TransCrypt: A Secure and Transparent Encrypting File System for Enterprises

Satyam Sharma, Rajat Moona and Dheeraj Sanghi
Department of Computer Science and Engineering
Indian Institute of Technology Kanpur
{ssatyam, moona, dheeraj}@cse.iitk.ac.in

Abstract— **Increasing thefts of sensitive data owned by individuals and organizations call for an integrated solution to the problem of storage security. Most existing systems are designed for personal use and do not address the unique demands of enterprise environments. An *enterprise-class* encrypting file system must take a cohesive approach towards solving the issues associated with data security in organizations, such as flexibility for multi-user scenarios, transparent remote access of shared file systems and defense against an array of threats including insider attacks while trusting the fewest number of entities. In this paper, we formalize a general threat model for storage security and discuss how existing systems tackle a narrow threat model and are thus susceptible to attacks. We present the conceptualization, design and implementation of *TransCrypt*, a kernel-space encrypting file system that incorporates an advanced key management scheme to provide a high grade of security while remaining transparent and easily usable. It examines difficult problems that are not considered by any existing system such as avoiding trusting the superuser account or privileged user-space processes and proposes novel solutions for them. These enhancements enable TransCrypt to protect against a wider threat model and address several lacunae in existing systems.**

I. Introduction

The need for data security emerges from the widespread deployment of shared file systems, greater mobility of computers and the rapid miniaturization of storage devices. It is increasingly obvious that the value of data is much more than the value of the underlying devices. The theft of a laptop or a USB thumbdrive leaves the victim vulnerable to the risk of identify theft in addition to the loss of personal and financial data. Several recent incidents accentuate the need for a cohesive solution to the problem of storage security that protects data using strong cryptographic methods in both personal and organizational scenarios.

An *encrypting file system* employs secure and efficient mechanisms to encrypt or decrypt data on-the-fly as it is being written to or read from the underlying disk, to provide a level of data privacy that goes beyond simple access control. Issues such as trust models, backups and data recovery must also be resolved. Other challenges faced when designing a storage security framework include immunity from attacks launched by privileged entities, enabling legitimate remote access to shared encrypted volumes and providing a scalable and transparent key management scheme suitable for enterprise deployment.

In this paper, we describe the conceptualization, design and implementation of *TransCrypt*, an encrypting file system for Linux that takes a holistic approach towards solving the aforementioned issues. TransCrypt makes a crucial distinction between the kernel and user-space from a security perspective. It incorporates an advanced key management scheme that excludes the administrator account and user-space processes with superuser privileges from the trust model. This enables TransCrypt to assume a wider threat model than that tackled by existing systems making it immune to several attacks that may be launched from the inside. It also provides fine-grained access control and supports the use of hardware authentication tokens. Mechanisms for data recovery and secure remote sharing are also included to make TransCrypt truly enterprise-ready. Encouragingly, initial performance evaluations indicate that these security and usability benefits can be availed with small and tolerable run-time overheads.

The next section discusses a representative cross-section of the popular encrypting file systems in use today. Section III formalizes a generic attack model on file system security and describes the salient features of our solution. In Section IV, the cryptographic design and key management scheme are presented in an operating system agnostic fashion. The architecture, configuration parameters, software components, implementation and working of TransCrypt are explained in Section V. Section VI provides an informal performance analysis and compares the features of TransCrypt against related work, clearly bringing out the differentiating aspects. Finally, concluding remarks are presented in Section VII.

II. Related Work

The *Cryptographic File System* [1] was the first encrypting file system for Unix. It is implemented as a user-space NFS server to introduce a cryptographic layer between the virtual file system and the disk, employing a double-mount technique. This design leads to data copy and context switch overheads. Also, it uses a common passphrase-derived mount-wide key and a basic key management scheme that provides coarse granularity of access control. Several other solutions such as *EncFS* [2] use a similar design approach.

Microsoft Windows includes a native *Encrypting File System* [3] tightly integrated with NTFS. It uses per-file encryption keys and per-user public and private keypairs, thus enabling fine-grained access control. Although bulk encryption is done in kernel-space, key management tasks are delegated to a user-space service. This hybrid design makes it vulnerable to user-space attacks and the superuser account must thus be a trusted entity. Also, it makes no provision for file integrity. Apple *FileVault* [4], the native file encryption mechanism in Mac OS, creates and maintains encrypted sparse images to store home directory contents and provides virtual memory encryption. However, it fails to provide any file sharing or integrity and cannot be used outside the users' home directory.

The most popular encrypting file system for Linux that is part of the standard kernel is *dm-crypt* [5]. It uses the kernel's device mapper infrastructure to implement a cryptographic layer using functions provided by the native kernel *CryptoAPI* [6]. However, it lacks flexibility due to the use of a common mount-wide key and tackles a narrow threat model.

eCryptfs [7] is the first attempt at designing an *enterprise-class* cryptographic file system for Linux. It uses per-file keys and user-specific keys to enable fine-grained sharing. A PGP-inspired file header format stores the cryptographic metadata associated with each file. The kernel implementation uses stackable vnodes to introduce a layer of encryption that can fit over any underlying file system, similar to Cryptfs [8]. eCryptfs is the only existing work with specifications similar to those of TransCrypt, though vastly different in design. Its hybrid design employing a user-space key management daemon exposes the system to user-space attacks and fails to exclude the superuser account from the trust model. Also, including metadata in file contents as a header reduces transparency for end-use and requires separate tools for managing file sharing.

TransCrypt trusts fewer entities, protects against a wider threat model, provides fine-grained file sharing and integrity protection, and utilizes a kernel-space approach for greater security and better performance. The additional demands of enterprise environments such as secure remote sharing, data recovery and immunity from attacks launched from the inside are also met.

III. The TransCrypt Approach

We now present a generic threat model for file systems and a formal categorizaton of various attacks. A consistent understanding of the kind of attacks that must be prevented and those that are beyond the threat model has been the deciding factor in the design of our key management scheme.

A. Threat Model

Threat models for network security assume that any message sent over the communication channel can be intercepted, read, deleted or modified by an attacker at will. Similarly, TransCrypt's storage security threat model assumes the attacker to be capable of reading, deleting or modifying any data on the disk at any point of time. This assumption significantly widens our threat model to include attacks that are not addressed by any related work, such as those discussed below. Essentially, an attack on stored data is an unauthorized attempt to access it. Attacks may thus be classified into *offline* and *online* attacks based on the mode of access.

1) Offline attacks: In the most common offline attack, an attacker gets physical access to the storage device and connects it to another system to circumvent the access control on the victim's file system and read the plaintext data. Encrypting file systems generally deal with this attack by encrypting the disk contents with a key that may be randomly generated or derived from a passphrase. Systems that provide for user-specific keys encrypt copies of the file encryption key itself with the keys of various users who can access that file.

We also believe that file keys must only be known to the kernel of the operating system. Such a view is necessary for sharing files securely among multiple users. This prevents an attack in which a user whose access to a file has been revoked continues to use the old key to read it. Changing the file key and re-encrypting the entire file with the new key at every revocation may not be acceptable from a performance point of view. Systems that provide for user-specific keys encrypt copies of the file key itself with the keys of various users who can access that file. These encrypted per-user tokens are then stored along with the file. However, such systems are easily broken in our wide threat model that assumes any data or metadata can be read and modified on the disk at any instant. For example, a malicious user can read his token, decrypt it using his private key and store the file key for future use. He may even share it with other unauthorized users, thus granting them access to the file illegally. This attack is easy to execute but not addressed by any existing system. We call it the *key leak attack* and propose a solution to it.

2) Online attacks: Online attacks exploit the fact that sensitive data or encryption keys may be present in the system memory in plaintext when an encrypted volume is mounted. Although superuser privileges may be required for launching such attacks, we choose not to trust the administrator account.

- A general *key scavenging attack* attempts to intelligently run through the system memory (the kernel's address space in case of kernel-based encrypting file systems) trying to locate and read encryption keys.
- Any system that requires the communication of key material between kernel-space and user-space components is vulnerable to a general *user-space attack*, wherein the attacker tries to capture the key when it is present in the address space of an exploitable user-space process.
- *Daemon masquerading* is another online attack on schemes that employ a trusted user-space process for key management. An attacker can replace the trusted daemon with a malicious program or launch a man-in-the-middle attack on the kernel-space user-space channel to gain access to encryption keys.
- In the *page cache attack*, an attacker may insert and load a malicious kernel module from the user-space (using the *insmod* command, for example) at run-time to traverse the page cache and access file contents.

Attacks that require superuser privileges may be classified as *hard* or *easy* depending on the level of sophistication and effort required to execute them. Although accessing or modifying specific kernel-space data structures of a running system is difficult even for the administrator, exploiting user-space vulnerabilities is clearly easy. In a typical compromise, an attacker may gain superuser privileges for a short duration and trivially undertake the above attacks. A distinction between the kernel and user-space from a security perspective is becoming crucial as temporary root attacks become common. Such a distinction is sound, given that it is easier to trust the kernel, a smaller entity that can be conveniently secured, than user-space processes that can be compromised in several and simpler ways. Newer Linux kernels, moreover, include several advancements such as disabling potential points of attack (like */dev/kmem* and */proc/core*) and enforcing verification of digitally signed kernel modules before loading them [9] through compile-time options, thus making it possible to efficiently secure the few entry points into a running kernel even from a privileged attacker. The design decisions of TransCrypt have been taken to prevent practically viable attacks that are not addressed by existing systems, while ignoring attacks that require substantially greater sophistication. No existing system attempts to exclude the superuser account from the trust model and tackle the above attacks.

B. TransCrypt Specifications

TransCrypt's basic feature is the *privacy* and *integrity* of a user's data and the ability to *share* file system objects among multiple users without having to share passphrases or use common mount-wide keys. Avoiding the use of a common key also leads to less ciphertext encrypted with the same key, making cryptanalysis more difficult. Thus, the *key management scheme* proposes per-file encryption keys and per-user public and private keypairs. Also, TransCrypt uses only randomly generated keys instead of passphrase-derived ones, thus preventing dictionary-based attacks.

1) File integrity: It must be ensured that a file or its metadata can only be modified by an entity that knows its file encryption key. This is accomplished by utilizing message authentication codes or keyed hashes that combine the plaintext with the secret key to generate a hash that is used to detect when a file has been tampered by an attacker. This scheme is secure because the only entity that ever has access to plaintext file keys in TransCrypt is the kernel itself that recovers it after an authenticated user has decrypted his file token.

2) Recovery agents: A critical demand in enterprise environments is a recovery mechanism to deal with the loss of a user's private key. An administrator cannot read the contents of files that do not grant access to him explicitly. Hence, we support a *data recovery agent* with the privilege to read any file system object on the encrypted file system. The policies of the organization may ensure that the private key of the data recovery agent is split and entrusted with multiple persons to ensure that a successful subversion of this facility can only be undertaken with the collusion of all the concerned administrators, which may be difficult to achieve in practice.

3) Minimum trust model: TransCrypt employs a *minimum trust model*, trusting the fewest number of entities for correct operation. The kernel implements all the cryptographic processes and is hence completely trusted. A malicious kernel may trivially leak encryption keys or sensitive plaintext data present in its address space. Therefore, the superuser account is only partially trusted. Although it would be easy for a malicious administrator to substitute the kernel image with a malicious version over a system reboot, our design protects against a more common scenario in which an attacker temporarily gains root privileges. TransCrypt is thus immune to the key leak attack and other threats by avoiding the centralization of power and responsibility with the administrator. User-space processes, even those running with superuser privileges, are untrusted. The division of functionality between the kernel and the user-space components of TransCrypt has been done suitably to ensure consistency with our threat and trust models so that all security rests with the kernel and untrusted user-space processes merely perform non-cryptographic jobs. Thus, even if those user-space components are compromised or replaced with malicious versions, the system's security is not compromised.

4) Smart cards: TransCrypt supports the use of *smart cards* as a trusted tamper-proof hardware authentication mechanism for users. This provides the highest grade of security wherein a file encryption key can be recovered only after being decrypted on a user's smart card to ensure that the private key is never sent out of the smart card. This feature, however, comes at a small run-time overhead when opening any encrypted file. Also, issuing and deploying smart card infrastructure may not be feasible for some organizations. TransCrypt leaves this choice with the end-user who must evaluate his security threshold and may configure TransCrypt to use alternative less secure mechanisms such as storing the users' encrypted private keys on the disk or on USB thumbdrives.

5) Remote access: Encrypted volumes may be shared over the network in most organizations. TransCrypt's design enables secure remote access to such shared file systems and integrates a protocol for communication between a client system and a server component such that a user's file metadata token is appropriately routed to the client system to be decrypted whenever opening an encrypted file. The encryption of file system data over the network itself is not considered by TransCrypt, as it is the job of network encryption systems like *IPsec* to secure such traffic.

6) Performance: The system's design must be least intrusive in the normal working of the protected file system. Often, the demands of performance are orthogonal to those of security. The system needs to balance the two. For example, we may choose to keep plaintext file data in the page cache. Clearly, this provides performance benefits by avoiding repeated decryption of file contents that have been read and decrypted once already. But an attacker with superuser privileges can launch the page cache attack to read this plaintext. TransCrypt's flexible design leaves this choice with the end-user who can specify whether to maintain plaintext or ciphertext file data in the page cache as a configuration parameter at the time of creating new encrypted volumes.

7) Other issues: Incremental differential backup software copy only the changes detected in a file system since the last backup. A typical solution may traverse the file system to detect files whose 'last modified time' falls after the last backup, thus requiring access to metadata such as filenames stored in directory entries and timestamps in the corresponding inode. However, an encrypting file system must not leave metadata exposed to avoid leaking information. TransCrypt chooses to encrypt directories, thus ruling out backup software operating in such file-by-file incremental mode.

We choose not to support file access control based on *user groups*. The owner of a file must explicitly share it with other users on an individual basis. Although groups can be easily supported using methods such as group keys (keypairs associated with groups) or reference counts, their introduction leads to messy administrative overheads making them unusable in practice. Also, the *swap* partition on a disk may contain fragments of sensitive data. TransCrypt does not encrypt swap, but other solutions for the same exist [10] that may be used alongside TransCrypt for this functionality.

IV. Cryptographic Design and Key Management

This section describes TransCrypt's key management scheme independent of operating system details. First, the entities and keys that lie at the heart of TransCrypt are introduced. We then explain the cryptographic metadata format and related mechanisms.

A. Key Management Scheme

The main entities in TransCrypt are the individual files, the file system and users. The superuser account and data recovery agent are like any other users. The kernel is the primary active agent of the system, implementing all cryptographic processes and key management.

1) File encryption keys: Files are automatically encrypted using random secret *per-file encryption keys* generated at the time of their creation. We denote a file encryption key (FEK) as k. For bulk encryption of file contents, any block cipher in an appropriate mode such as Cipher Block Chaining (CBC) may be used, where the ciphertext is chained only within one block of the underlying file system to support random access within the file. This avoids re-encryption overheads due to minor changes at the start of a file or decrypting the entire file to read the last few bytes. A new Initialization Vector (IV) must be used for every file system block, which may be derived from the physical block number itself, using a secure method such as *Encrypted Salt-Sector IV* [11]:

$$IV(block) = E_{salt}(block) \text{ where } salt = H(k)$$

Here, *block* is the file system block number and E_{salt} represents symmetric encryption using *salt* as the key. The *salt* is derived from the FEK k using a hash function H whose blocksize equals the keysize of the encryption algorithm E. Encrypting files on a block-by-block basis also enables TransCrypt to transparently support *sparse files* containing holes.

TransCrypt incorporates a scheme utilizing keyed hashes similar to the one proposed by eCryptfs [12] to provide file *integrity* and prevent undetectable modification of file contents without the knowledge of its secret encryption key. A separate message authentication code is computed for every file system block used by a file's contents and metadata such as access control lists. Finally, a top hash is also computed over all the other hashes. Such a scheme employing a hash list and a top hash enables random write access to any part of the file without requiring the recalculation of any single hash over the entire file. The top hash prevents an attack where the contents of a file as a whole may be modified by swapping (but not altering) two complete underlying blocks so that their individual block-level hashes remain the same.

2) User keypairs: At least one *public and private keypair* is associated with every user of the authentication domain. A single user is allowed to possess more than one keypair, which would be useful when transitioning from an old keypair to a new one. This keypair enables the design of a hybrid cryptosystem in which separate copies of a file encryption key may be encrypted with the public keys of all users who have access to the file. When a file is created, such metadata entries are created and stored only for the owner and those users with default access to the file. Later, such metadata entries are also created for other users when they are granted access to a file. We denote the public key of a user with ID *uid* as KU_{uid} and the corresponding private key as KR_{uid}.

The public keys of all users are encapsulated in X.509 certificates signed by a certification authority trusted by the organization. An *authentication domain-wide certificate repository* containing these user certificates is established at a publicly known network location. This enables the owner of a file system object to grant access to another user transparently, without the need for any communication between them. A successful attack and subversion of the repository clearly does not compromise the security of TransCrypt, because certificates are always verified before using their public keys. A corresponding *trusted certificates store* is maintained by all computer systems that contain an encrypted file system. This store contains the certificates of trusted certification authorities used to verify user certificates. The private key parameters of users may either be stored on smart cards or on a separate disk or USB thumbdrive after encryption.

3) File system key: The *file system key* is a secret key specific to a particular encrypted file system. It is used and managed solely by the kernel. An encrypted copy of the file system key is stored in the superblock of the encrypted volume. The file system key is denoted as FSK. FEKs are first encrypted using the FSK before being encrypted using a user's public key.

B. Cryptographic Metadata Format

A cryptographic context must be associated with every file, containing its FEK in the form of separate per-user tokens, similar to the scheme used by *Pretty Good Privacy* [13] when encrypting mail intended for multiple recipients. The per-user records have the following 3-tuple schema:

$$\begin{aligned} & uid : certid : token \\ \text{where, } token &= E_{KU_{uid}}(E_{FSK}(k)) \\ \text{therefore, } k &= f(token,\ KR_{uid},\ FSK) \end{aligned}$$

Here, *uid* is the user's UID (for that authentication domain) and *certid* is a string that uniquely identifies the user's certificate. A single user may have multiple keypairs and thus the pair <*uid*, *certid*> uniquely identifies a particular keypair. E_{FSK} represents symmetric encryption using a block cipher with FSK as the key and $E_{KU_{uid}}$ represents public key encryption using the appropriate algorithm corresponding to KU_{uid}. Thus, decrypting the FEK from a token is a function requiring three inputs: the token, the user's private key and the file system key. This combines the security of multiple entities. Clearly, a malicious user is unable to launch an offline key leak attack on the token without first compromising and gaining access to FSK.

The token format has been constructed to incorporate a form of *blinding*. TransCrypt does not trust the user-space and hence a file encryption key must only be known to the kernel of a running system. However, it may still be necessary to send cryptographic metadata outside the kernel for various purposes. In such cases, the security of the FEK must be maintained by blinding it using a factor known only to the kernel. Here, symmetric key encryption is the blinding operation and FSK is the blinding factor.

The file system key FSK is a novel feature of TransCrypt that is not provided by other solutions. It may be recalled that any scheme that encrypts FEKs using only public keys is vulnerable to the key leak attack. Also, employing daemons without blinding makes the system susceptible to user-space exploits. The file system key provides blinding and prevents the key leak attack. However, it makes a file's cryptographic context dependent on the particular encrypted file system on which it is stored. This is not an issue when the file is copied from one encrypted volume to another, in which case a new FEK would be generated for the target file and the target volume's FSK would be used for blinding. But the file and its metadata cannot be simply plucked out of an encrypted file system and sent by mail to a recipient while remaining encrypted. We believe that the security benefits of the file system key far outweigh this feature. Moreover, email security is clearly not the job of an encrypting file system. Software such as PGP may be used for the same.

V. TransCrypt Architecture

Figure 1 illustrates the architecture of TransCrypt. The platform chosen for the first reference implementation is Linux kernel version 2.6.11. In this section, we describe the software modules that need to be implemented in the Linux kernel and allied user-space support utilities. Finally, we explain the installation and operation of the entire system.

A. Authentication Domain-wide Certificate Repository

The public certificate repository is maintained at a single network location and made available throughout the authentication domain using an appropriate service such as NIS, NFS or LDAP. In the case of NFS, the certificate repository server exports a directory that is mounted by systems containing encrypted volumes as */etc/efscertificates/*. It has a single file called *certtab* containing user records of the format:

$$uid : cert$$

Here, *uid* is the user's UID in the authentication domain and *cert* is his Base64-encoded X.509 certificate in PEM format. Every user (including the data recovery agent) has exactly one such record. When a user is transitioning from an old keypair to a new one, only the latest certificate is maintained in the repository. Whenever a user's public key is needed, the corresponding certificate is retrieved from the repository and verified using the local trusted certificates store. This avoids administrative and performance overheads associated with certificate revocation lists and online certificate status protocols.

B. Cryptographic Metadata Storage

TransCrypt provides user-level access control and hence chooses to integrate the per-user metadata tokens with an encrypted file's access control list itself. In Linux, POSIX ACLs are implemented using extended attributes [14]. It must be noted that TransCrypt does not support groups and hence the ACL of an encrypted file must only utilize 'named user' entries [14]. We augment such entries with two more fields, *certid* and *token*, introduced in the previous section. Hence, a typical named user ACL entry in TransCrypt is:

$$user : username : rwx : certid : token$$

Here, *rwx* are the permissions of user *username*. The 'others' ACL entry for all files in the encrypted volume is set to null permissions. This ensures that a separate ACL entry (and *token*) exists for every user who can access the file.

C. Linux Kernel Hooks

Kernel-based encrypting file systems introduce a *layer of indirection* between the upper virtual file system layers and the low-level block device driver to implement the encryption and decryption processes. We believe encryption is merely a *property* of any underlying file system that can be turned on or off using a *mount option*. When specified, the 'encrypt' mount option enables kernel hooks that implement the cryptographic processes and key management transparently. When not specified (the default case for unencrypted volumes), these hooks are simply bypassed. Such an approach maintains the same on-disk partition layout for encrypted and unencrypted volumes. It gives the added benefit that encrypted backups may be taken simply by remounting the encrypted file system and turning off the 'encryption' functionality. The choice of whether smart cards or alternative means are used for private key storage is also specified as a mount option. The bulk of

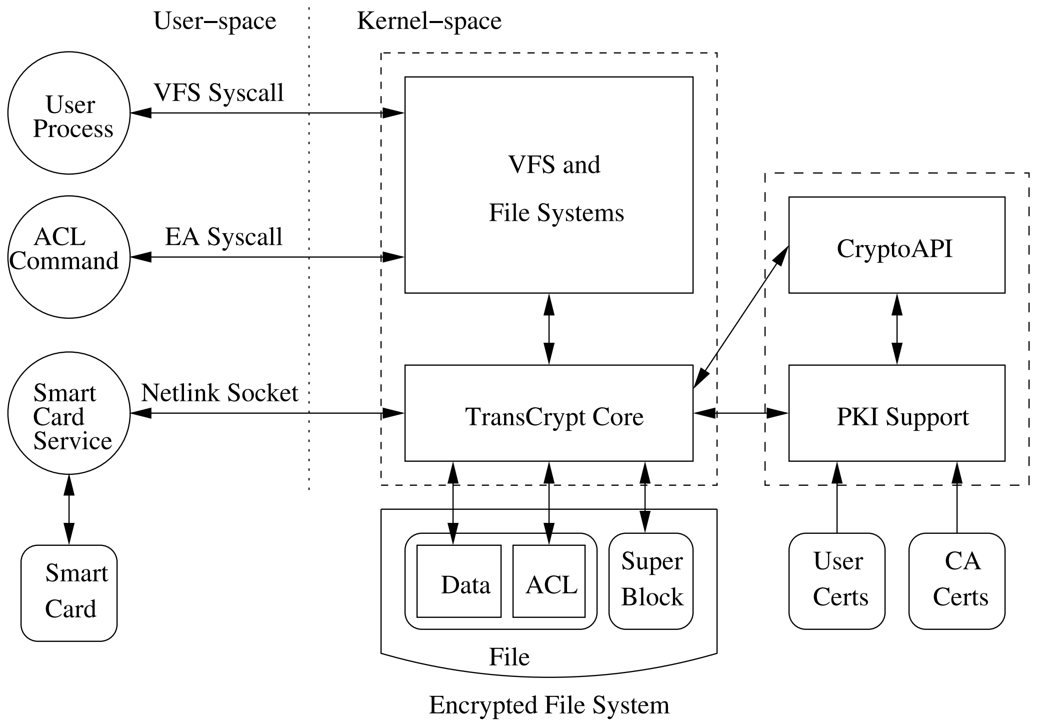


Fig. 1. Overview of TransCrypt architecture

TransCrypt's implementation is independent of the underlying file system type. Only the on-disk superblock and ACL data structures need to be altered for all supported underlying file systems. These changes are fairly basic and easily duplicated for various file system types. Also, all cryptographic processes and key management are performed in the kernel and user-space utilities are only support applications. The kernel patches required by TransCrypt are described later in this section.

D. Public Key Cryptography Support

Public key cryptography support has been incorporated into the kernel's native CryptoAPI [6] as a part of TransCrypt. A generic asymmetric key API has been integrated that provides an interface to call public key encryption, decryption, signature generation and verification functions from within the kernel. Public key cryptosystems such as RSA are implemented as kernel modules underlying the generic asymmetric API. The *Multi Precision Integer* support patch recently ported to the kernel [15] is used to provide the underlying math functions. However, a scheme that calls on a user-space service for only certificate verification is as insecure as one in which all public key management and operations occur in user-space. Thus, a skeletal Public Key Infrastructure support library must also be integrated into the kernel that provides functionality to decode and parse Base64-encoded PEM format X.509 certificates, verify their validity and extract the public key.

It may be argued that PKI support and public key cryptography should not be implemented in the kernel due to their complexity, computational costs and space costs. However, we choose to do so because our threat model does not trust user-space key management daemons which can be easily masqueraded or attacked in practice. Incorporating full PKI support in the kernel also simplifies the design and increases performance by avoiding bouncing around from the kernel to user-space and back for system calls such as *open* and *creat*.

The kernel also maintains the trusted certificates store that contains the certificates of trusted certification authorities. It may be implemented as a local unencrypted file called */etc/efstrustedcerts* containing records of the format:

$$CAid : cert$$

Here, *CAid* is a string that uniquely identifies a particular CA and *cert* is its Base64-encoded X.509 certificate in PEM format. In the simplest case, the organization runs its own special root CA to issue TransCrypt-specific user certificates. In this case, the trusted certificates store contains only one record corresponding to the organization's CA. On the other hand, the organization may issue certificates signed by a commercial CA (such as Verisign). It is also possible that user certificates are not directly signed by a root CA. In case a hierarchy of intermediate CAs exists, *efstrustedcerts* stores the certificates of the root CA as well as all the intermediate CAs. An intermediate CA certificate is verified using the trusted root CA certificates already present in this file before being added to the store. However, the security of this system is contingent upon the integrity of the local file that contains the trusted certificates. Clearly, this is not acceptable if the superuser account is kept out of the trust model. Hard-coding the public keys of well known CAs in the kernel itself is a possible solution, but it reduces the maintainability of the code. A better solution is to maintain the trusted certificates (or public keys) in a separate local file whose integrity is protected with a message authentication code computed using a key known and used only by the kernel.

An implementation issue here is the overhead due to accessing *certtab* and *efstrustedcerts* whenever new encrypted files are created. A more fundamental issue is accessing and reading configuration files from within the kernel itself. Although possible, such a design is generally deprecated as it reduces the maintainability of the kernel. A solution is to use an alternative mechanisms such as *configfs*. For example, a special user-space program may be executed once whenever the system is booted up that feeds the configuration data to the kernel. An optimization could be to run this module periodically (or on-demand) to parse the configuration files and maintain their information in appropriate kernel-space data structures to minimize run-time overheads. Maintaining such a cache of certificates or verified public keys in the kernel also ensures seamless operation when the system is disconnected from the network apart from providing the obvious performance benefits.

E. Key Acquisition for Token Decryption

The use of a user's private key by the kernel to decrypt the per-user tokens is trivial when the private parameters are stored on the disk or a USB thumbdrive. However, TransCrypt also supports the use of smart cards to store the private keys of users and perform operations requiring them. The Linux kernel does not provide a smart card interface library to enable kernel modules to directly interact with smart cards. Ongoing projects such as *SmartK* [16] aim to integrate smart card support into the kernel but TransCrypt utilizes a user-space daemon for this purpose. The architecture of TransCrypt has been designed to partition responsibilities between the kernel and the support service such that all cryptographic operations are performed by the kernel.

The kernel uses Netlink sockets to send per-user metadata tokens to the smart card service when opening encrypted files. The service forwards the token to the smart card that decrypts it using the user's private key and sends back the response that consists of the FEK still encrypted with the file system key *FSK* to the kernel. Although this blinding prevents daemon masquerading, the blinded FEK is still vulnerable to replay attacks. A malicious daemon may log responses received from the smart card and attempt to replay them in future. This attack may be thwarted by establishing an authenticated and encrypted session between the kernel and the smart card before any data exchange. This protects against eavesdropping and also prevents replay attacks on the physical channel between the computer and the smart card. The PKI-capable kernel acts as the trusted end point in the authentication and session key establishment protocol with the smart card. The user-space daemon merely routes tokens and their decrypted responses (blinded FEKs) on the secure channel between the kernel and the smart card and may thus be an untrusted process.

An issue is acquiring the file key from a per-user token when remotely accessing and opening encrypted files on a networked file system. The support daemon on the file server also routes the token from the encrypted volume to the corresponding smart card support daemon on the client system for decryption using the user's smart card (the decrypted blinded FEK is similarly routed back to the server) so that the file encryption key decrypted from it can then be used to encrypt or decrypt file data on the server. An authenticated and encrypted session must first be established between the file server kernel and the remotely inserted smart card. This allows TransCrypt to transparently serve multiple simultaneous remote requests, thus satisfying a basic demand of enterprise environments.

F. ACL Manipulation Commands

Storing a file's cryptographic metadata in the ACL itself offers several transparency benefits. The design becomes cleaner and the implementation effort reduces to merely patching the ACL manipulation mechanisms to ensure consistency between user entries and per-user tokens by creating and storing an additional per-user token every time a user is granted access to a file system object by its owner. Whenever a user's access is revoked, the corresponding ACL entry (that includes the token) is simply removed.

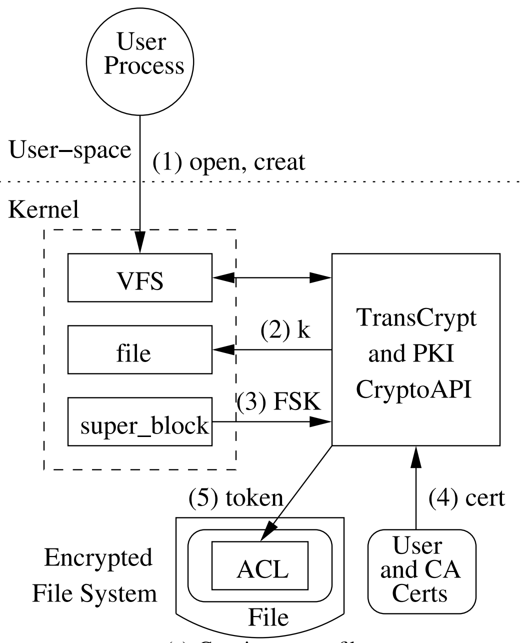
POSIX ACL manipulation in Linux [14] is implemented using library functions provided by *libacl*. The kernel does not yet provide ACL system calls but uses the extended attributes interface to copy ACLs between user-space and kernel-space. The *libacl* library and the *chacl* and *setfacl* commands that modify the ACL of a file must be patched to support the augmented ACL entry structure defined earlier. Modifications must also be made to prevent the owner of a file system object from removing the data recovery agent entry or specifying non-null permissions for the 'others' entry in the ACL.

Blinding protects the file encryption key from being leaked when tokens are handled by untrusted user-space programs. Also, including a separate keyed hash to protect the integrity of the ACL prevents an attack wherein a malicious user may attempt to re-encrypt the blinded FEK (obtained after decrypting his token) using another user's public key and insert the resultant token with an illegal entry into the file's ACL. The keyed hash ensures that only the kernel of a running system can modify an ACL after getting the owner's token decrypted in response to a access granting command. However, this requires the kernel to do ACL manipulation work. Hence, TransCrypt proposes to shift ACL manipulation into the kernel and introduce the necessary system calls.

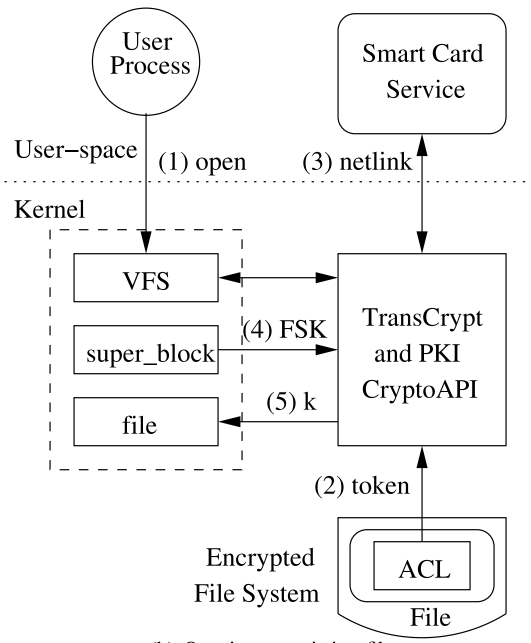
G. TransCrypt in Action

1) Enterprise deployment: The following pre-requisite activities must first be carried out:

- A public and private keypair must be generated for all users in the authentication domain who require access to encrypted file systems.
- The public keys must be signed by an appropriate CA and the certificates made publicly accessible in the certificate repository. The trusted certificates store must be



(a) Creating a new file



(b) Opening an existing file

Fig. 2. File operations: creation and access

established for all computer systems that mount encrypted volumes. Smart cards, if used, must be issued to users.

- The data recovery agent account must be established in the authentication domain and its certificate added to the repository. The corresponding private key may be split into multiple smart cards and issued to different persons.

2) Encrypted file system creation: The on-disk superblock structure of the underlying file system and the corresponding *mkfs* command are suitably modified to take the following actions when creating an encrypted volume:

- The block cipher algorithm to be used for symmetric key encryption, chaining mode, keysize (of FEKs and the file system key), IV generation method and the user's choice regarding page cache encryption are specified on the *mkfs* command line. These parameters are appropriately encoded and stored in the superblock.
- The file system key *FSK* is randomly generated and encrypted using key material derived from a passphrase (or an external trusted hardware to avoid trusting the administrator). The result is stored in the superblock.
- The DRA is added as a named user with read and execute permissions to the default and access ACLs [14] of the root directory of the encrypted volume. Additionally, the permissions for the 'others' entry are set to null. This recursively ensures that all further subdirectories and files created in the encrypted file system would automatically inherit these two entries in their access control lists.

3) Mounting an encrypted file system: An encrypted file system is mounted by specifying the *encrypt* option. TransCrypt is integrated with POSIX ACLs and hence the *acl* mount option must also be specified. Also, the mechanism that is used to store private keys is specified as a mount option. During *mount*, algorithm parameters are copied from the on-disk superblock of the underlying file system to the kernel's in-core superblock structure. The encrypted *FSK* is also read from the on-disk superblock, decrypted using the same mechanism used at the time of creation and copied into the in-core superblock.

4) File creation and access: The general architecture of TransCrypt for *open* and *creat* is shown in Figure 2. The following actions are taken whenever a new file (or directory) is created in an encrypted file system:

- A file encryption key k is randomly generated. It is put into the entry corresponding to this instance of *open* (or *creat*) in the VFS open *file* structure. Also, the *FSK* is read from the kernel's in-core *super_block* structure associated with the underlying volume. It is used to encrypt the FEK using the specified algorithm parameters.
- The kernel determines the UID of the file's owner from the *current* process context. This is used to access the owner's certificate from the repository. The certificate is first verified and then its public key is used to encrypt the result of the previous step. The resulting *token* is copied into the corresponding field of the owner's ACL entry.
- The above step is repeated for all the users present in the default ACL inherited by the file (or directory) from its parent directory.

The actions taken when opening an existing encrypted file are as follows:

- The kernel determines the UID of the *current* process context and checks the user's permissions to open the file using the appropriate ACL entry. If successfully verified,

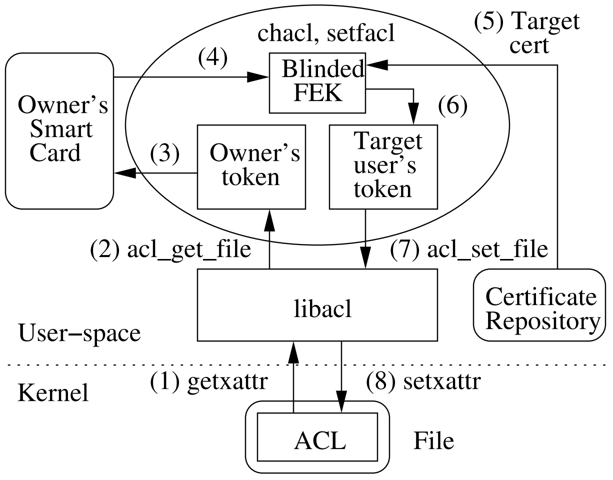


Fig. 3. Granting file access to other users

the corresponding token is pulled out of the ACL entry and decrypted using either the smart card or the user's private key acquired from the disk.

- The file system key is read from the in-core *super_block* and used to decrypt the result of the previous step.
- If the user is genuine, we now have the original FEK k used to encrypt the file. It is copied into the *file* structure corresponding to this call of *open*. If a wrong smart card was inserted, an incorrect FEK would be decrypted.

5) Reading and writing file data: Other than *read* and *write*, file data may be accessed using the *mmap* system call. The 2.6 series kernels incorporate a unified page cache and *bio* infrastructure that provide a common interface to the disk regardless of the system calls used. TransCrypt takes advantage of these unified interface to hook in the encryption and decryption processes.

A file's contents are accessed after it has been opened. The FEK already present in the corresponding *file* structure is used to do encryption or decryption transparently. Implementation issues such as locking and synchronization determine the exact placement of the encryption and decryption hooks in the kernel. The implementation effort in TransCrypt has proceeded in an exploratory fashion and evolved towards the best alternative. A preliminary version plugged encryption and decryption at the page cache layer around the *submit_bh* function. This approach leads to individual *bio* requests being submitted for every file system block, thus causing a significant performance degradation of about 40%, as determined experimentally. This preliminary implementation approach is being discarded in favour of a design that uses the *workqueue* interface, thus enabling the coalescing of multiple *bio* requests to avoid the aforementioned overhead. Separate per-CPU kernel threads created in advance are executed in *user process context*. After encrypted data is read from the disk, the callback function executing in *hard IRQ context* merely enqueues the actual decryption job in the corresponding kernel thread's workqueue. The implementation of dm-crypt [5] uses a similar design and integrating TransCrypt with it may be explored in the future.

Test	Parameter	Read (sec)	Write (sec)
TransCrypt	Elapsed Real Time	30.193	47.651
	User CPU Time	0.061	0.074
	System CPU Time	14.273	2.499
Normal	Elapsed Real Time	9.687	13.104
	User CPU Time	0.065	0.071
	System CPU Time	0.754	2.458

TABLE I

PERFORMANCE OF TRANSCRYPT AGAINST NORMAL UNENCRYPTED FILE OPERATIONS

In case encrypted file data must be maintained in the page cache, the encryption and decryption processes are implemented in the actor functions that copy the data between the kernel's page cache and the user-space buffers specified by the application to the *read* or *write* system calls.

As discussed earlier, TransCrypt also utilizes keyed hashes to enforce file integrity. The hash list for a file consists of a separate message authentication code computed for every file system block used by that file's contents (including the block containing its ACL) that are verified for every read and updated on every write operation on the corresponding block. The top hash, computed over the hash list itself, must be verified and updated for every read and write call respectively.

The present architecture of TransCrypt does not use the *in-kernel key management service* [17] recently introduced in Linux and stores encryption keys directly in VFS objects. Future versions of TransCrypt may integrate with kernel keyrings. Another implementation tweak could be to store the secret keys used by TransCrypt in kernel structures in a key schedule form. This avoids the repeated overhead of converting the plaintext key into a key schedule at every *read* or *write* and speeds up encryption or decryption at the cost of greater space. However, this is not yet supported by the present architecture of the kernel CryptoAPI and hence left out of our design.

6) Granting and revoking file access: TransCrypt utilizes a keyed hash to protect the integrity of a file's ACL. Hence, ACL manipulation system calls must be provided by the kernel to make the mechanism of file access granting and revoking similar to the handling of *open* and *creat*. Presently, however, ACL manipulation in Linux is through the *libacl* library that exports ACL interface functions to the *chacl* and *setfacl* commands and in turn communicates with the kernel using extended attribute system calls. Figure 3 illustrates the flow of control and data whenever the owner of an encrypted file grants access to another user.

- The owner's token is read from his ACL entry and decrypted using his smart card or private key to produce the blinded FEK.
- The target user's certificate is fetched from the repository and verified. The public key is extracted from it.
- The blinded FEK (the FEK encrypted with the file system key) is directly re-encrypted using the target user's public key to get his per-user token.
- The *certid* and *token* fields of the target user's newly created ACL entry are updated and stored.

Feature	CFS	Windows EFS	dm-crypt	eCryptfs	TransCrypt
Design Approach	user-space	hybrid	kernel	hybrid	kernel
Key Management	common mount-wide key	per-file keys and per-user keypairs	common mount-wide key	per-file keys and per-user keypairs	per-file keys and per-user keypairs
Recovery or Escrow Agent	no	yes	no	yes	yes
Superuser Account	trusted	trusted	trusted	trusted	untrusted
User-space	trusted	trusted	trusted	trusted	untrusted
FEK Blinding	no	no	no	no	yes
Integrity	not supported	not supported	not supported	supported	supported
Sparse Files	supported	supported	supported	not supported	supported
Smart Card Support	no	no	no	no	yes
ACL Integration	no	no	no	no	yes

TABLE II
FEATURES OF TRANSCRYPT VERSUS OTHER ENCRYPTING FILE SYSTEMS

When revoking access, the target user's ACL entry, including the token, is simply deleted. Re-encrypting the file with a new key is not necessary.

H. Procedures

We now discuss various administrative and maintenance procedures that must be followed by the organization.

1) Change of user keypair and smart card: This case is handled using a special utility implemented in user-space. When a user changes his keypair, the old certificate in the public repository is immediately replaced with the new one to ensure that his token for any newly created files would be generated using the new public key for encryption. Then, the special tool must be run through the entire encrypted file system to find all files that are accessible by the user. The user's token for such files is extracted from the corresponding ACL entry and decrypted using the old smart card. The resultant blinded FEK is re-encrypted using the new public key. This new token is then stored back into the ACL. This tool runs with special privileges so that the ACLs of files that are not owned by the user may also be modified. Because this operation may potentially take a lot of time, TransCrypt allows it to be run in the background and provides for a period of transition during which the user may possess both the keypairs. The public certificate repository, however, always only contains the latest public key certificate for every user.

2) Backups: Encrypted backups of the full image of a file system are taken by remounting the encrypted volume *without* the 'encrypt' option in read-only mode. Any backup software may be used for this purpose after turning off the incremental mode of operation. Because the cryptographic context of a file is dependent on the file system blocks containing it, the restore process must be applied using the full source image to the same target file system. Thus, TransCrypt disables the recovery of individual files to other file systems, preventing the leak of data from stolen backups. Also, it must be noted that FEKs are stored as tokens generated after encrypting them with the FSK, thus making them dependent on the file system. The FSK is itself stored encrypted in the superblock of the volume and must also be backed up.

3) Data recovery: The recovery process employs a special user-space solution. A separate tool must be implemented that may require multiple administrators to insert multiple smart cards simultaneously to reconstruct the private key of the data recovery agent. It is then used to decrypt the files to be recovered.

VI. PERFORMANCE EVALUATION AND FEATURE COMPARISON

A. Analysis of Results

A brief analysis of the performance results of TransCrypt follows, in terms of the time overhead to read and write file contents on an encrypted file system utilizing the AES block cipher with 128-bit keys against the normal unencrypted case, as shown in Table I. The system under test had an Intel Pentium 4 CPU running at 3 GHz with Hyper-Threading and 2 GB RAM. For each test, the numbers indicate the elapsed real time, user CPU time and system CPU time outputs of the *time* command when reading or writing a file of size 512 MB. These results do not consider the overhead due to file integrity checks or smart card access and assume the availability of a user's public and private key parameters with the kernel itself. Although a degradation of more than 200% has been observed, the performance is likely to improve as the implementation is refined and optimized, as explained in the previous sections. A more formal analysis and comparison with existing solutions may be undertaken in the future when the implementation stabilizes as planned. Moreover, the existing performance overhead is still better than most user-space cryptographic file systems that degrade performance by several times [1] and may be acceptable in typical end-use scenarios given the unique security benefits offered by TransCrypt.

B. Comparison with Related Work

Table II provides a tabular comparison of the usability features and security benefits offered by TransCrypt against Cryptographic File System (CFS), the native Microsoft Windows Encrypting File System (Windows EFS), dm-crypt and eCryptfs, clearly bringing out the differentiating aspects of TransCrypt.

VII. Summary

Data security has emerged as a critical need in both personal and multi-user scenarios. Most existing encrypting file systems do not meet the diverse requirements of security and usability, due to the lack of flexible key management, fine-grained access control and security against a wide range of attacks.

TransCrypt provides a solution that is both secure and practically usable. We assume an attacker has the capability to launch attacks that are beyond the threat models of existing systems and propose solutions to such threats. We make a crucial distinction between the kernel and user-space from a security perspective. Employing a completely kernel-space implementation enables us to avoid trusting the superuser account and protect against various user-space attacks. Integration of cryptographic metadata with POSIX ACLs greatly simplifies key management. Enterprise-class requirements such as integrity, data recovery, backups and remote secure access to shared file systems are also supported. Future versions of TransCrypt would explore deeper integration with trusted platform module hardware, especially in the areas identified in this paper, to further minimize the number of trusted entities and provide even greater security.

References

[1] M. Blaze, "A Cryptographic File System for UNIX," in *Proceedings of the ACM Conference on Computer and Communications Security*, Fairfax, VA, USA, Nov. 1993, pp. 9–16.

[2] Encfs - Virtual Encrypted Filesystem for Linux. [Online]. Available: http://encfs.sourceforge.net/

[3] How the Encrypting File System Works. [Online]. Available: http://technet2.microsoft.com/WindowsServer/en/Library/997fdd99-73ec-4041-9cf4-1370739a59201033.mspx

[4] Apple Mac OS X FileVault. [Online]. Available: http://www.apple.com/macosx/features/filevault/

[5] dm-crypt: a device-mapper crypto target for Linux. [Online]. Available: http://www.saout.de/misc/dm-crypt/

[6] J.-L. Cooke and D. Bryson, "Strong Cryptography in the Linux Kernel," in *Proceedings of the Linux Symposium*, Ottawa, Canada, July 2003, pp. 139–144.

[7] M. A. Halcrow, "eCryptfs: An Enterprise-class Encrypted Filesystem for Linux," in *Proceedings of the Linux Symposium*, Ottawa, Canada, July 2005, pp. 201–218.

[8] E. Zadok, I. Badulescu, and A. Shender, "Cryptfs: A Stackable Vnode Level Encryption File System," Department of Computer Science, Columbia University, Tech. Rep. CUCS-021-98, 1998.

[9] Cryptographic signatures on kernel modules. [Online]. Available: http://lwn.net/Articles/92617/

[10] N. Provos, "Encrypting Virtual Memory," in *Proceedings of the USENIX Security Symposium*, Denver, CO, USA, Aug. 2000, pp. 35–44.

[11] C. Fruhwirth. New Methods in Hard Disk Encryption. [Online]. Available: http://clemens.endorphin.org/nmihde/nmihde-letter-os.pdf

[12] M. A. Halcrow, "Demands, Solutions, and Improvements for Linux Filesystem Security," in *Proceedings of the Linux Symposium*, Ottawa, Canada, July 2004, pp. 269–286.

[13] S. Garfinkel, *PGP: Pretty Good Privacy*. O'Reilly Media, 1995.

[14] A. Grunbacher, "POSIX Access Control Lists on Linux," in *Proceedings of the USENIX Annual Technical Conference (FREENIX Track)*, San Antonio, Texas, USA, June 2003, pp. 259–272.

[15] D. Hardeman. (2006, Jan.) [PATCH] add multi-precision-integer maths library. Linux Kernel Mailing List. [Online]. Available: http://lkml.org/lkml/2006/1/26/295

[16] SmartK: a smart card framework for the Linux Kernel. [Online]. Available: http://smartk.dia.unisa.it/

[17] D. Howells. (2004, Aug.) [PATCH] implement in-kernel keys & keyring management. Linux Kernel Mailing List. [Online]. Available: http://lkml.org/lkml/2004/8/6/323