurity
- File sizes, access times and structure of directory hierarchy is not encrypted

CFS – Performance

- Data is copied several extra times

```
Application
  -> kernel
    -> CFS daemon (User Layer)
      -> back to the kernel
        -> underlying file system.
```

- No write cache, only read caches

TCFS – Transparent CFS

- Implemented as a modified kernel-mode NFS client
  - Kernel module recompilation required
  - User level tools recompilation required

TCFS

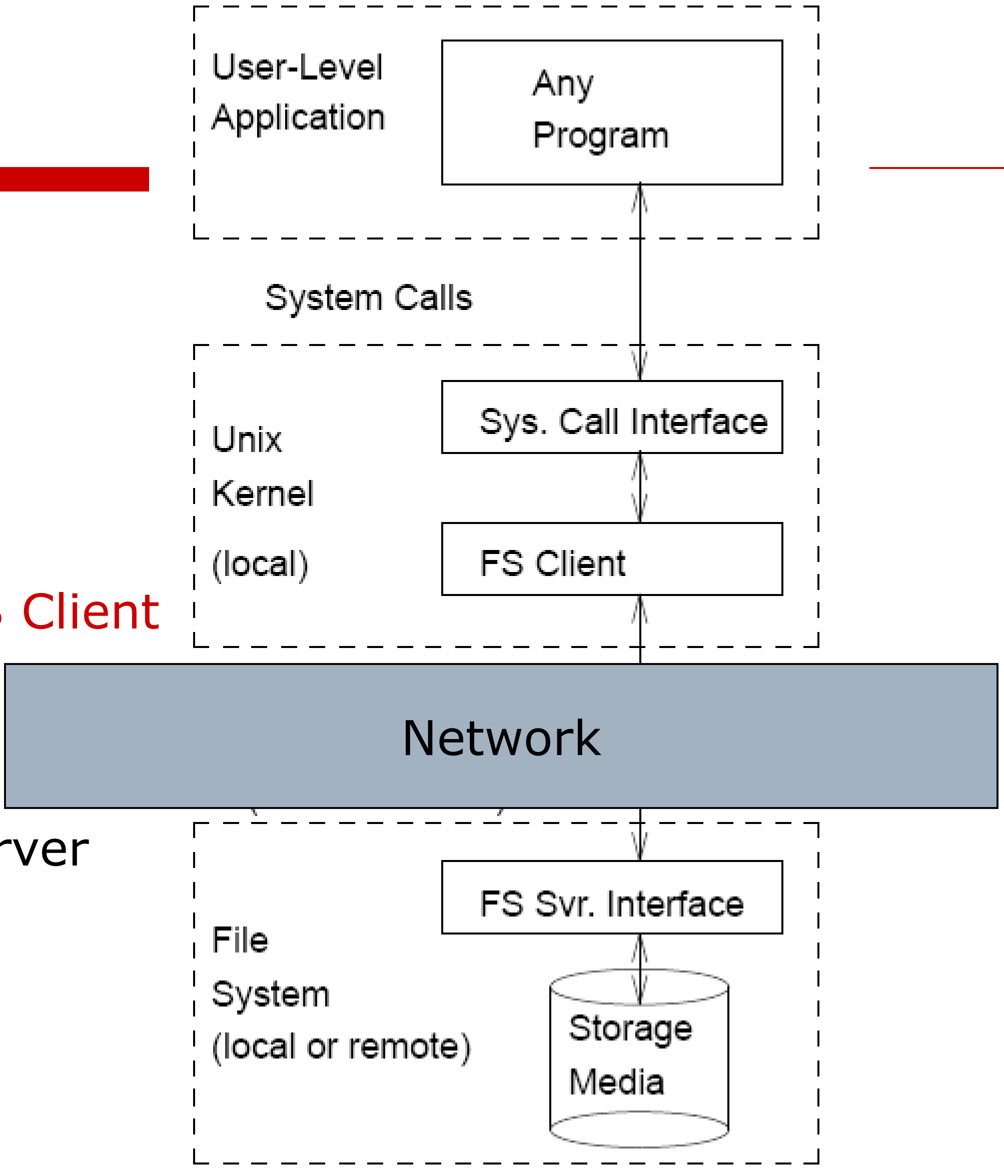
mountd
ioctl

TCFS NFS Client

RPC
XDR

NFS Server

xattrd



TCFS - Operation

- Server exports a directory
  - /etc/exports

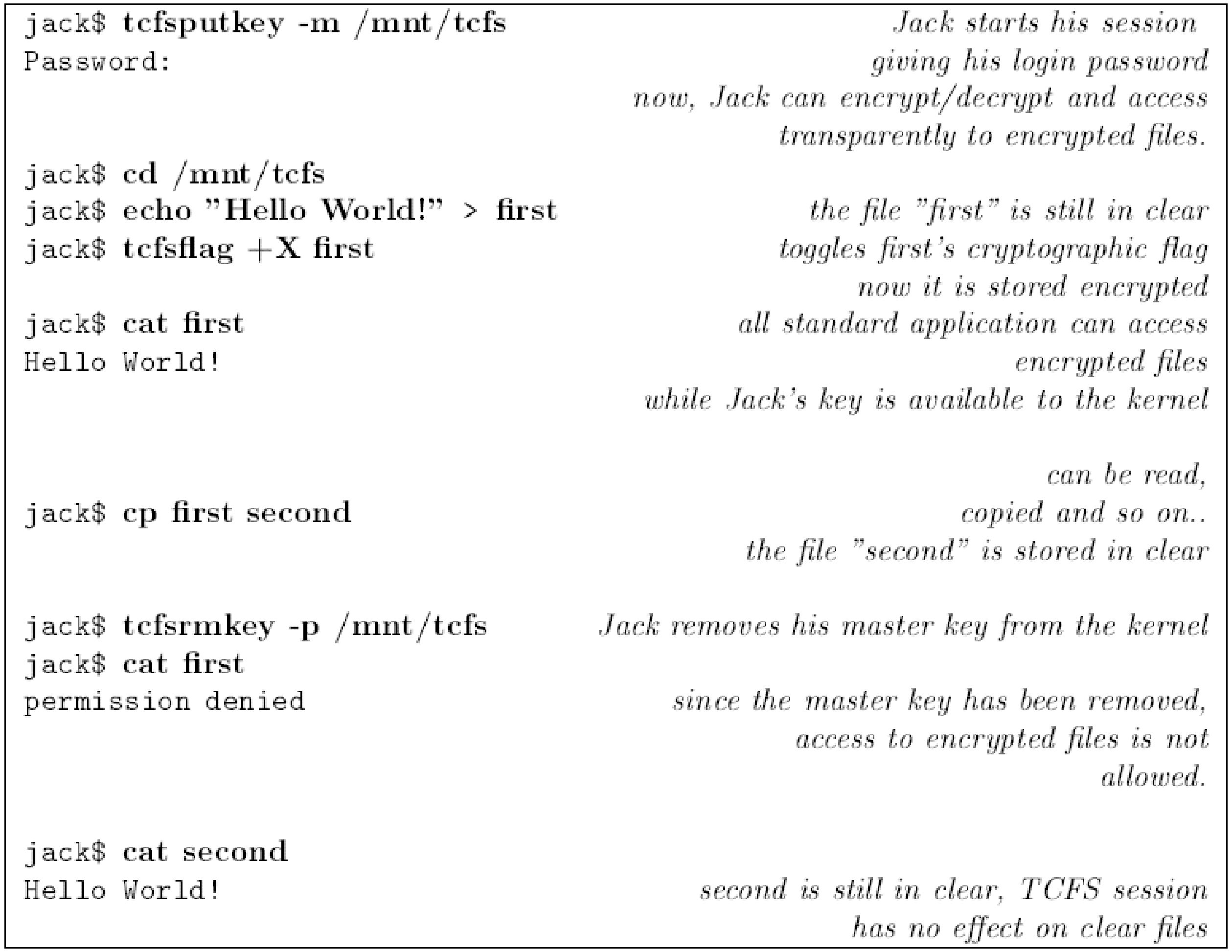
```
/exports bar(rw,insecure)
```

- NFS server not modified
- Client mounts a remote dir with type "tcfs"

```
mount -t tcfs foo:/exports /mnt/tcfs
```

- A modified mount command in nfs-utils
- Encrypted files are set with special attribute
  - A modified xattrd
- User master key must be set to access files

TCFS - Operation

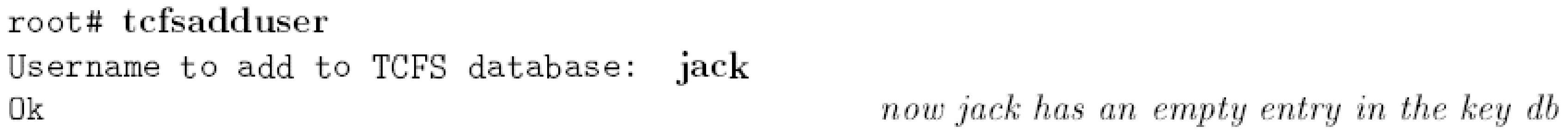


```
jack$ tcfsputkey -m /mnt/tcfs                              Jack starts his session
Password:                                                giving his login password
                                              now, Jack can encrypt/decrypt and access
                                                    transparently to encrypted files.

jack$ cd /mnt/tcfs
jack$ echo "Hello World!" > first                        the file "first" is still in clear
jack$ tcfsflag +X first                               toggles first's cryptographic flag
                                                            now it is stored encrypted
jack$ cat first                                      all standard application can access
Hello World!                                                           encrypted files
                                              while Jack's key is available to the kernel

                                                                        can be read,
jack$ cp first second                                               copied and so on..
                                                    the file "second" is stored in clear

jack$ tcfsrmkey -p /mnt/tcfs             Jack removes his master key from the kernel
jack$ cat first
permission denied                               since the master key has been removed,
                                                     access to encrypted files is not
                                                                            allowed.


jack$ cat second
Hello World!                                        second is still in clear, TCFS session
                                                              has no effect on clear files
```

TCFS – Key Management

- Raw key management
  - New ioctls recognized by client
  - Provides basis for other schemes
- Basic Key Management
  - The key database

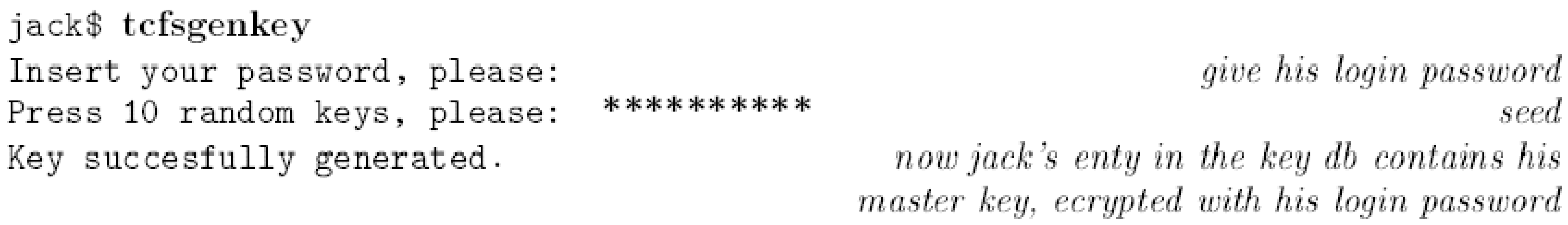
    /etc/tcfspwdb
  - sysadmin registers a user



```
root# tcfsadduser
Username to add to TCFS database:  jack
Ok                                                    now jack has an empty entry in the key db
```

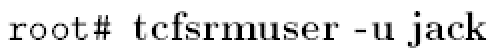
TCFS – Key Management

- User creates a master key



```
jack$ tcfsgenkey
Insert your password, please:                                    give his login password
Press 10 random keys, please:  **********                                           seed
Key succesfully generated.                       now jack's enty in the key db contains his
                                               master key, ecrypted with his login password
```

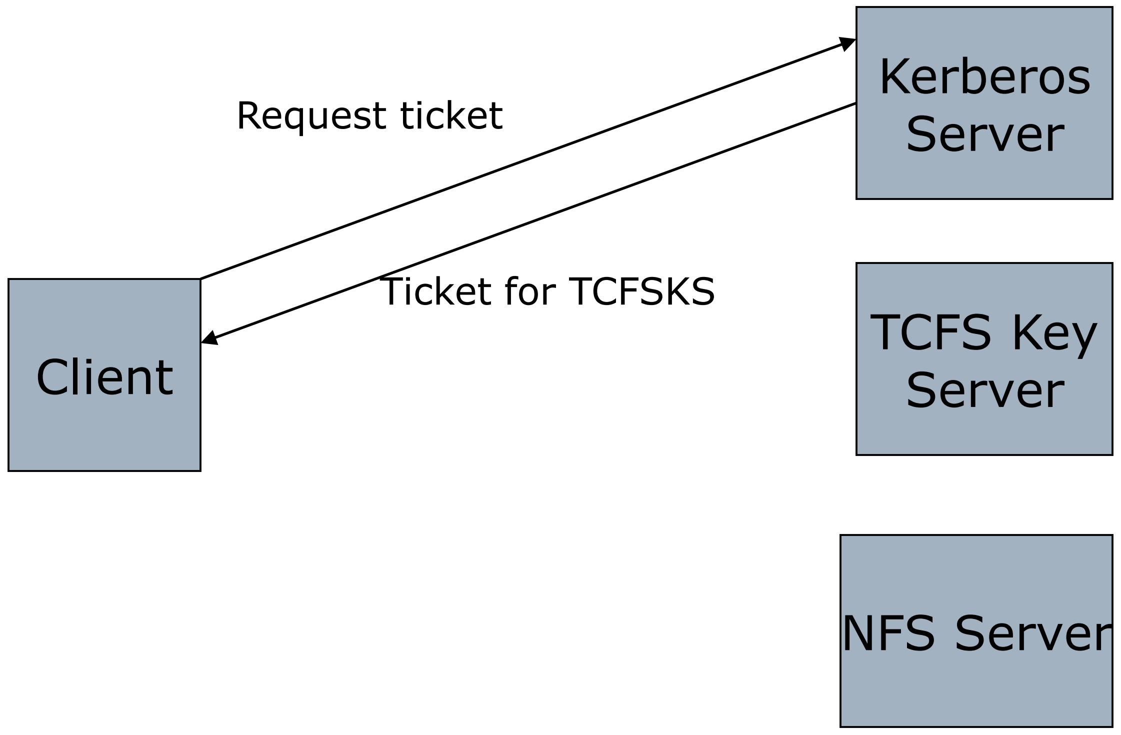
- sysadmin can remove a user



```
root# tcfsrmuser -u jack
```

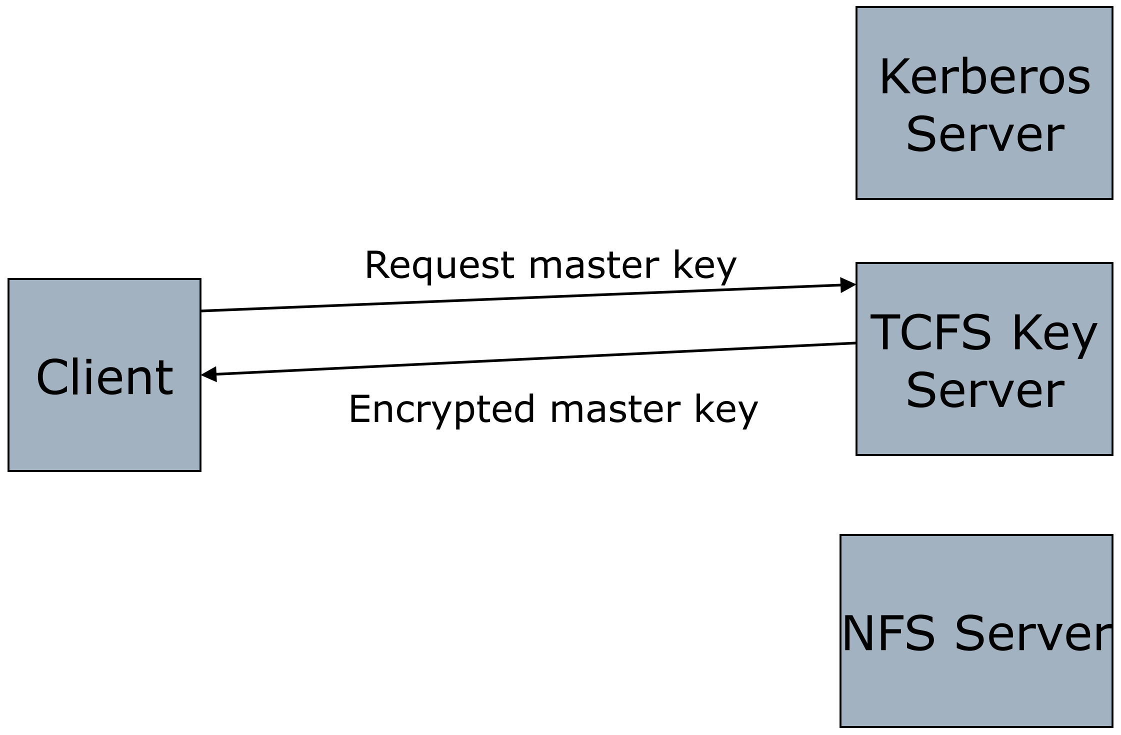
TCFS – Key Management

- The Kerberized Key Management Scheme



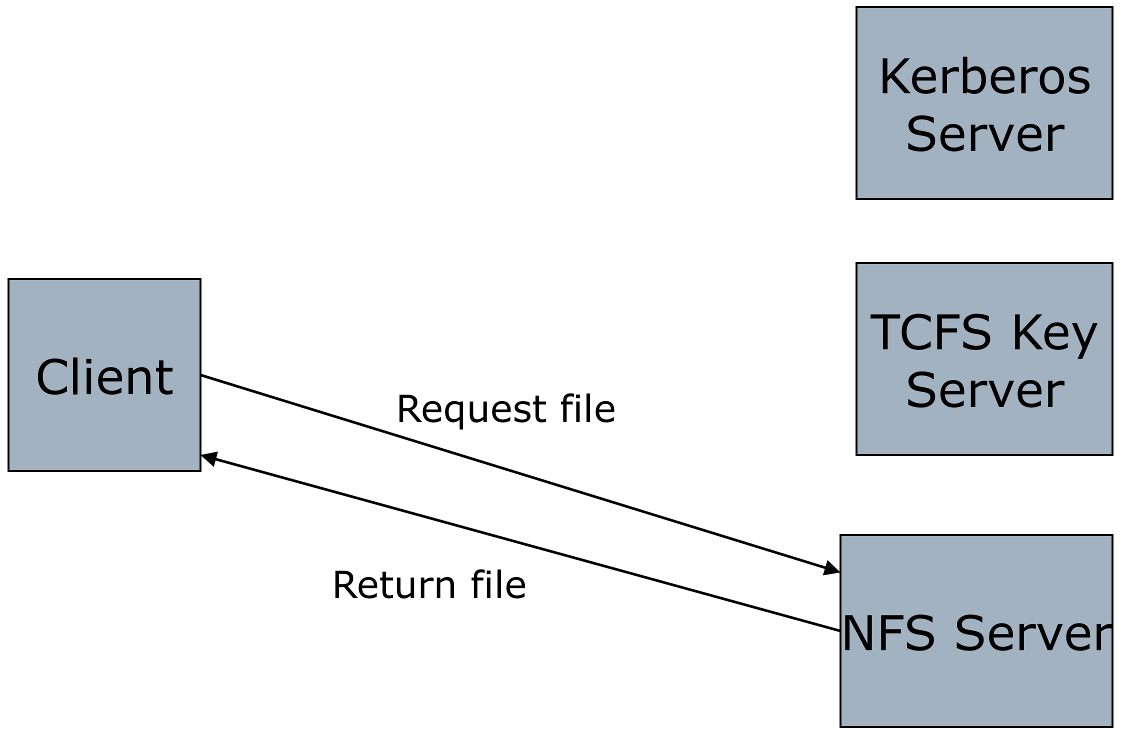
TCFS – Key Management

- The Kerberized Key Management Scheme



TCFS – Key Management

- The Kerberized Key Management Scheme



TCFS – Key Management

- Group/Threshold Sharing
  - Similar to secret splitting
  - sysadmin creates a group

```
tcfsaddgroup –g <group>
```

    - # of users
    - name of users
    - threshold
    - password
  - User activates a group

```
tcfsputkey –g <group>
tcfsrmkey –g <group>
```

TCFS - Encryption

- Multiple cipher support
- File specific key
- File header
  - file specific key
  - cipher
- Block encryption
  - block key
    - Hash(File Key || Block no)
  - Protection against structural analysis
  - Authentication tag
    - Hash(Block data || block key)
    - Detect data change/swap

TCFS - Encryption

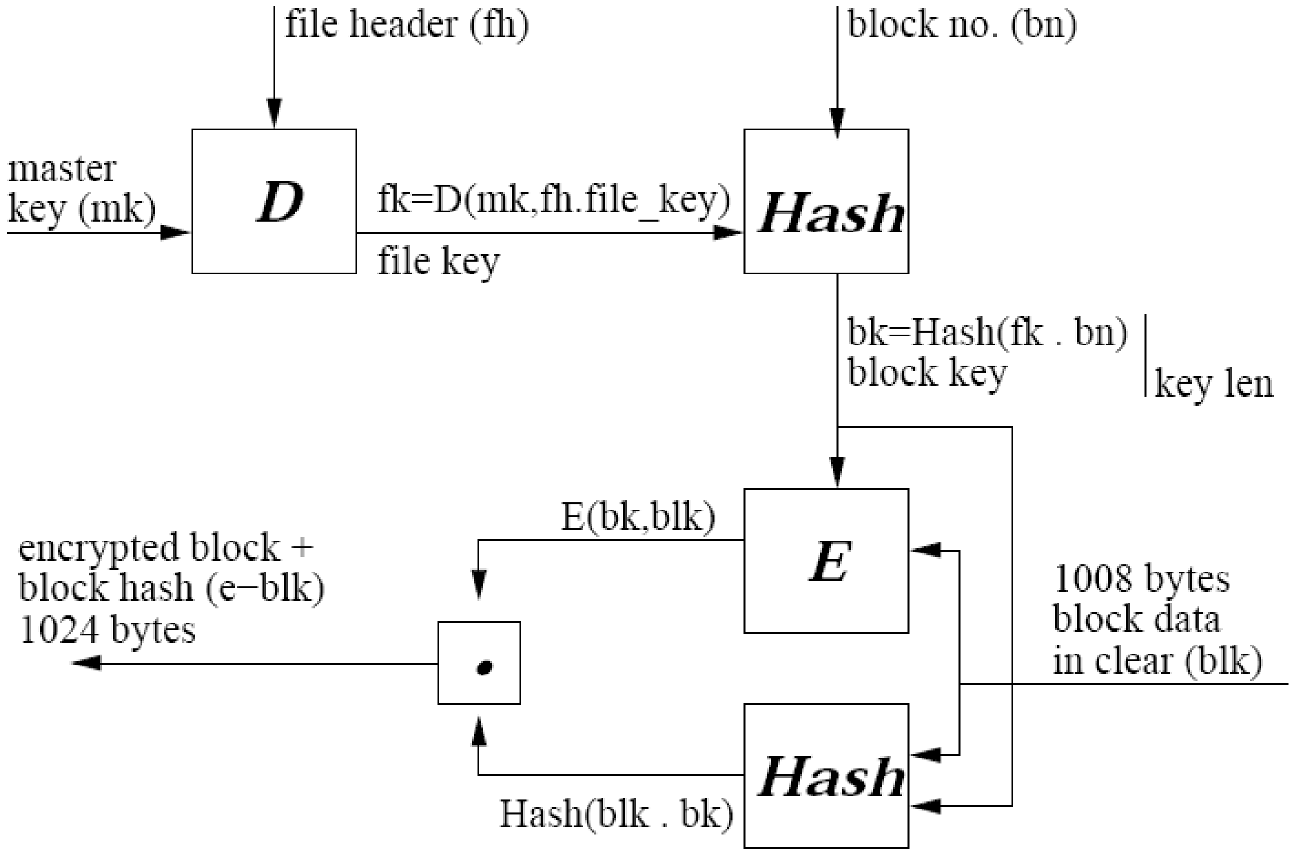
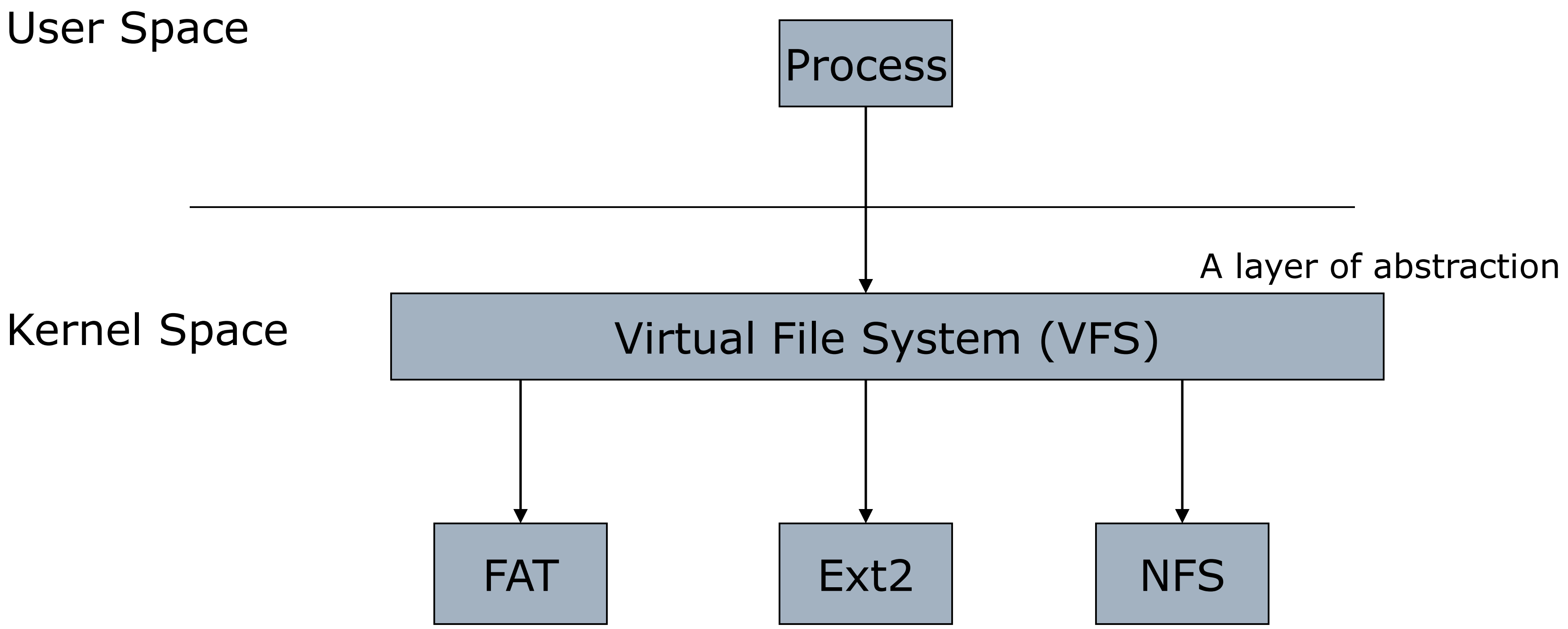


Figure 3: Encryption of blocks in TCFS

TCFS - Performance

- Less overhead than CFS
  - data copied fewer times
- Random access is slower
- RTT for remote attribute checking makes is slower than vanilla NFS

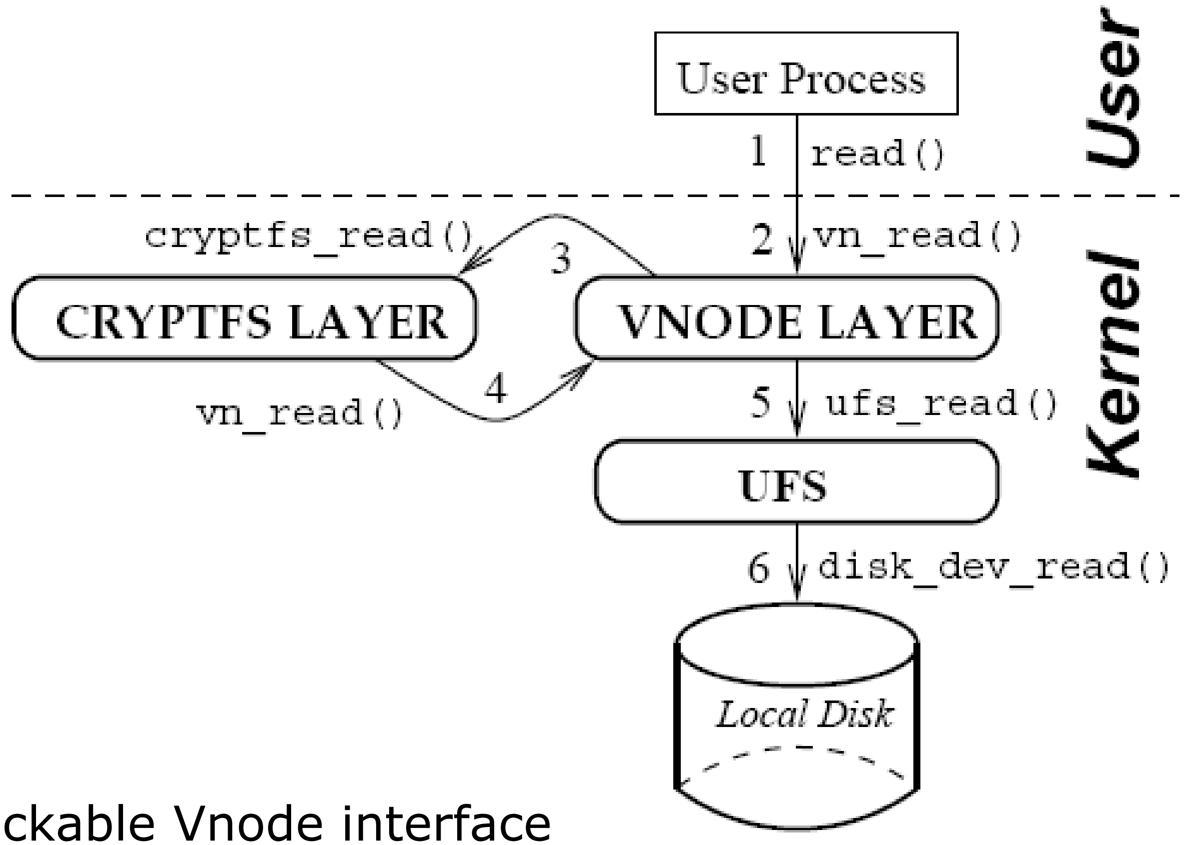
Cryptfs: A Stackable Vnode Level Encryption File System



Cryptfs

- VNodes
  - open file, directory, device, socket
  - Higher layers access all entities uniformly
- VNode stacking
  - Modularize file system functions

Cryptfs



A stackable Vnode interface

Cryptfs – Key Management

- Root mounts an instance of Cryptfs
- User passphrases
- User Key = MD5Hash(passphrases)
- Special ioctl to manage keys
  - set/reset/delete keys
- Two modes of operation
  - Key lookup on user id alone

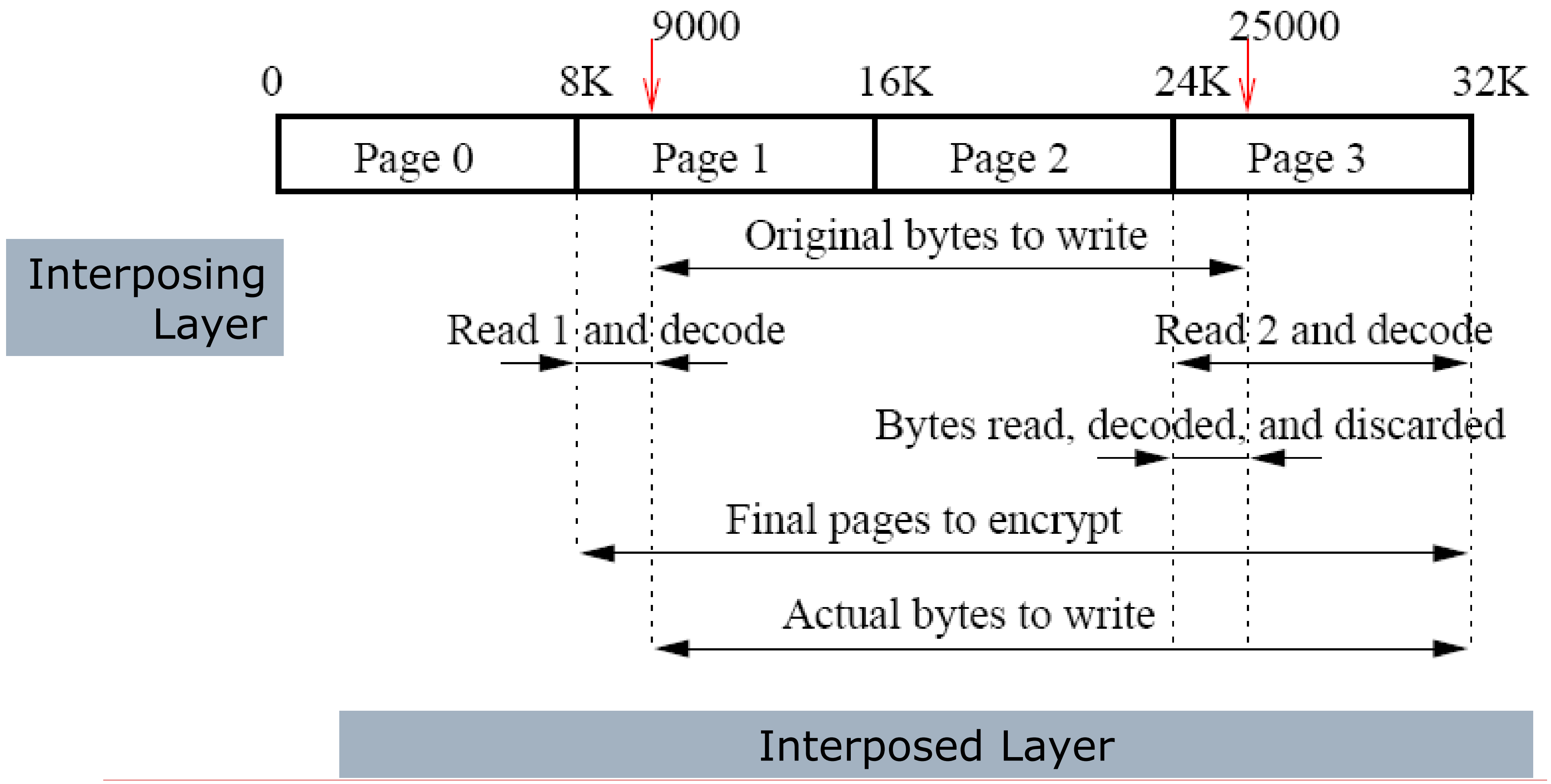
Cryptfs – Key Management

- Key lookup on <user id, session id>
    - What is a session? Unix sessions!
    - Protected again user account compromise
- Keys associated with real UID, not effective ones
- Groups
    - Decouple from unix groups
    - Must share the key
    - Use multiple keys in different sessions

Cryptfs – Security

- block size = page size
- Cipher: Blowfish
  - Does not change the size of file
- Mode: CBC
  - Only inside a block/page
  - Limits dependency between blocks
  - Allows random access
- One IV per mount
- No file specific key
- Encrypt file and directory names
  - uuencode
    - 3 bytes of binary = 4 bytes of ascii (44-111)
    - File names become 33% longer
  - Checksums for filenames

Cryptfs: write bytes 9000-25000



Cryptfs

- Works on top of any native FS
- No other daemons required
- Portable
  - Exceptions
    - Exporting symbols
    - Modifications to FS data structure
- Kernel resident
  - Kernel memory is difficult to get at
    - vs.:CFS stores in user level memory
  - Fewer context switches than CFS and TCFS

NCryptfs

- Advanced version of Cryptfs
- Attachments
  - A single mount operation

    under “/mnt/ncryptfs”
  - “Attach” an encrypted directory

```
nc_attach -c blowfish /mnt/ncryptfs mail /home/kvthanga/mail
% Enter key:
```

NCryptfs

Mounts	Attaches
Done by the superuser - modify /etc/fstab	Can be done by any user - A light weight mount
Encrypted directories can be mounted on any other directory	Attaches are created only under /mnt/ncryptfs
May execute many mount commands	One mount to mount /etc/ncryptfs
Directory mounted on must already exist	No directories or files can be created on /etc/ncryptfs - Entries created in dcache
May hide underlying dirs	Does not hide any underlying data
OS have hard limits for mounts	No limits

NCryptfs

- Attachments
  - Encryption key
  - Authorizations
  - Active Sessions

NCryptfs

- Encryption key
  - Long lived key for
    - Data
    - File names
      - checksums
  - No file specific key
  - Created from hash of user passphrase
  - Key related data is “pinned” in memory
    - Pages with keys are not swapped
  - Support multiple ciphers
  - CFB - Cipher feedback mode of operation
    - File size does not change

NCryptfs

- Players
  - System Administrator
    - Mounts NCrytpfs
    - Installs the NCryptfs kernel and user-space components
  - Owners
    - Controls encryption key
    - Delegates access rights
  - Reader & Writers
    - Don’t have the encryption key

NCryptfs

- Authorizations
  - Gives an entity access to an attach
  - Entity
    - process, session, user or group
  - Create an authorization
    - Entity selects a passphrase
    - Sends salted MD5 hash of it to owner
      - Entity does not have to share passphrase with owner
      - What is a salted MD5 hash?
    - Owner adds hash to configuration file

NCryptfs

- Use an authorization

```
nc_auth /mnt/ncryptfs mail
```

  - Creates a session

- Active sessions
  - Entity
  - Permissions granted to the entity - bitmask
    - Unix permissions
      - Read, Write, Execute

NCryptfs

- Detach
- Add an Authorization
- List Authorizations
- Delete an Authorization
- Revoke an active session
- List active sessions
- Bypass VFS Permissions

NCryptfs

- Attach access control
  - Attach – default everyone
  - Authentication
- Attach names
  - User specified
  - NCryptfs
    - u<userid>s<sessionid>
    - Random name
      - Prevents namespace clash

NCryptfs

- Groups
  - Supports native groups
    - has to be setup ahead of time
  - Support ad-hoc groups
    - still need permission to modify low level objects
      - Use Bypass VFS permission

NCryptfs

Bypass VFS permission

```
                      sys_unlink {                      /* system call service routine */
                        vfs_unlink {                                  /* VFS method */
                          call nc_permission()
                          if not permitted: return error
                          nc_unlink {                             /* NCryptfs method */
current->fsuid = owner's    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 */
Restore(current->fsuid)     call nc_perm_fixup()                    /* code we added */
                          }                                  /* end of nc_unlink */
                        }                                 /* end of outer vfs_unlink */
                      }                                      /* end of sys_unlink */
```

NCryptfs

- Timeouts
  - Active sessions
    - permission denied
    - new file opens fail
    - new file open suspends process until re-authentication
    - all operations suspend process until re-authentication
  - Authorizations
    - new uses can’t create new sessions
    - old sessions may continue
  - Keys
    - key is deleted or
    - use denied for new files
  - User space timeout callbacks

NCryptfs

- Revocation
  - Similar to timeout
  - Can re-authenticate
- Portability
  - Modification to task structure
    - On-exit callbacks
      - delete keys
      - memory resources
    - Challenge response authentication
  - Cache clearing

eCryptfs from IBM

- ☐ Motivation/ Problem
- ☐ History and Overview
- ☐ eCryptfs solutions
- ☐ Design overview
- ☐ Design Details
- ☐ Key management
- ☐ VFS operations
- ☐ Using eCryptfs
- ☐ Future enhancements

Motivation

- Confidentiality when outside host operating environment.
- Easy to use secure data store.
- Convenient backup procedures.
- Key retrieval.
- Intuitive – minimal learning by users.
- Policies and owners.
- Cost of technology and adoption.
- Knowledge and extent of risks

History/ Overview

- Derived from Erez Zadok’s cryptfs (FIST framework).
- Part of Linux from version 2.6.19 onwards.
- Encryption at file level.
- File contains metadata for decryption.
- Native kernel FS (POSIX)- no need for patches.
- Seamless security - data encryption on the fly
- Seamless key mgmt - Linux kernel keyring.
- Incremental development – current ver 0.1.

Why a new thing ?

- extends Cryptfs to provide advanced key management and policy features
- stores cryptographic metadata in the header of each file written, so that ***encrypted files can be copied between hosts***
- the file will be decryptable with the proper key, and there is no need to keep track of any additional information aside from what is already in the encrypted file itself.

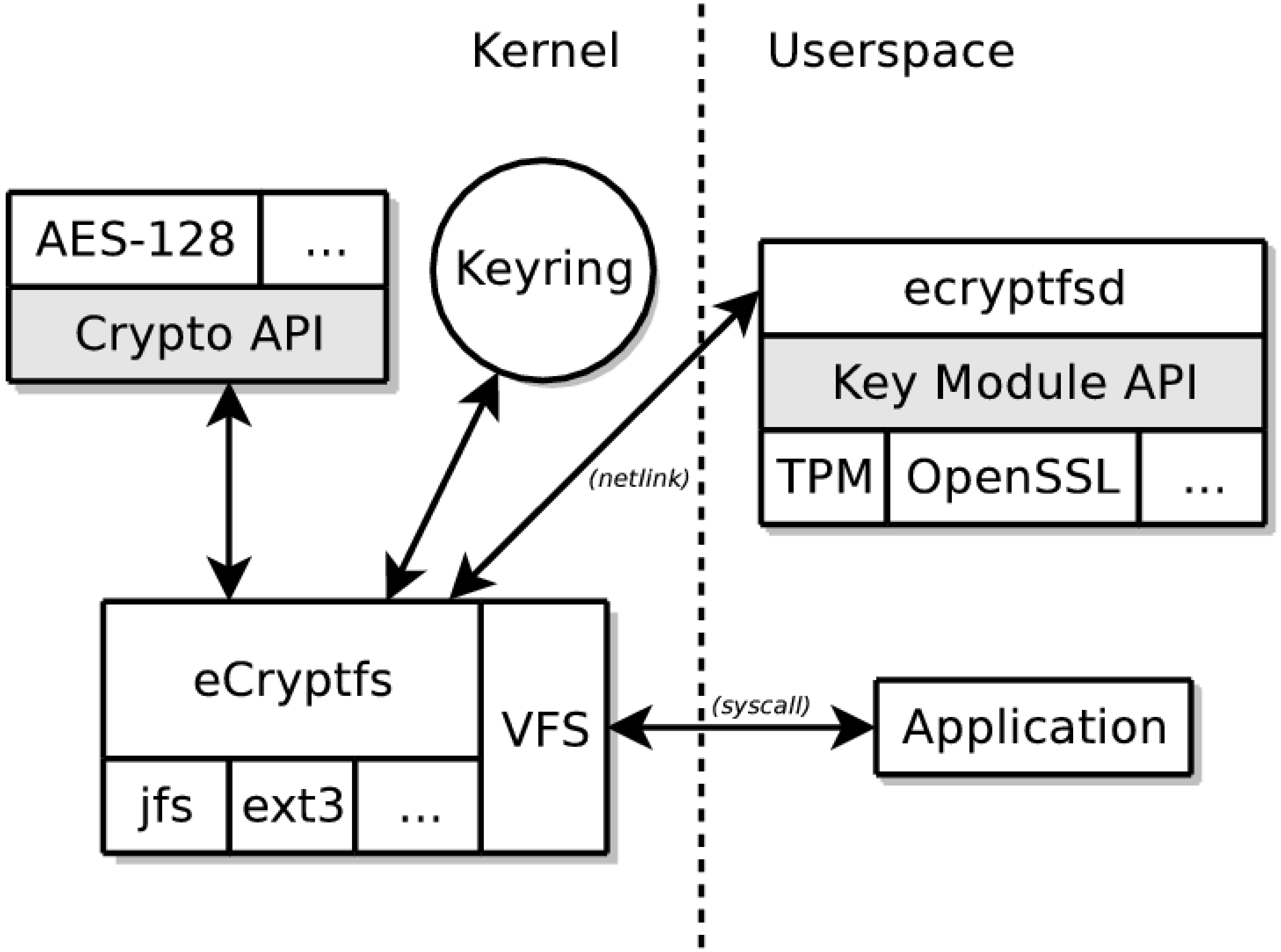
eCryptfs from IBM

- ☐ Motivation/ Problem
- ☐ History and Overview
- ☐ eCryptfs solutions
- ☐ Design overview
- ☐ Design Details
- ☐ Key management
- ☐ VFS operations
- ☐ Using eCryptfs
- ☐ Future enhancements

eCryptfs solutions

- Confidentiality - Integration of security into FS (Lotus Notes analogy of secure transmission)
- Ease of deployment – No kernel modifications, No separate partition, per-file meta data
- TPM utilization- generate key pair for session key encryption.
- Key Escrow usage. (Author’s suggestion)
- Easy Incremental backups.
- Lower File System independent.

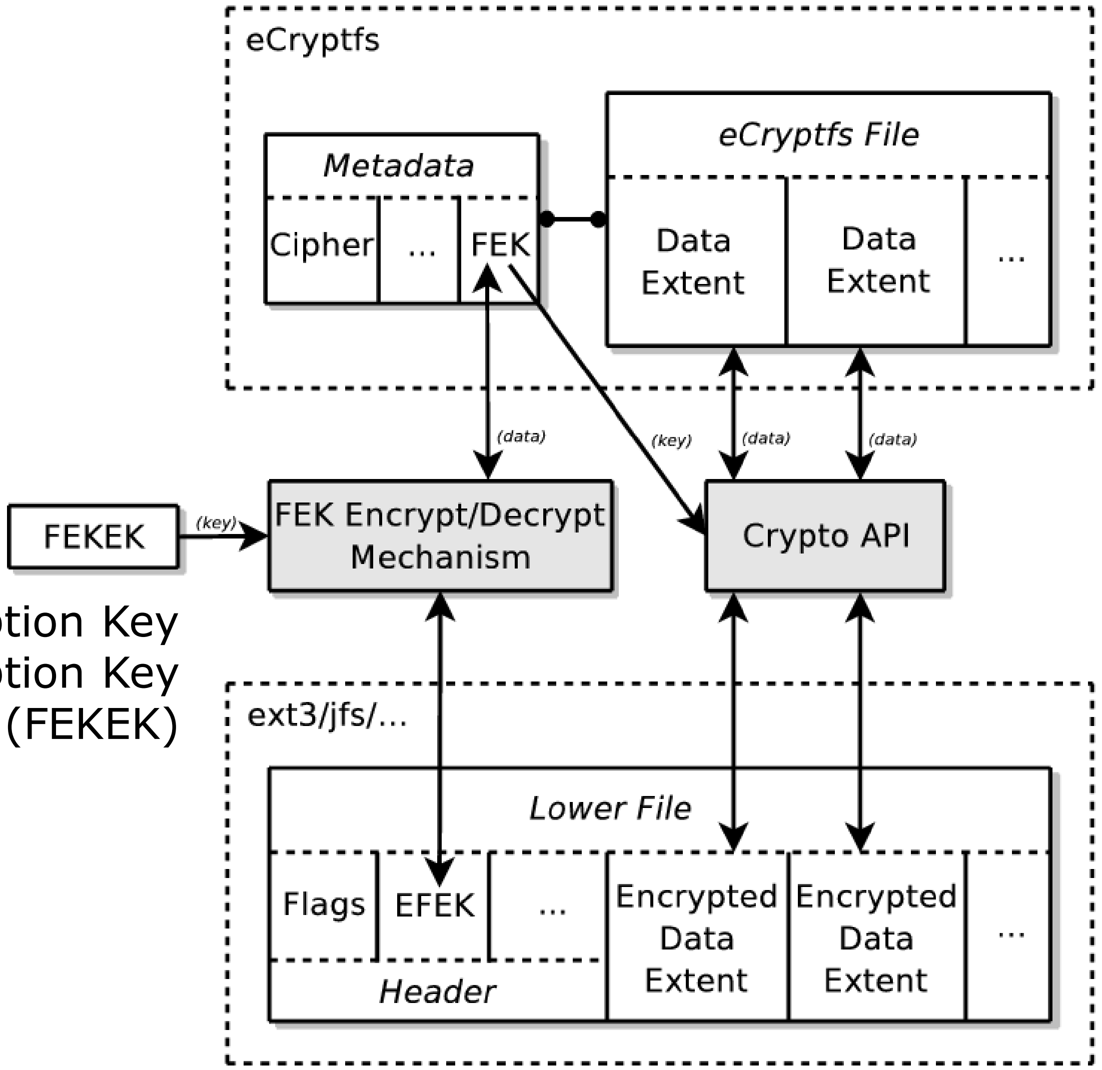
Design overview



eCryptfs from IBM

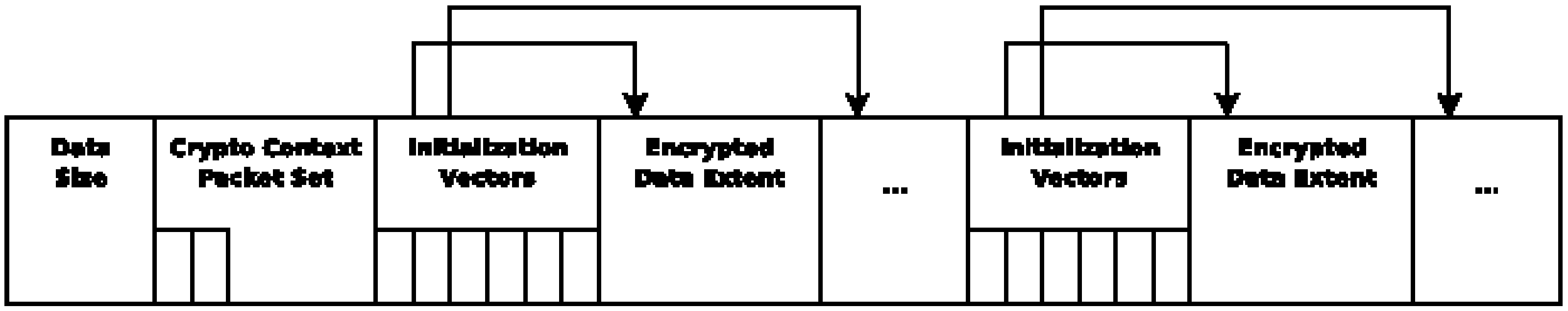
- ☐Motivation/ Problem
- ☐History and Overview
- ☐eCryptfs solutions
- ☐Design overview
- ☐Design Details
- ☐Key management
- ☐VFS operations
- ☐Using eCryptfs
- ☐Future enhancements

Details: enc/decrypt individual data extents



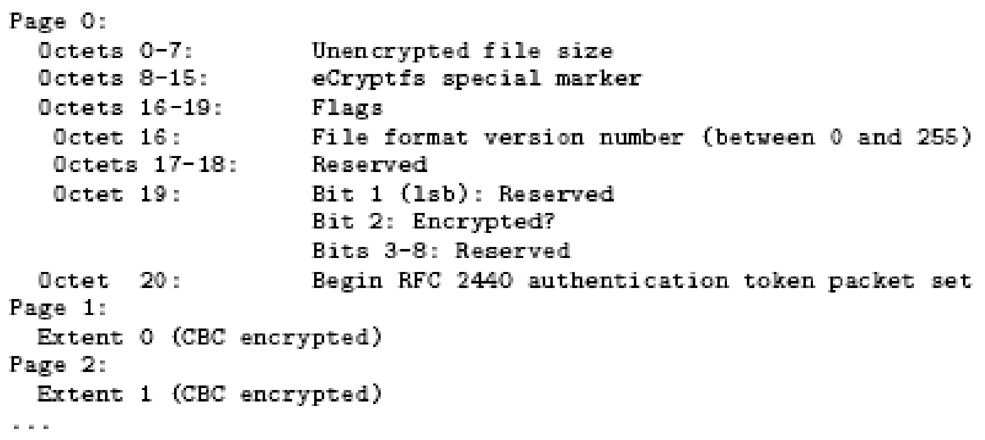
File Encryption Key
Encryption Key
(FEKEK)

Design Details



- File format – Follows OpenPGP format
  - Deviation for PGP – Encryption on extents
  - Each extent has unique IVs.
  - Some extents contain only IVs for data extents
  - Sparse file support – fill encrypted 0s
  - CBC block cipher for extents

Design Details (Contd..)



```
Page 0:
  Octets 0-7:         Unencrypted file size
  Octets 8-15:        eCryptfs special marker
  Octets 16-19:       Flags
   Octet 16:          File format version number (between 0 and 255)
   Octets 17-18:      Reserved
   Octet 19:          Bit 1 (lsb): Reserved
                      Bit 2: Encrypted?
                      Bits 3-8: Reserved
  Octet  20:          Begin RFC 2440 authentication token packet set
Page 1:
  Extent 0 (CBC encrypted)
Page 2:
  Extent 1 (CBC encrypted)
...
```

PGP File header format

- File format (contd)
  - Page 0– Header, Page 1-n: Data + Extent.
  - Bytes 0-19- Standard information for file.
  - Marker– 32 bit number for uniquely identification
  - Byte 20 onwards
    - Set of all authentication tokens for the file
    - Encrypted File Encryption Key

Design Details (Contd..)

- Kernel Crypto API
  - In kernel encryption – faster
  - Any symmetric cipher supported by cryptoAPI
- IV (Initialization Vector)
  - Avoid risk of cryptanalysis- unique IV for extents
  - Initial IV – MD5 sum of file encryption key (K_R)
- Integrity verification
  - Keyed hash over extents using $K_{R.}$
  - Generate hash whenever data changes
  - Verify during read, assert hash verifies.

Design Details (Contd..)

- In-memory Cryptographic Context - Stored in user session’s keyring.
  - Session key for the file.
  - Encryption status.
  - crypto API context – cipher, key size, etc
  - Size of the extents.
- Key revocation
  - Acquire the passphrase and the session key from it.
  - Regenerate a new session key and encrypt all data once again.

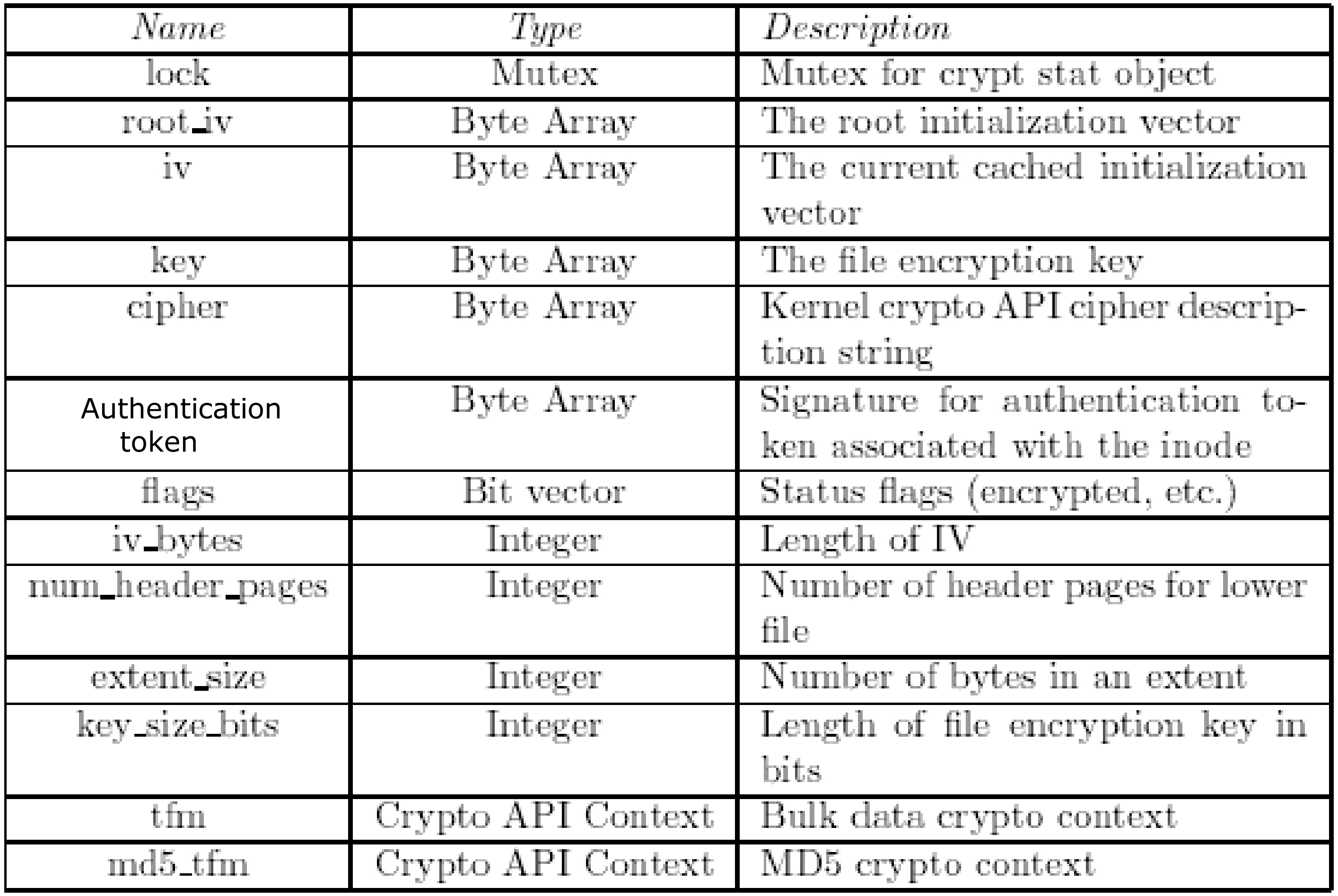
Design Details (Contd..)

- Is a stackable FS
  - Does not write directly onto block device.
  - Each VFS object maps onto a lower object.
  - Any POSIX compliant FS can act as a lower FS.
- VFS objects' private data holds:
  - The reference to lower objects.
  - Current context required for encryption/ decryption.

eCryptfs from IBM

- Motivation/ Problem
- History and Overview
- eCryptfs solutions
- Design overview
- Design Details
- Key management
- VFS operations
- Using eCryptfs
- Future enhancements

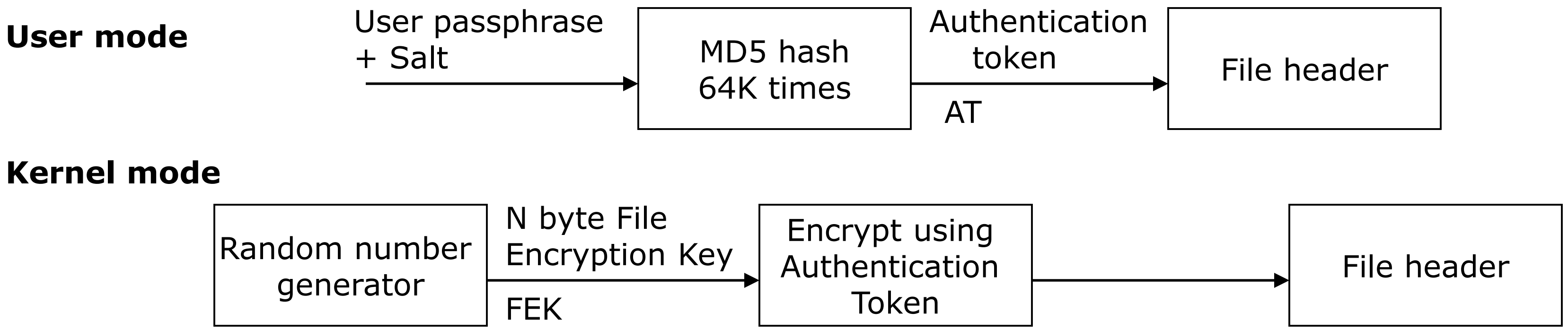
In memory context in the inode



Name	*Type*	*Description*
lock	Mutex	Mutex for crypt stat object
root_iv	Byte Array	The root initialization vector
iv	Byte Array	The current cached initialization vector
key	Byte Array	The file encryption key
cipher	Byte Array	Kernel crypto API cipher description string
Authentication token	Byte Array	Signature for authentication token associated with the inode
flags	Bit vector	Status flags (encrypted, etc.)
iv_bytes	Integer	Length of IV
num_header_pages	Integer	Number of header pages for lower file
extent_size	Integer	Number of bytes in an extent
key_size_bits	Integer	Length of file encryption key in bits
tfm	Crypto API Context	Bulk data crypto context
md5_tfm	Crypto API Context	MD5 crypto context

Key management

- Supports all ciphers and key sizes of cryptoAPI
- Default AES-128



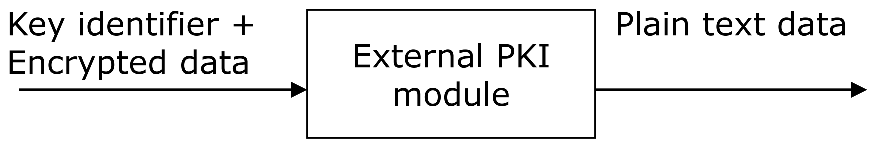
Key management (Contd…)

- Encryption
  - Authentication token found in keyring after mount.
  - FEK encrypted with each user’s AT and stored in header.
  - Authentication token of each user stored in header

- Decryption:
  - Authentication token matched with each token in header
  - File Encryption Key decrypted with proper AT and stored in keyring – Support for multiple users

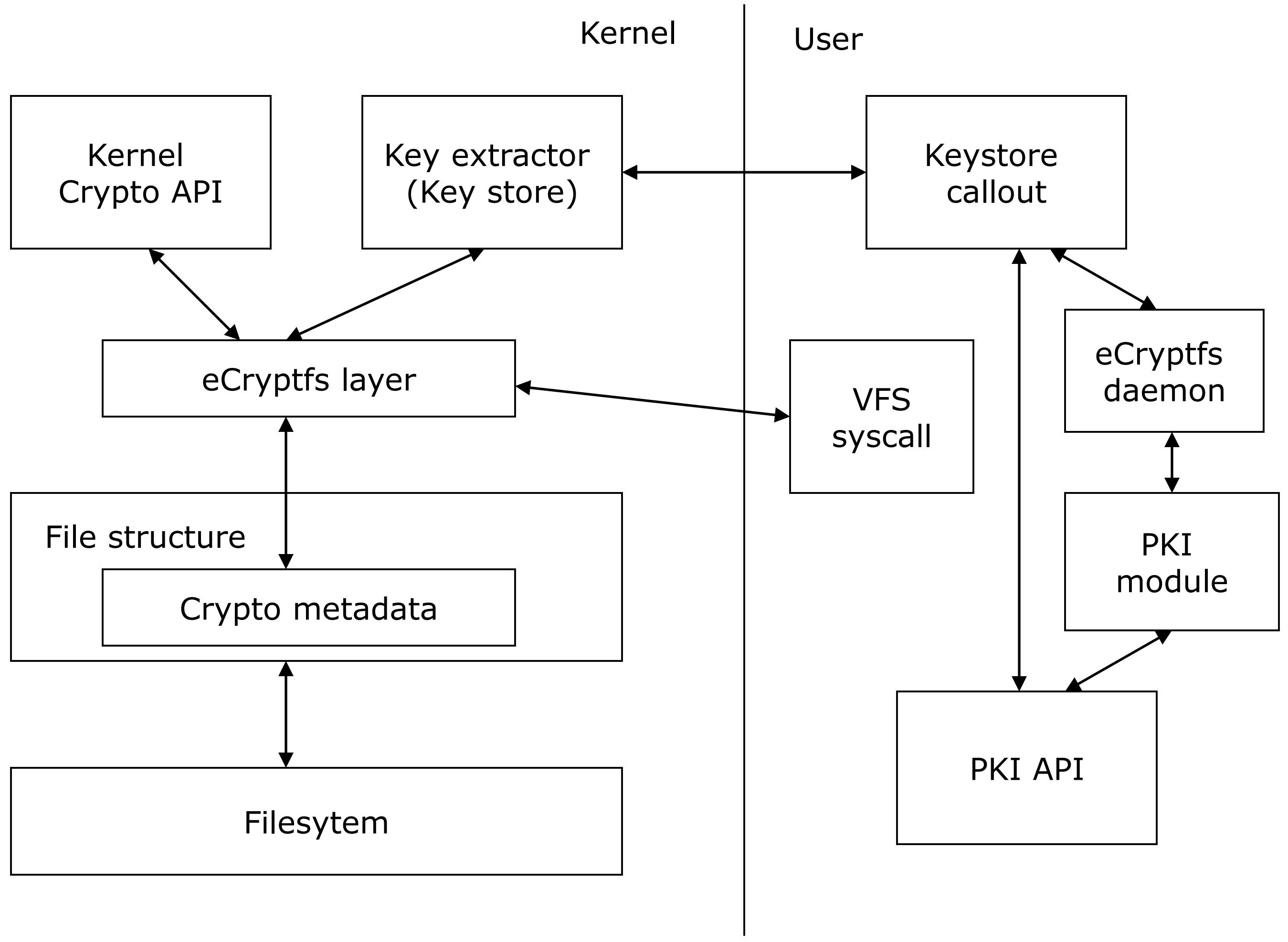
Key management (Contd..)

- Pluggable Authentication Module – Configure ways to authenticate the user (generate token)
  - Passphrase (salted)- Stored in keyring
  - Use passphrase to extract public key
  - Use this derived key in combination with key from TPM
  - Use a smart card or USB to store the key
- Pluggable PKI Module – use x509 certificates, revocation lists etc and manage keys better



Authenticates the identifier
and decrypts the data if valid

Key Callout, eCryptfs Daemon

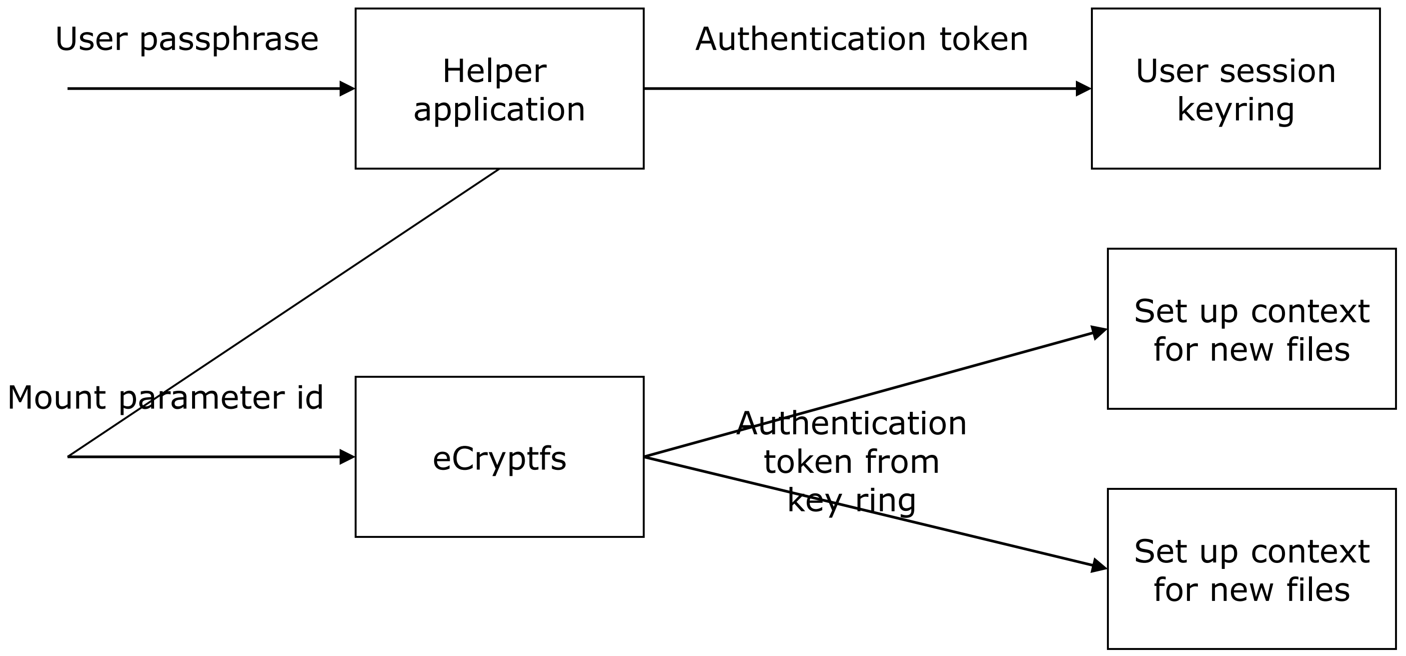


Key management (Contd..)

- Key Callout
  - Means of communication between kernel and user module – Parses policy information on target
  - Finds passphrase or public keys of users
- eCryptfs Daemon
  - Means to get to the user X-session if need to be prompted for a passphrase
- Key Escrow
  - A centralized trusted party stores all keys
- Secret sharing/ splitting
  - In a dynamic environment, this could be used for a balance between key secrecy and sharing

VFS Operations (version 0.1)

- Mount



VFS Operations (Contd..)

- File Open – Existing file
  - Validate the unique eCryptfs marker
  - Match the Authentication token
  - Decrypt File Encryption Key
  - Root IV = N bytes of MD5(File Encryption Key)
  - Update the context in the inode with
    - File Encryption key
    - Key size
    - Cipher name
    - Root IV
    - Number of header pages and extent size

VFS Operations (Contd..)

- File Open – New file
  - Generate a File Encryption Key in kernel
  - Fill inode context
    - Cipher name – AES 128
    - Root IV – N bytes of MD5(File Encryption Key)
    - Header page – 1, extent size – kernel page size
  - Initialize the kernel crypto API context for the file
    - CBC mode
  - Get Authentication token, Encrypt FEK with it
  - Header to be written to disk on close

VFS Operations (Contd..)

- Page Read/ Write
  - File is open and inode contains relevant context
  - Lower page index= index + Num of header pages
  - IV = Root IV + page index
  - Fetch the key and cipher used from context
  - Calculate the extent boundaries for operation
  - Set up state to be used by crypto API
  - Read – Disk -> Encrypted page + context -> crypto API -> Clear text page -> Caller
  - Write – Caller -> Clear text page + context -> crypto API -> Encrypted text page -> Disk

VFS Operations (Contd..)

- File truncation
  - File size updated in header
  - Write encrypted 0s after new EOF
- File Append
  - Translated into write to the appropriate page in the lower file
- File Close
  - Free up associated VFS objects
  - If new file, write the header on disk
  - Existing file, no change to the on disk header

eCryptfs from IBM

- ☐Motivation/ Problem
- ☐History and Overview
- ☐eCryptfs solutions
- ☐Design overview
- ☐Design Details
- ☐Key management
- ☐VFS operations
- ☐Using eCryptfs
- ☐Future enhancements

Using eCryptfs

- Linux Journal article dated 04/01/07 – Detailed usage instructions
  - Sample usage

    #modprobe ecryptfs - Load the module

    #mount –t ecryptfs /sec /sec – overlay mount

    Enter passphrase:

    Enter cipher:

    #cat “Hello world” > secret.txt

  - PKI modules can be selected by mount options for public key support

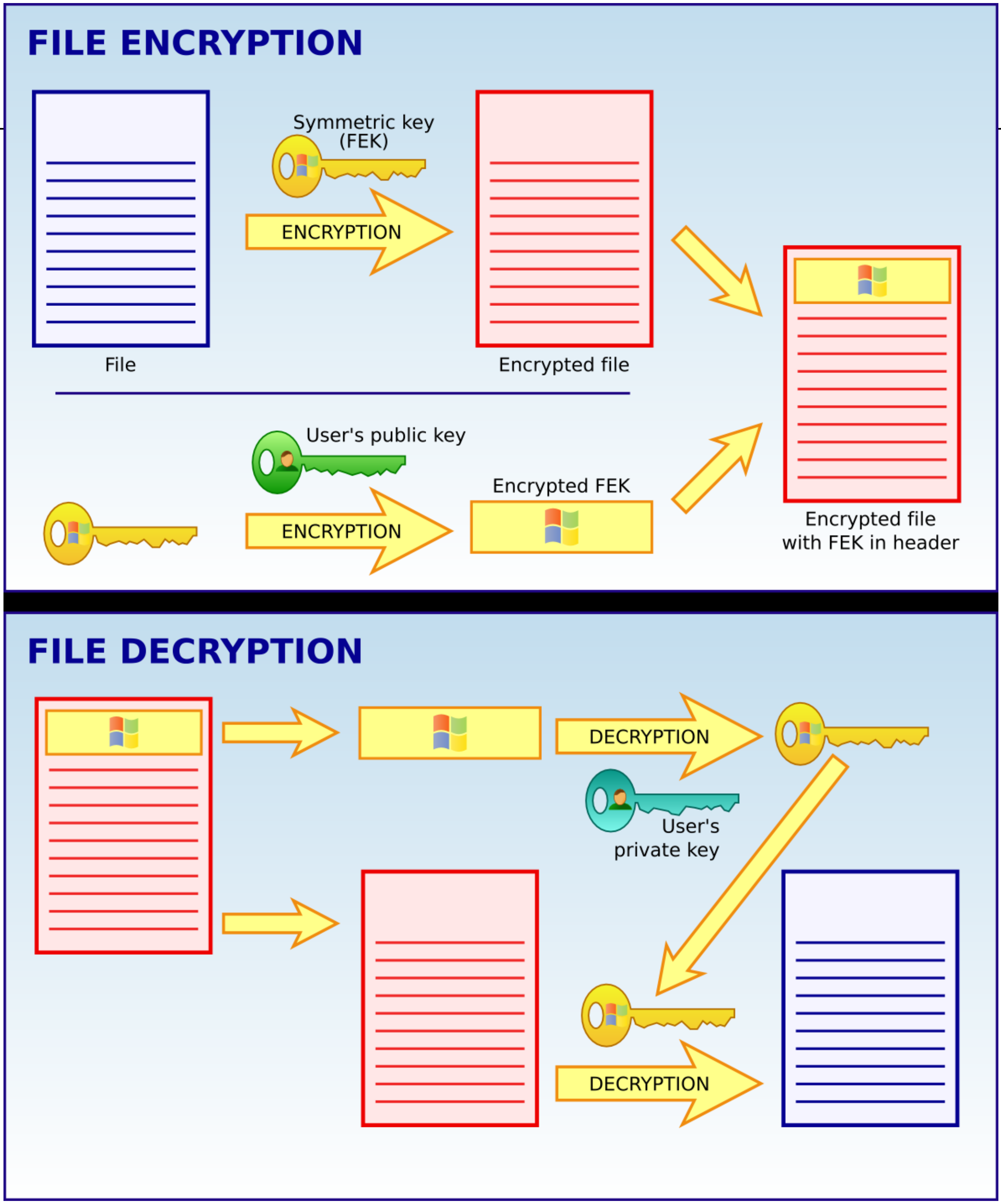
Future work

- Incremental development – versions 0.1, 0.2, 0.3 planned
  - Mount wide public key support
  - Filename and metadata (size and attributes) encryption
  - eCryptfs policy generators using generic utils
  - Convenient GUI for ease of use
  - Timeouts as supported by Ncryptfs
- Yet to address
  - Temporary files left unencrypted
  - Data on swap partition unencrypted (!!!)

EFS (Microsoft)

- Background of Invention
- Objects and Summary of invention
- General architecture
- Components of EFS
- EFS Driver
- File System Run Time Library (FSRTL)
- FSRTL callouts
- EFS service
- Win32 API
- Data Encryption/ Decryption/ Recovery
- General operations
- Miscellaneous details
- Security holes in EFS

Overview



Q: Forward secrecy?

Background of Invention

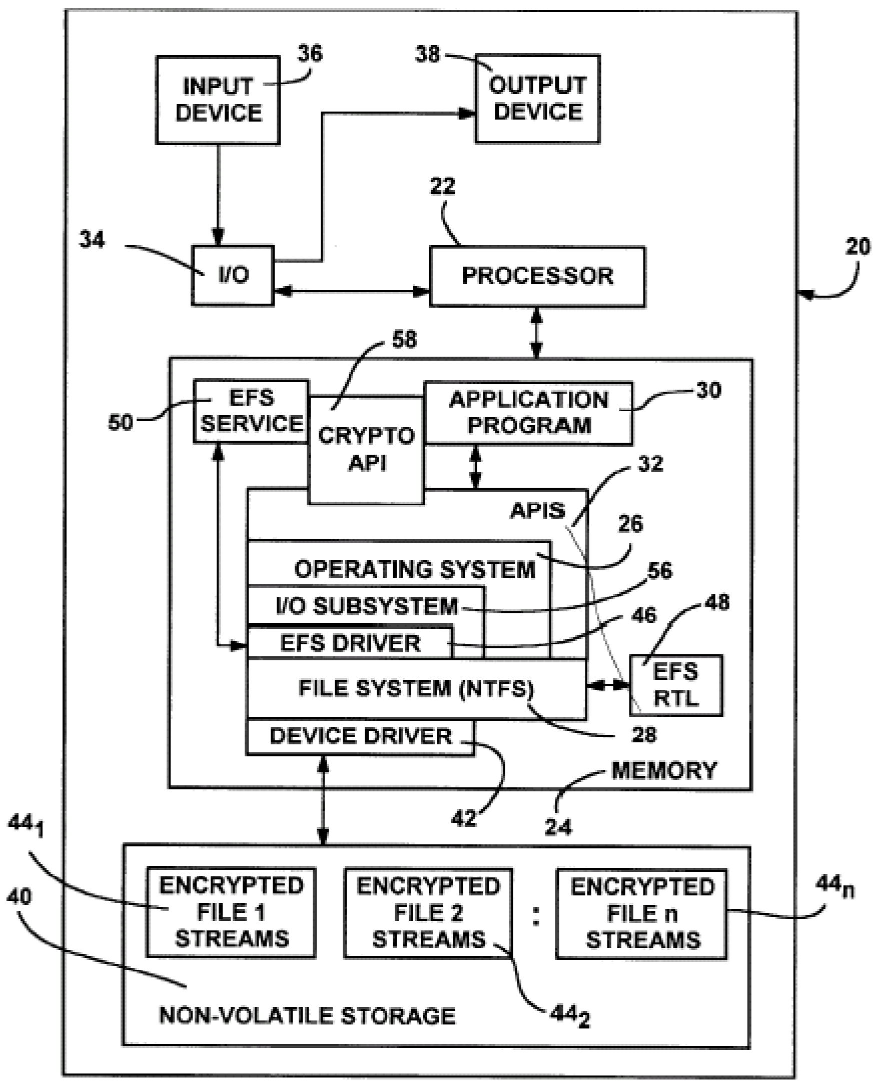
- Problem: Protecting sensitive data on disk
- Solution: Encrypt sensitive data

- Associated problems with naïve approach
  - Users choose weak passwords
  - Lost keys – share keys, compromise security
  - Key revocation
  - Overhead in encrypting each file
  - Intermediate temporary files
  - Application level encryption– key prone to attack
  - Not scalable to large number of users

Objects/ Summary of Invention

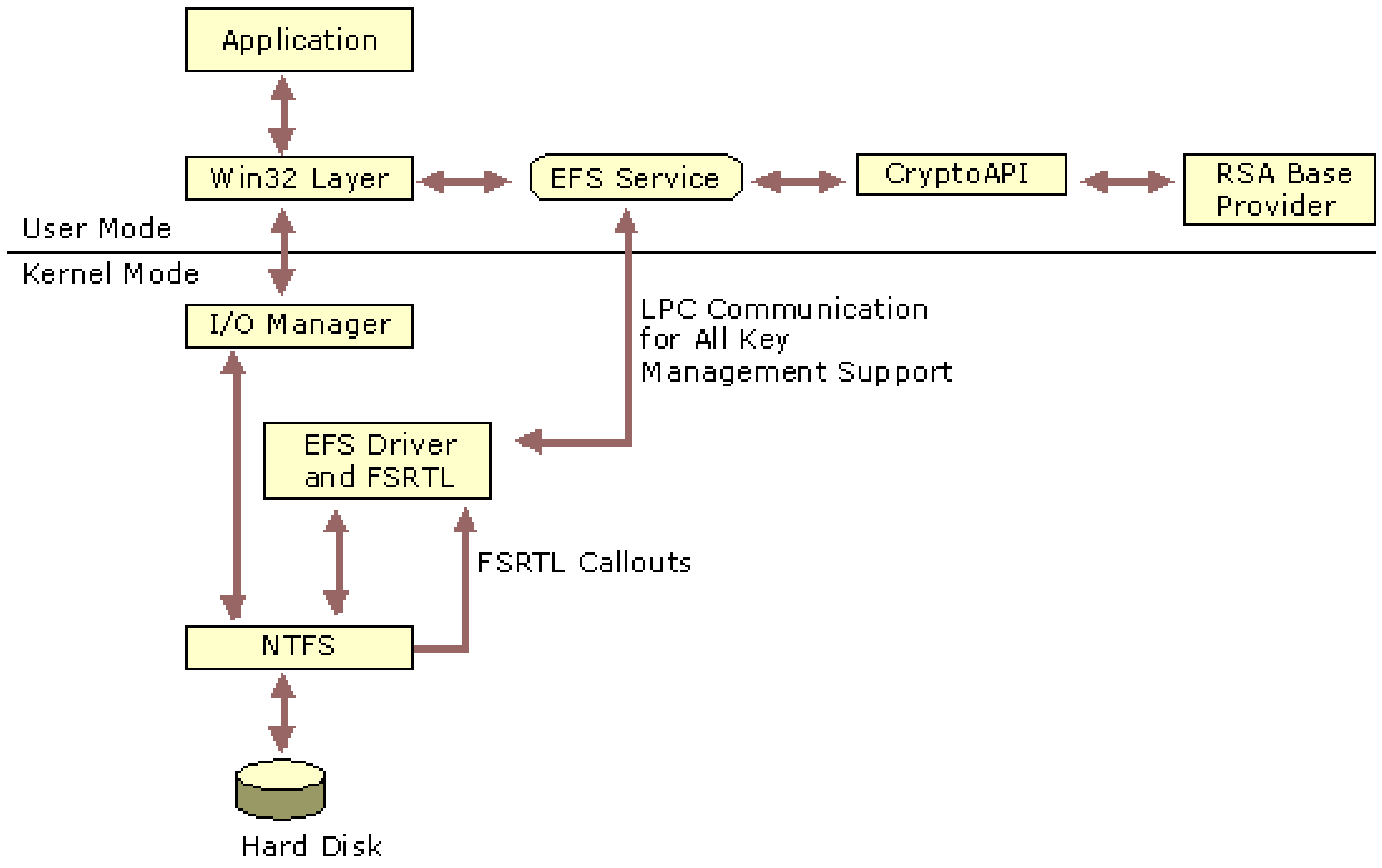
- Secure Storage- Integrate security into storage
- Security transparent to legitimate users
- Share data legitimately and securely
- Extensible – Adding new users/ ciphers
- Data recovery when user key lost
- Symmetric + Asymmetric – Performance
- Reference cipher: RSA + DES
- Quick idea
  - User chooses to encrypt – System generates a key (FEK) and prepares the context.
  - Data encrypted transparently using context
  - FEK encrypted with user public key in the file

General Architecture where EFS exists



- ☐Workstation/ Server/ Standalone system
- ☐Processor
- ☐Memory
- ☐Operating System (Win NT)
- ☐File System (NTFS)
- ☐Set of APIs
- ☐I/O devices
- ☐Non volatile storage device
- ☐Swap space – VM

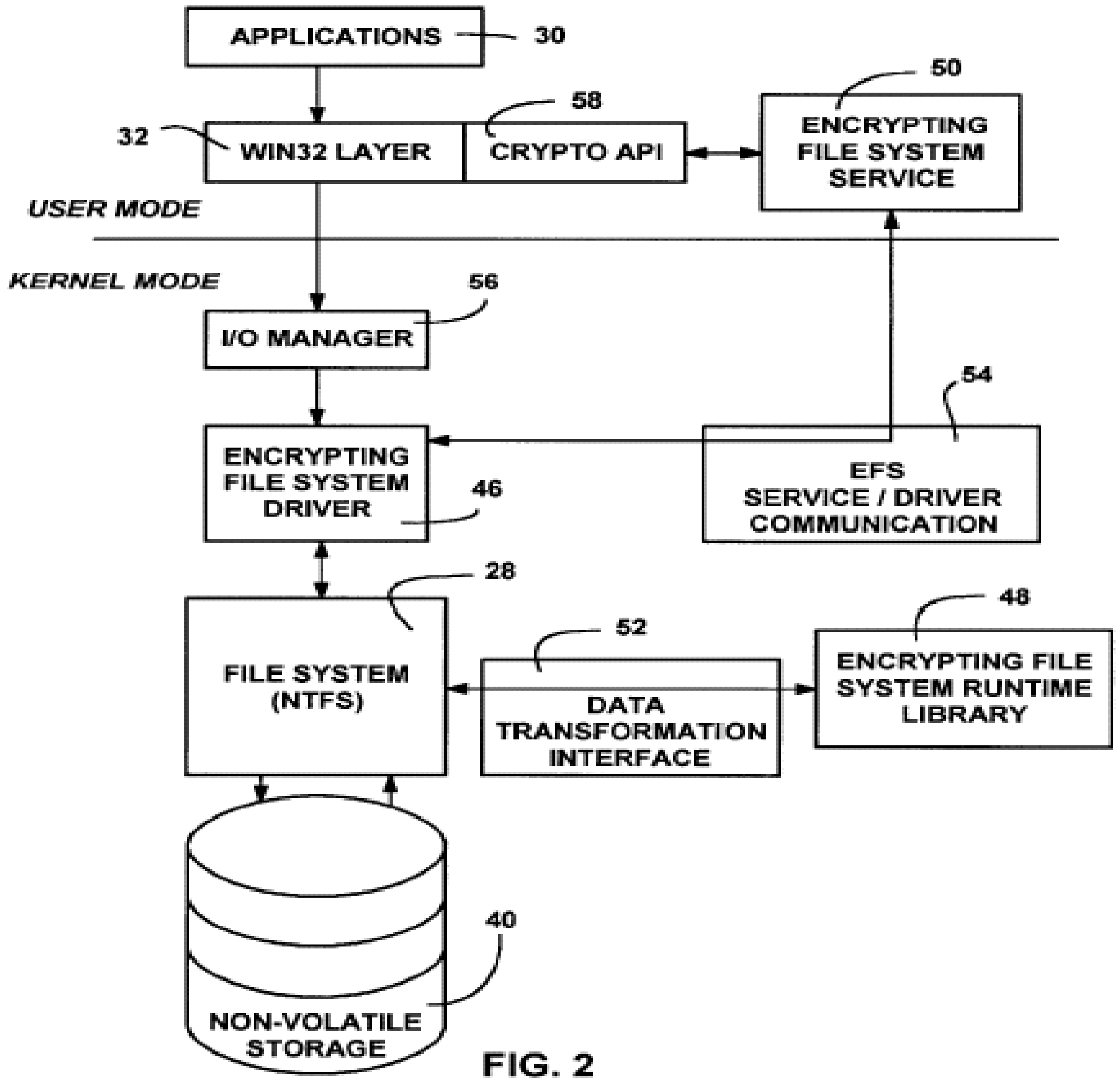
General Architecture where EFS exists



Encrypting File system and Method

- Background of Invention
- Objects and Summary of invention
- General architecture of EFS
- Components of EFS
- EFS Driver
- File System Run Time Library (FSRTL)
- FSRTL callouts
- EFS service
- Win32 API
- Data Encryption/ Decryption/ Recovery
- General operations
- Miscellaneous details
- Security holes in EFS

Components of EFS



APPLICATIONS
30
50
58
32
WIN32 LAYER
CRYPTO API
ENCRYPTING FILE SYSTEM SERVICE
USER MODE
KERNEL MODE
56
I/O MANAGER
54
ENCRYPTING FILE SYSTEM DRIVER
46
EFS SERVICE / DRIVER COMMUNICATION
28
48
52
FILE SYSTEM (NTFS)
DATA TRANSFORMATION INTERFACE
ENCRYPTING FILE SYSTEM RUNTIME LIBRARY
40
NON-VOLATILE STORAGE

FIG. 2

EFS Driver (EFSD)

- Sits above NTFS
- Instantiation of EFSD
- Registers FSRTL CB with NTFS
- EFSD <-> EFSS
  - Key mgmt services
  - Generate keys, Extract key from metadata, Get updated key
  - GenerateSessionKey for secure communication
  - Session Key used for EFSS<->EFSD<->FSRTL
- EFSD <-> FSRTL through NTFS
  - To perform FS operations read/write
  - Update with latest key

EFS FSRTL (FS Run Time Library)

- Implements callout functions for FS operations
- Generic Data Transformation interface
- FSRTL uses this for data encryption
- Gets FEK from EFSD
- Maintains cryptographic context
- EFSD and FSRTL – Part of same component
- EFSD <-> FSRTL through NTFS to maintain consistent FS state

Encrypting File system and Method

- ☐Background of Invention
- ☐Objects and Summary of invention
- ☐General architecture
- ☐Components of EFS
- ☐EFS Driver
- ☐File System Run Time Library (FSRTL)
- ☐FSRTL callouts
- ☐EFS service
- ☐Win32 API
- ☐Data Encryption/ Decryption/ Recovery
- ☐General operations
- ☐Miscellaneous details
- ☐Security holes in EFS

EFS FSRTL Callout Functions

- FileCreate for existing file
  - Called by NTFS if it determines FSRTL is interested in it.
  - Reads metadata from file and fills context
  - EFSD later reads context, gets key from EFSS
  - EFSD sets up key context with the key and stores in NTFS
- FileCreate for new file
  - Called by NTFS if the directory is set as encrypted.
  - Fills up context as requisition for new key
  - EFSD requests new key from EFSS
  - EFSD sets up key context with the key and stores in NTFS

EFS FSRTL Callout Functions (Contd..)

- Filecontrol_1
  - Called by NTFS when the state of the file changes
  - If encrypting – no other operations until complete
- Filecontrol_2
  - Communication between EFSD and FSRTL
  - Various requests with associated data for context preparation
  - EFS_SET_ATTR – write new metadata to FSRTL
  - EFS_GET_ATTR – get stored metadata from FSRTL
  - EFS_DECRYPT_BEGIN – FSRTL locks file until decrypt ends
  - EFS_DEL_ATTR – Decryption done, delete metadata
  - EFS_ENCRYPT_DONE – Encryption done, allow other operations

EFS FSRTL Callout Functions (Contd..)

- AfterReadProcess
  - FS calls this if stream needs to be decrypted
  - FSRTL decrypts the stream, FS returns to user
- BeforeWriteProcess
  - FS calls this if stream needs to be encrypted
  - FSRTL encrypts the stream, FS stores on disk
- CleanUp
  - FS calls this before freeing resources for stream
  - FSRTL frees up its context and resources allocated

EFS FSRTL Callout Functions (Contd..)

- AttachVolume
  - FS calls this on first user [en/de]cryption on the volume
  - FSRTL requests attachment to the device
  - All calls routed to EFS Driver before NTFS
- DismountVolume
  - FS calls this if when drive ejected or power off
  - Free allocated resources during AttachVolume

EFS Service

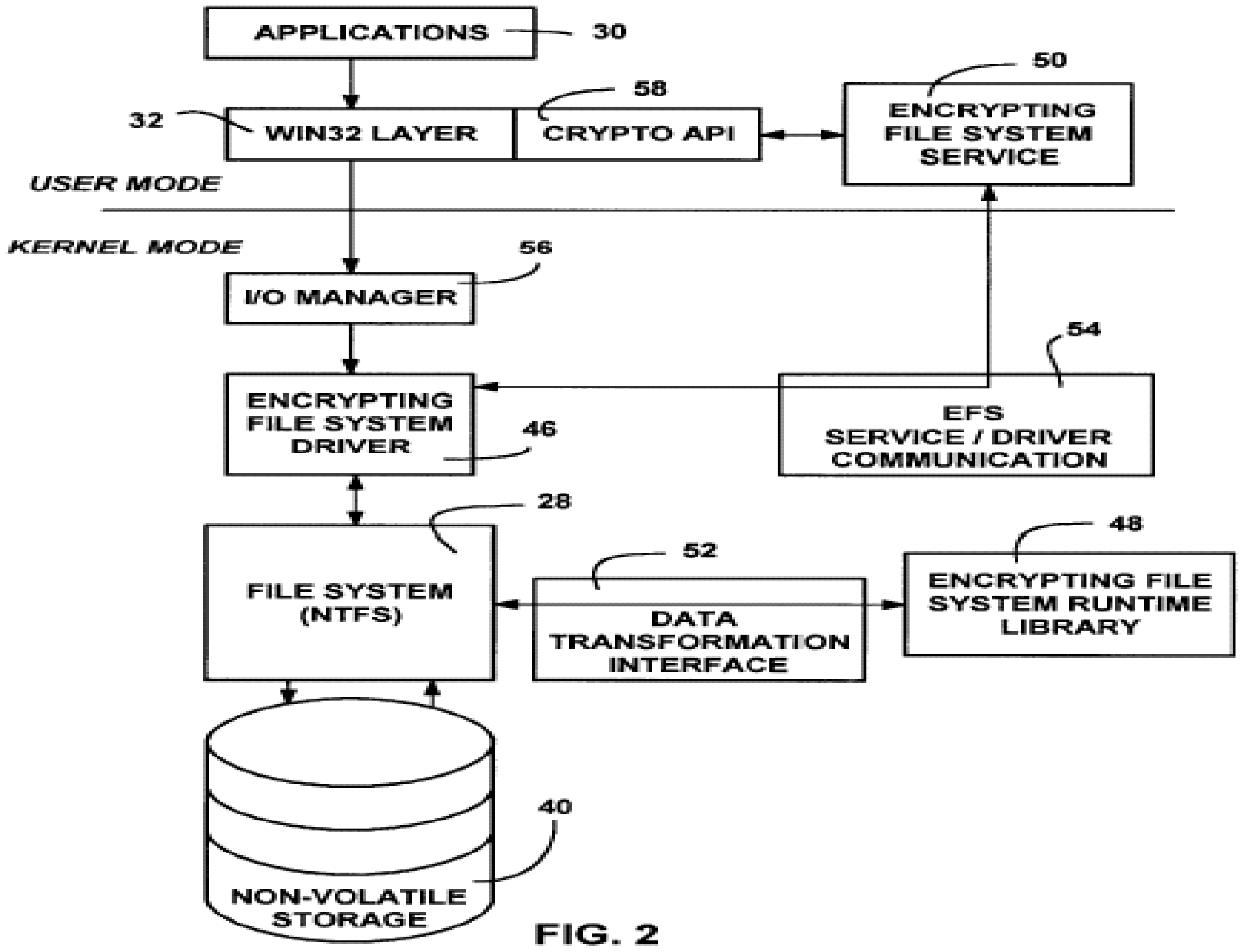

APPLICATIONS
30
50
58
32
WIN32 LAYER
CRYPTO API
ENCRYPTING FILE SYSTEM SERVICE
USER MODE
KERNEL MODE
56
I/O MANAGER
54
ENCRYPTING FILE SYSTEM DRIVER
46
EFS SERVICE / DRIVER COMMUNICATION
28
48
52
FILE SYSTEM (NTFS)
DATA TRANSFORMATION INTERFACE
ENCRYPTING FILE SYSTEM RUNTIME LIBRARY
40
NON-VOLATILE STORAGE

FIG. 2

EFS Service

- Part of Win NT security service
- Secure communication with kernel through LSA
- Talks to CryptoAPI in user space
- Services provided
  - Generate Session Key
  - Generate File Encryption Key (FEK)
  - Extract FEK from metadata using user’s private keys
  - Win32 API support
- EFSD and EFSS synchronize with one other on startup and exchange session key

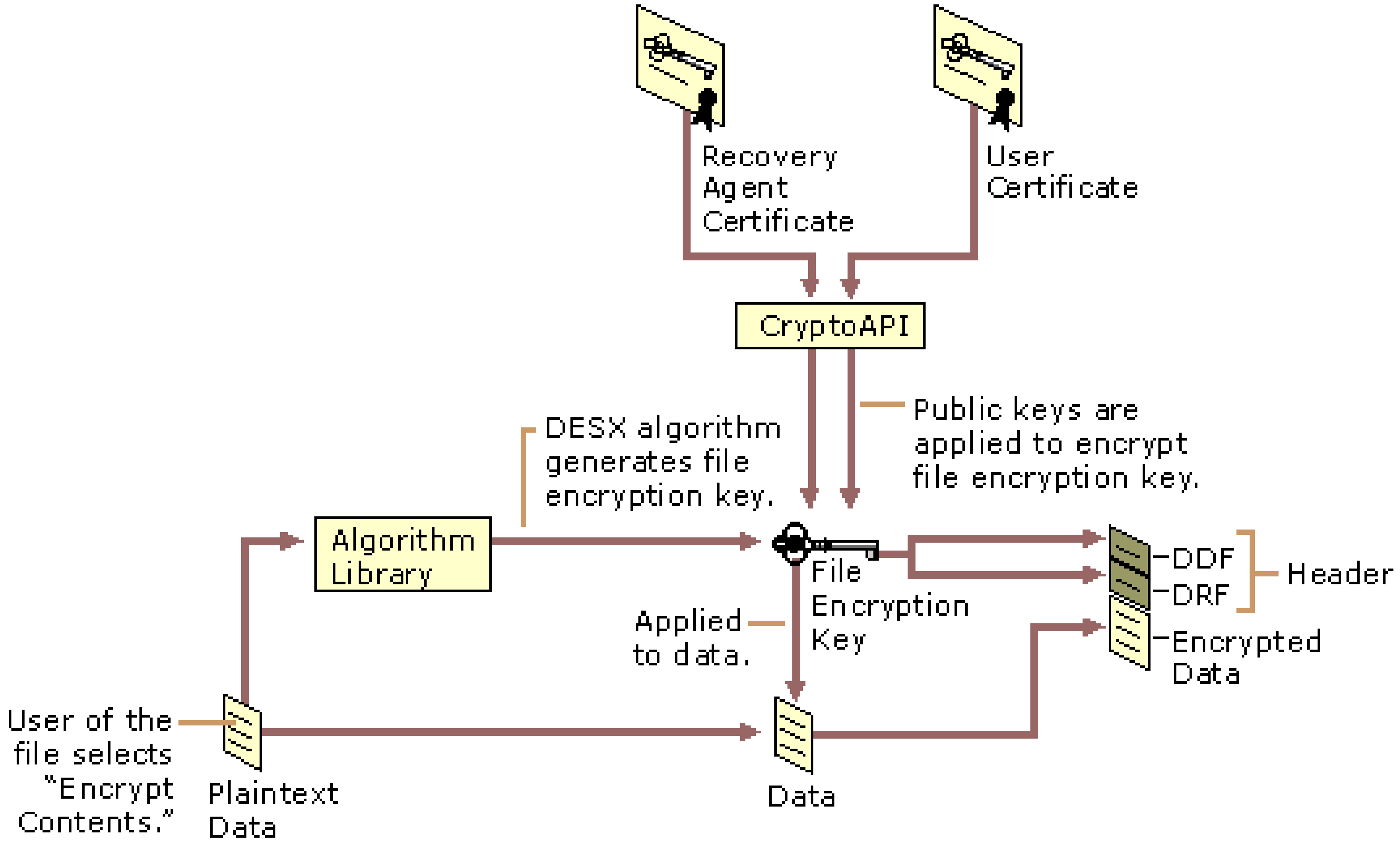
Encrypting File system and Method

- ☐Background of Invention
- ☐Objects and Summary of invention
- ☐General architecture
- ☐Components of EFS
- ☐EFS Driver
- ☐File System Run Time Library (FSRTL)
- ☐FSRTL callouts
- ☐EFS service
- ☐Win32 API
- ☐Data Encryption/ Decryption/ Recovery
- ☐General operations
- ☐Miscellaneous details
- ☐Security holes in EFS

Win32 API

- User mode services by EFSS to use encryption
- Interfaces provided for operations on plain text files
    - EncryptFile
    - DecryptFile
- Interfaces provided for backup encrypted files
    - OpenRawFile
    - ReadRawFile
    - WriteRawFile
    - CloseRawFile
- During raw file transfer, EFSS informs FSRTL through FileControl_2 not to encrypt/decrypt data

Overview


Recovery Agent Certificate
User Certificate
CryptoAPI
Public keys are applied to encrypt file encryption key.
DESX algorithm generates file encryption key.
Algorithm Library
File Encryption Key
DDF
DRF
Header
Encrypted Data
Applied to data.
User of the file selects "Encrypt Contents."
Plaintext Data
Data

Data Encryption

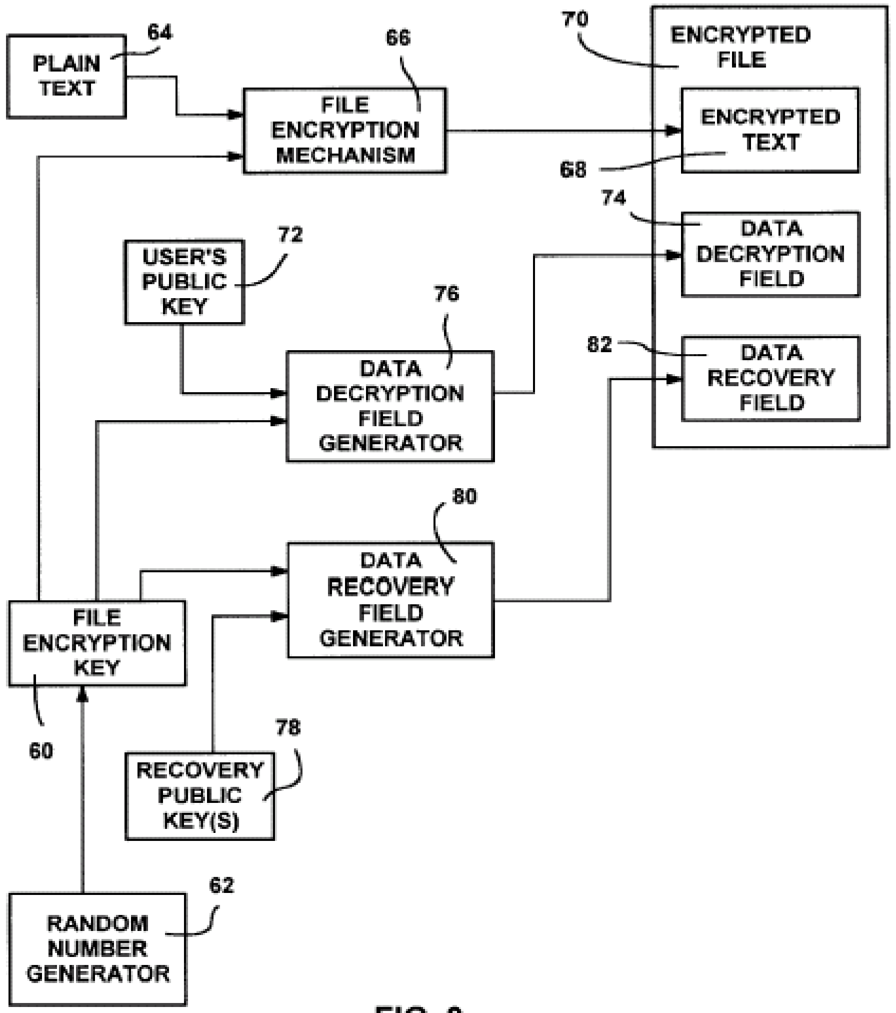
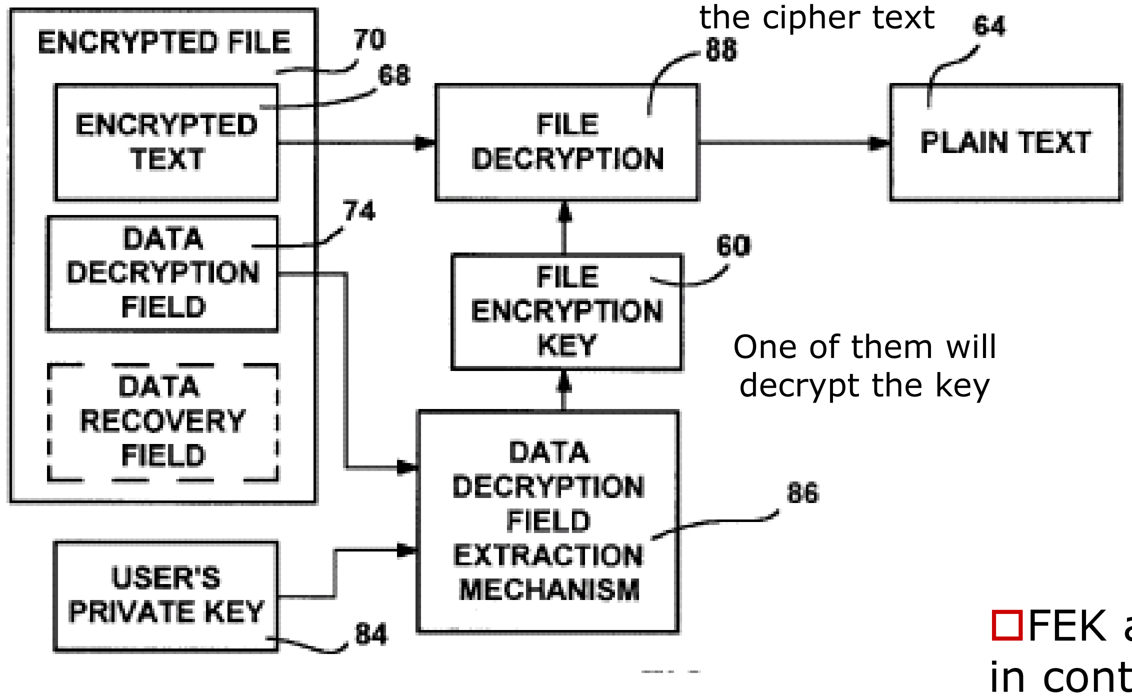


FIG. 3

- Encryption Key – Rand num
- Ref symmetric cipher DES
- Data Decryption Field - DDF
- Data Recovery Field - DRF
- Private keys on smart card – not used during encryption
- Ref asymmetric cipher RSA
- Not tied to any cipher or key length

Data Decryption

FEK used to decrypt the cipher text



One of them will decrypt the key

User private key is used to decrypt each DDF

- FEK and Decryption stored in context info
- Ease of random access

Data Recovery

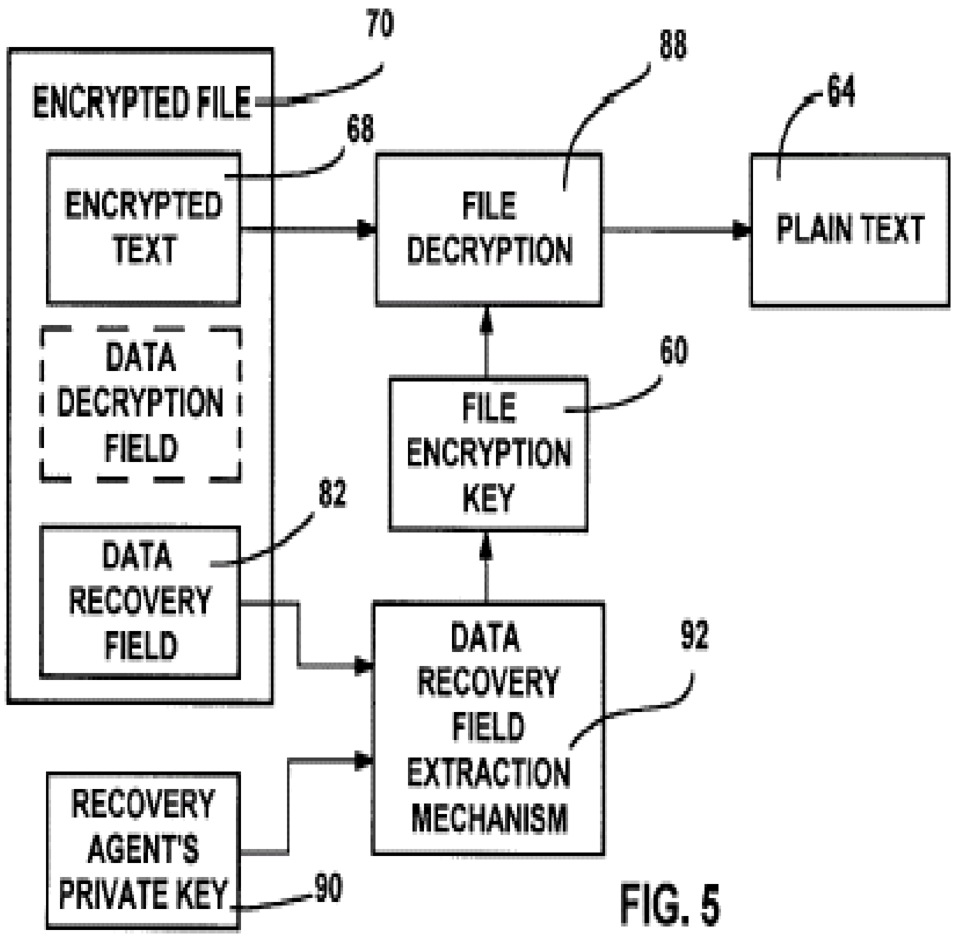


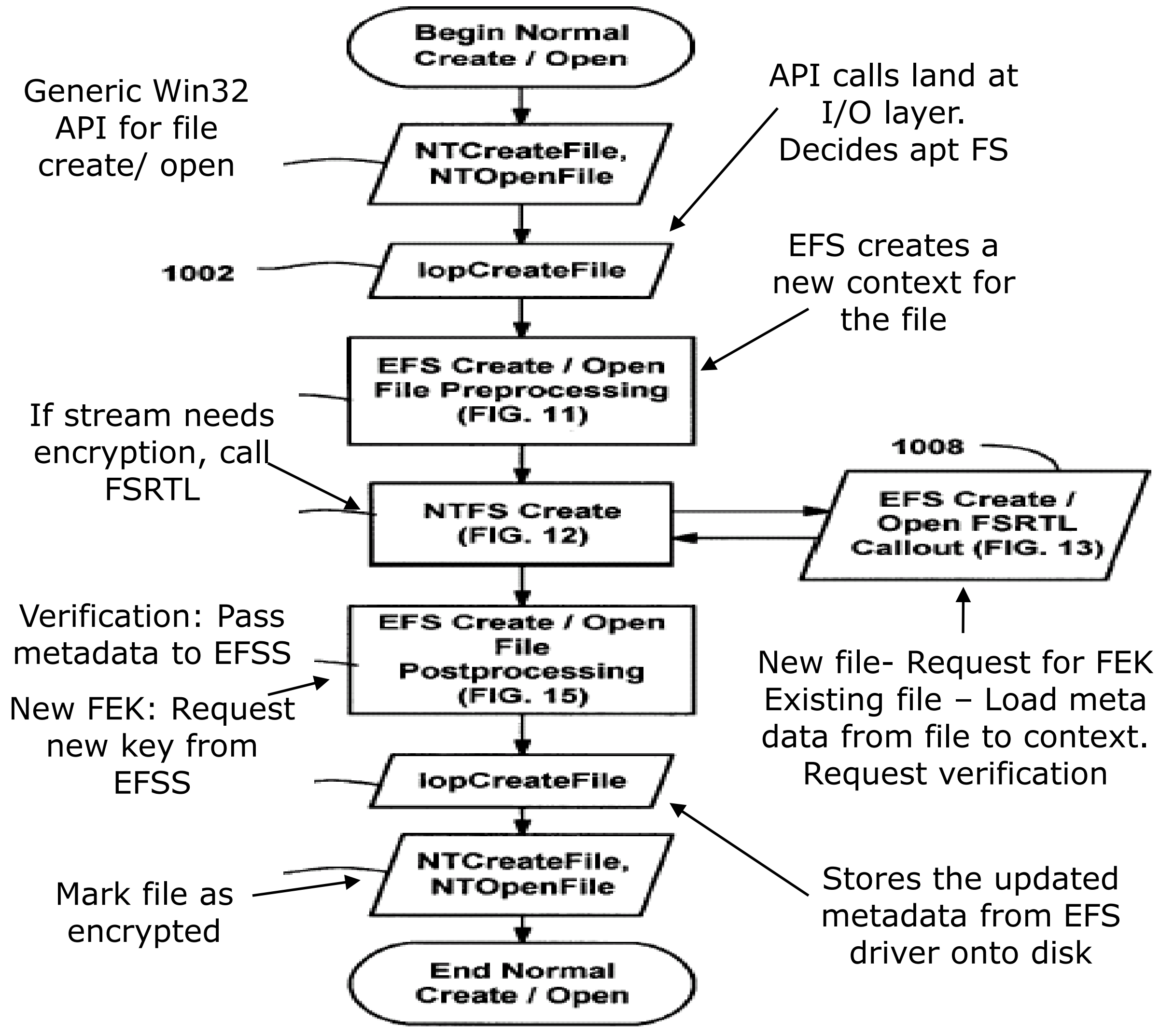
FIG. 5

- When users leave/ lose keys
- Search starts from DDF and goes on to DRF
- Reveals only FEK not user private key
- Domain policy decides the recovery agents
- Policy contains public keys
- Agent specifies private key
- Policy MD5 hashed to ensure authenticity
- Hash value authenticated before using the policy

Encrypting File system and Method

- Background of Invention
- Objects and Summary of invention
- General architecture
- Components of EFS
- EFS Driver
- File System Run Time Library (FSRTL)
- FSRTL callouts
- EFS service
- Win32 API
- Data Encryption/ Decryption/ Recovery
- General operations
- Miscellaneous details
- Security holes in EFS

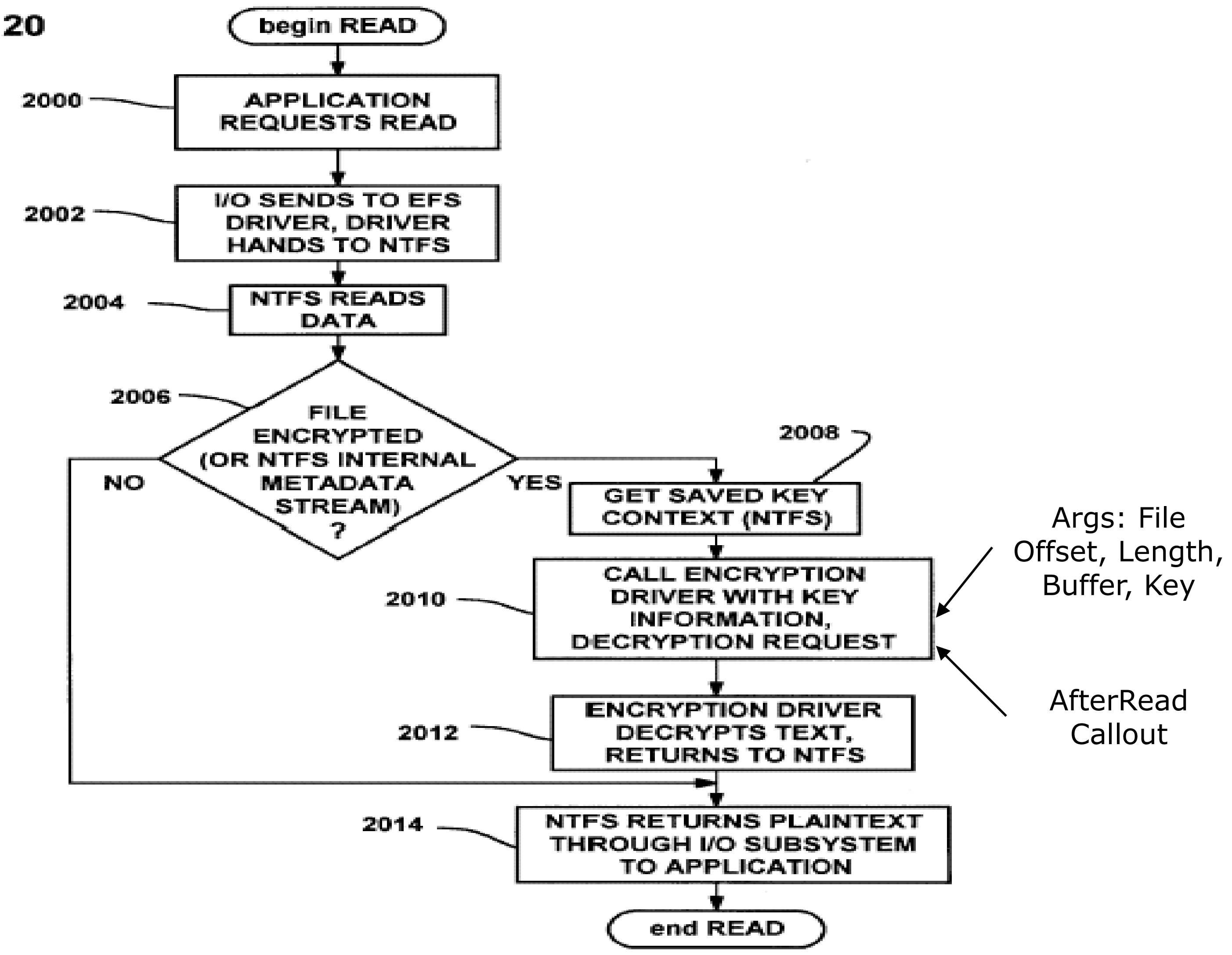
General operation – Create/ Open



Begin Normal Create / Open
Generic Win32 API for file create/ open
NTCreateFile, NTOpenFile
API calls land at I/O layer. Decides apt FS
1002
IopCreateFile
EFS creates a new context for the file
EFS Create / Open File Preprocessing (FIG. 11)
If stream needs encryption, call FSRTL
1008
NTFS Create (FIG. 12)
EFS Create / Open FSRTL Callout (FIG. 13)
Verification: Pass metadata to EFSS
New FEK: Request new key from EFSS
EFS Create / Open File Postprocessing (FIG. 15)
New file- Request for FEK
Existing file – Load meta data from file to context.
Request verification
IopCreateFile
Stores the updated metadata from EFS driver onto disk
Mark file as encrypted
NTCreateFile, NTOpenFile
End Normal Create / Open

General operation – Read

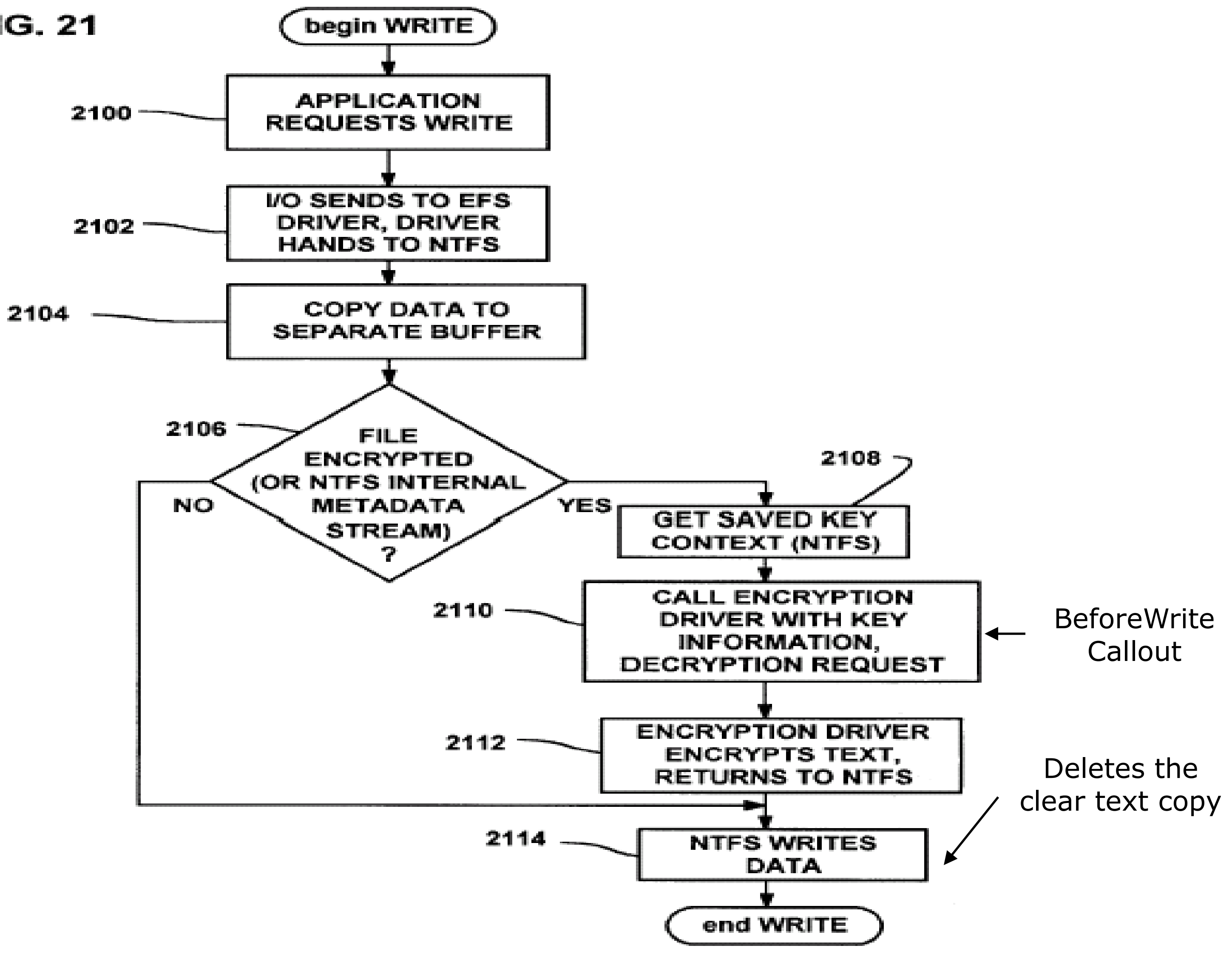
FIG. 20



begin READ
2000
APPLICATION REQUESTS READ
2002
I/O SENDS TO EFS DRIVER, DRIVER HANDS TO NTFS
2004
NTFS READS DATA
2006
FILE ENCRYPTED (OR NTFS INTERNAL METADATA STREAM) ?
NO
YES
2008
GET SAVED KEY CONTEXT (NTFS)
2010
CALL ENCRYPTION DRIVER WITH KEY INFORMATION, DECRYPTION REQUEST
Args: File Offset, Length, Buffer, Key
AfterRead Callout
2012
ENCRYPTION DRIVER DECRYPTS TEXT, RETURNS TO NTFS
2014
NTFS RETURNS PLAINTEXT THROUGH I/O SUBSYSTEM TO APPLICATION
end READ

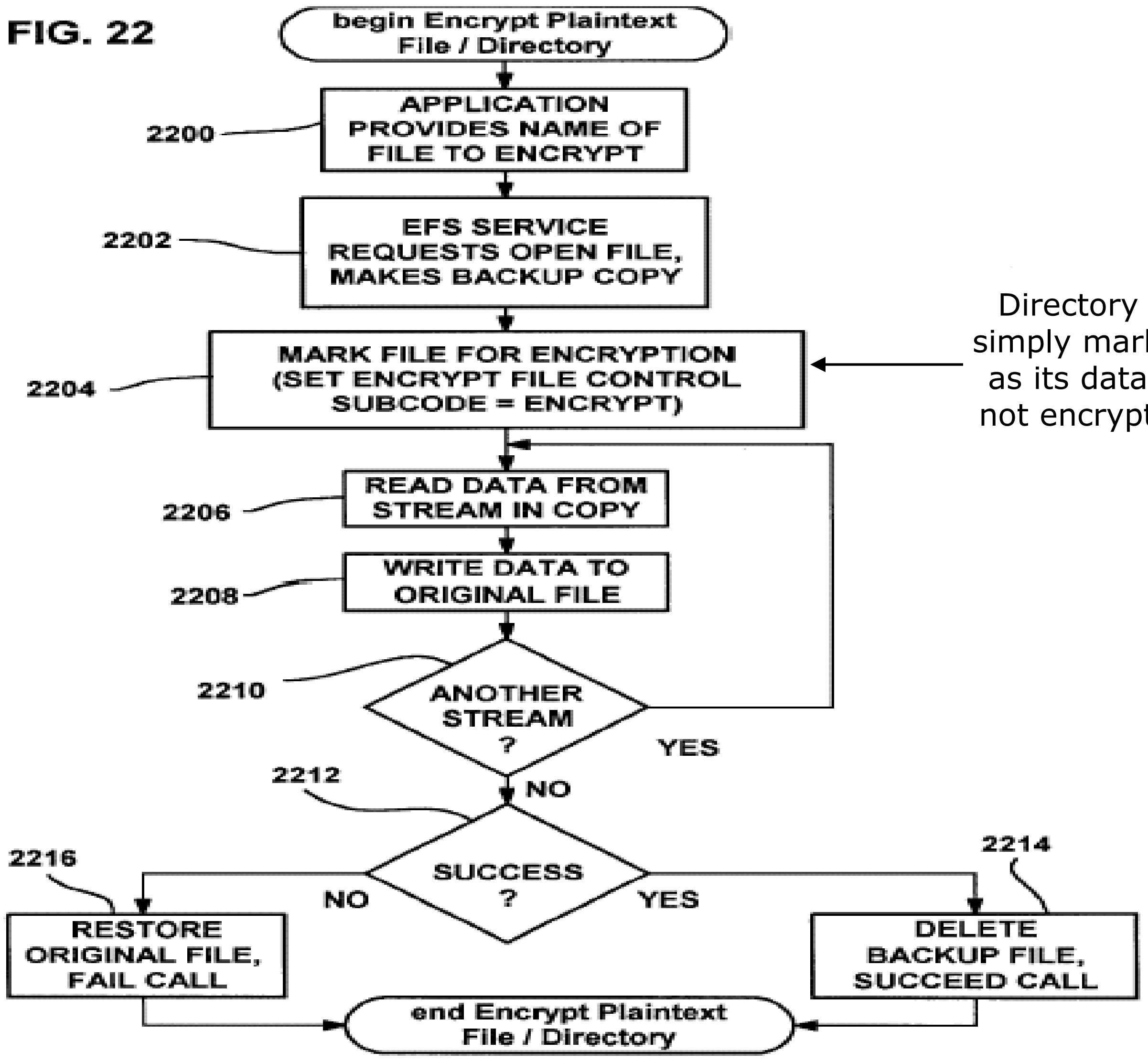
General operation – Write

FIG. 21


begin WRITE
2100
APPLICATION REQUESTS WRITE
2102
I/O SENDS TO EFS DRIVER, DRIVER HANDS TO NTFS
2104
COPY DATA TO SEPARATE BUFFER
2106
FILE ENCRYPTED (OR NTFS INTERNAL METADATA STREAM) ?
NO
YES
2108
GET SAVED KEY CONTEXT (NTFS)
2110
CALL ENCRYPTION DRIVER WITH KEY INFORMATION, DECRYPTION REQUEST
BeforeWrite Callout
2112
ENCRYPTION DRIVER ENCRYPTS TEXT, RETURNS TO NTFS
Deletes the clear text copy
2114
NTFS WRITES DATA
end WRITE

General operation – Win32 EncryptFile

FIG. 22

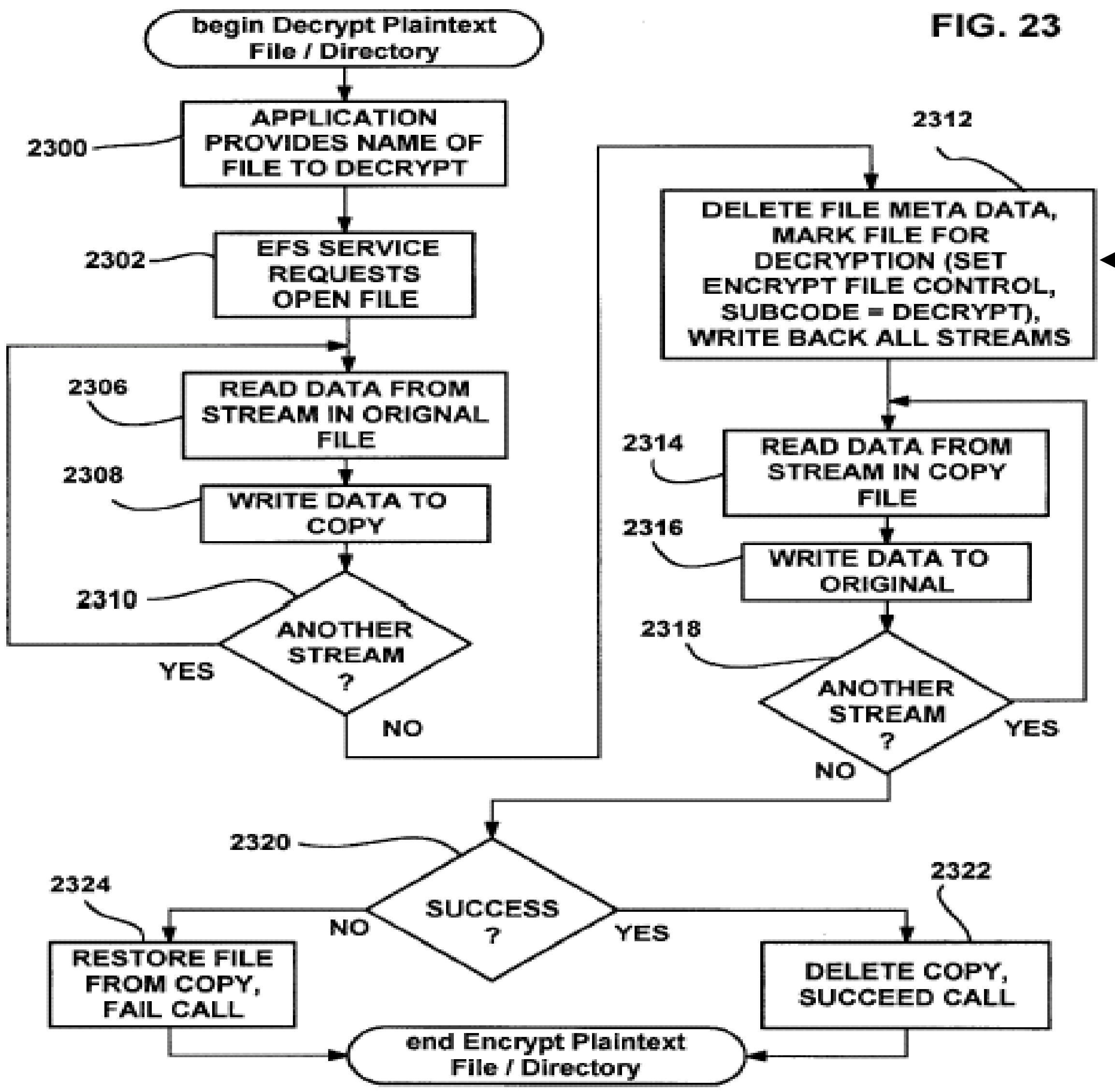


begin Encrypt Plaintext File / Directory
2200
APPLICATION PROVIDES NAME OF FILE TO ENCRYPT
2202
EFS SERVICE REQUESTS OPEN FILE, MAKES BACKUP COPY
2204
MARK FILE FOR ENCRYPTION (SET ENCRYPT FILE CONTROL SUBCODE = ENCRYPT)
2206
READ DATA FROM STREAM IN COPY
2208
WRITE DATA TO ORIGINAL FILE
2210
ANOTHER STREAM ?
YES
NO
2212
SUCCESS ?
NO
YES
2216
RESTORE ORIGINAL FILE, FAIL CALL
2214
DELETE BACKUP FILE, SUCCEED CALL
end Encrypt Plaintext File / Directory

Directory is simply marked as its data is not encrypted

General operation – Win32 DecryptFile

Make a copy of the original file



begin Decrypt Plaintext File / Directory
2300
APPLICATION PROVIDES NAME OF FILE TO DECRYPT
2302
EFS SERVICE REQUESTS OPEN FILE
2306
READ DATA FROM STREAM IN ORIGNAL FILE
2308
WRITE DATA TO COPY
2310
ANOTHER STREAM ?
YES
NO
2312
DELETE FILE META DATA, MARK FILE FOR DECRYPTION (SET ENCRYPT FILE CONTROL, SUBCODE = DECRYPT), WRITE BACK ALL STREAMS
2314
READ DATA FROM STREAM IN COPY FILE
2316
WRITE DATA TO ORIGINAL
2318
ANOTHER STREAM ?
YES
NO
2320
SUCCESS ?
NO
YES
2324
RESTORE FILE FROM COPY, FAIL CALL
2322
DELETE COPY, SUCCEED CALL
end Encrypt Plaintext File / Directory

FIG. 23

Directory is simply marked as its data is not encrypted

Overwrite the original with plaintext

Miscellaneous details

- Intermediate/ Temporary files encrypted too
- EFSD uses non paged pool of memory
  - FEK and other context details not swapped to disk
- Data sharing
  - FEK encrypted with public keys of all legitimate users
- Easy to use - no administrative effort involved
- Support for encryption on remote server
  - Server support for EFS, Data on wire in plaintext
- File copy across FS
  - Copy across EFS aware FS – encrypted content
  - Copy to EFS unaware FS (FAT32) – plaintext data copied

Security holes in EFS (Win 2K)

- Administrator – Default Recovery agent
  - Has access to all user data
  - Win XP has no default recovery agent – Policy decides agents
- User Private key protection
  - Protected by user password only – Not encrypted
  - Weak Hashes of pass-phrases are kept !!!
  - Key lies in all kinds of other places that are accessible at various times to different principals (e.g., pass reset etc.)
- No secure deletion in place
  - After encrypting files, plaintext version only deleted
  - Win XP does not yet solve this problem
  - Use third part tools for secure deletion
- Directory contents not encrypted