Paranoid: A Global Secure File Access Control System

Fareed Zaffar and Gershon Kedem
Department of Computer Science, Duke University
Box 90129, Durham, NC 27708
{fareed,kedem}@cs.duke.edu

Ashish Gehani
Department of Computer Science and Engineering, University of Notre Dame
384 Fitzpatrick Hall, Notre Dame, IN 46556
ashish.gehani@nd.edu

Abstract

The Paranoid file system is an encrypted, secure, global file system with user managed access control. The system provides efficient peer-to-peer application transparent file sharing. This paper presents the design, implementation and evaluation of the Paranoid file system and its access-control architecture. The system lets users grant safe, selective, UNIX-like, file access to peer groups across administrative boundaries. Files are kept encrypted and access control translates into key management. The system uses a novel transformation key scheme to effect access revocation. The file system works seamlessly with existing applications through the use of interposition agents. The interposition agents provide a layer of indirection making it possible to implement transparent remote file access and data encryption/decryption without any kernel modifications. System performance evaluations show that encryption and remote file-access overheads are small, demonstrating that the Paranoid system is practical.

1. Introduction

Computers around the globe are increasingly being used to create, store and share confidential and private data. Keeping such information secure is an increasingly difficult job. As more and more computers get interconnected using an untrusted and hostile network, the set of possible security attacks has increased manifold. Security concerns are also a major barrier to information sharing in a global computing environment. In-spite of security risks, the added functionality provided by network interconnectivity and sharing is so compelling that most people and companies willingly assume these risks. Moreover, current systems are poor at providing global information sharing mechanisms.

Traditional operating systems provide access protection and controlled access to resources. However, these mechanisms have proved to be fragile at best. Intruders routinely exploit programming errors, system design errors, errors in operating system settings or "social engineering" to gain super-user access rights, bypassing system protections.

New errors are introduced into the operating system just as fast as old errors are discovered and corrected. The time between a vulnerability being identified and its exploit being generated has reduced dramatically over the years. Operating systems have become large and complex as new devices and operating system services are continually added or upgraded. Every operating system update potentially introduces new errors that allow intruders to bypass the operating system protection mechanisms. This unfortunate steady-state of insecurity is not likely to improve any time soon.

Another major design weakness of traditional operating systems is the need for privileged access. A super-user account is a compromise, introduced in order to enable inter-user operating system services and facilitate administrative tasks such as, file-system and account management, email, files backup, etc. This compromise creates opportunities for attacks on the system. Most serious security breaches happen when an attacker finds a way to gain super-user privileges, totally bypassing the operating system access control mechanisms.

This paper introduces Paranoid, a global file Access Control system that allows users to selectively, securely, and easily share information with others, even those they don't know and don't have prior trust relationships. It provides users with a global computing environment, without the fear of compromising the security of information they consider private or privileged. Each user is able to grant

selective file access privileges to others outside his administrative domain without having to create accounts or grant outsiders any user privileges.

The file system lets users define access groups. A person with Paranoid access privileges can access files regardless of whether they are trusted local users or outsiders. One of the key features of Paranoid access control is that each group member has cryptographic access to the group accessible files without possessing a shared group secret. The Paranoid secure file system is implemented as a toolkit along with a set of dynamically loadable libraries. The file system is implemented at the user level completely. Paranoid is an encrypted file system, but the cryptographic nature of the underlying file system is application transparent through use of interposition agents [20]. The advantage of this approach is that group owners don't need super-user privileges to securely share files with others, and existing applications are used without re-compiling. The choice of using interposition agents limits file system access to dynamically linked programs. However, most modern day UNIX programs are dynamically linked. Statically linked programs can't use the Paranoid file system.

2. Design

2.1. Overview

The overall goal of the Paranoid system is to facilitate global peer-to-peer file sharing with enhanced security and privacy, minimal administrative overhead and application transparency. File contents are locked via encryption and are unlocked only with a correct key. Thus, access control transforms into a key management problem. Users are implicitly authenticated by their ability to gain access to keys. Paranoid uses a novel approach using *transform keys* (detailed in Section 3) to address the key distribution and revocation problems. The transformation key approach eliminates the need for a shared group secret for file sharing. To enhance security, key decryption is performed using a trusted device. We call this trusted device a Personal Security Proxy (PSP) (see section 2.4) and suggest using a commercial handheld computer for this purpose.

2.2. Encrypted Files

Files' contents are locked and made inaccessible by encryption. Paranoid uses a hybrid encryption system. Data are encrypted with a symmetric cipher and symmetric keys are encrypted with public key cipher. Each file is encrypted with a different random key. Since public key ciphers are too slow, a symmetric cipher is used. (Using public key ciphers to encrypt and decrypt the entire file would add significant delays to file operations.) The prototype system is

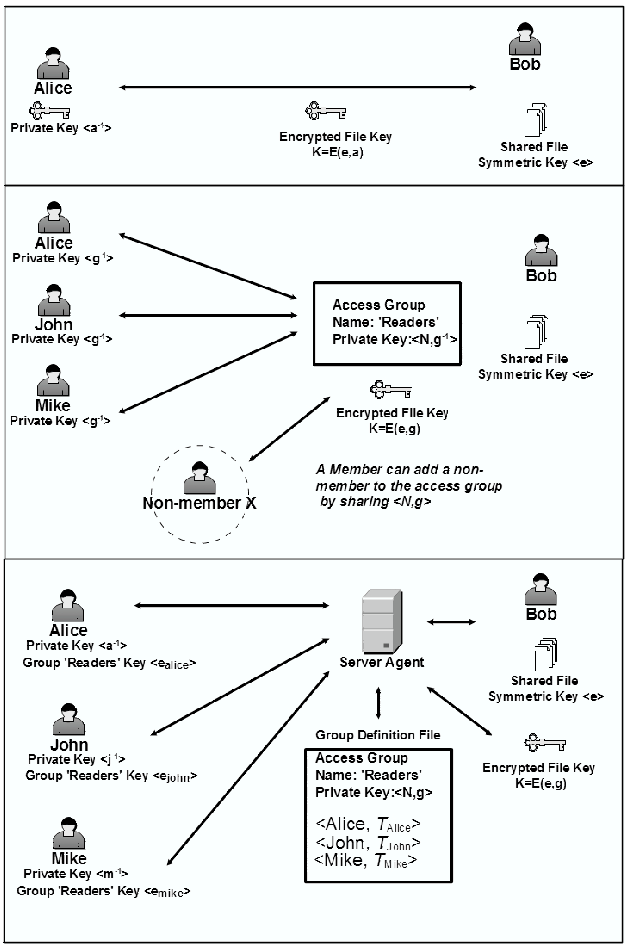

Figure 1. Sharing a file

implemented using DES [14] but any other symmetric cipher could be used, such as AES [15]. Each symmetric key is encrypted with the file owner's public key. For sharing, group access to a file is granted by encrypting the file's symmetric key with the group's public key. This information, along with the file digital signatures, version number and a time-stamp are stored in a header, together with the encrypted file's contents. The headers use XML formatting to store the data. Headers are described in Section 4.1. A super-user can access the encrypted Paranoid files for administrative tasks, such as file backup, but has no access to the files' contents.

2.3. Access Groups and File Sharing

When a file owner $\mathbf{A}$ wants to share a file with person $\mathbf{B}$, the owner can encrypt the file's symmetric key with $\mathbf{B}$'s public key. The encrypted key can be stored along with the encrypted file or sent directly to $\mathbf{B}$. Theoretically, this is all that is needed for file sharing. However, if a group of

people is sharing a set of files, a more efficient method is adopted. The owner of a set of files defines an access group for the files. Group members encrypt files with symmetric keys and encrypt the symmetric keys with the group public key. In this case, the owner is responsible for distributing the group's private key to all group members. This scheme poses logistical problems since explicit key distribution is needed. One solution is to store group access information in a file which can be provided upon request. A group access information file holds group identities along with the group's private key encrypted with the public keys of group members. In this scheme, when a user is trying to access a file, he uses his identity to retrieve the group's encrypted private key from the group information file. He then decrypts it using his own private key. The group's private key in turn allows him to decrypt the file's symmetric key, granting him access to the file's content. The group owner is responsible for group management tasks such as adding, deleting and updating entries.

This scheme is similar to the lockboxes adopted by Cepheus [5] although they used a central group database to distribute keys. Their scheme suffers from an inherent weakness. Not only is the database a central point of vulnerability, but the scheme gives users more rights than necessary. Group membership should only enable access to shared files. With Cepheus, any group member can add new members to the group by disclosing the group secret (private key). Additionally, revoking access rights is difficult. It requires changing the group's public and private keys and the re-encryption of all symmetric keys. Coordinating key changes over the Internet is difficult.

The Paranoid system uses a novel scheme that does not requires the sharing of a group-specific secret. When a user (a group owner) creates a new access group, he creates a new public and private key pair for the group using the RSA public key cipher [17]. He publishes the group public key, that is the modulus N and the public exponent g. All group members use the same modulus N but each group member is assigned a *different random exponent* as a private key. Associated with each group member's private key is a transform key known only to the group owner. When a group member requests access to a file, the group owner applies a member-specific transform key to the file's encrypted symmetric key. The transformation changes the symmetric key's encryption from an encryption with the group public key to an encryption that corresponds to the group member's unique private key. The encrypted file together with the transformed encrypted symmetric key are sent to the member. Please note that the system does not use explicit authentication. The system relies on the fact that only the designated group member posses the member-specific private key, and therefore only she can access the file content. Others may pretend to be group members, but they do not posses a valid private key and thus can't access the file's content. Details of how the transformation is computed are given in section 3. Detailed descriptions of file operations like read, write, create and delete are given in section 2.5. The Paranoid file system uses XML group definition files created by the owner. Users define their own read and write access groups. Group definition files are digitally signed so that any tampering can be detected. Each access group has a public and private key pair that are used by the group owner to encrypt and decrypt symmetric keys. The group private key is kept a secret and is not shared with the group members. Additions and deletions from the group are done by the group owner using his Personal Security Proxy.

2.4. The Personal Security Proxy

The Paranoid file sharing system is secure provided keys are random and are kept secret. To enhance key management security, we developed software for an attached coprocessor that we call the Personal Security Proxy. The PSP is an integral part of the system. The PSP can be implemented on an attached portable device such as a Personal Digital Assistant (PDA) or a smart card. The PSP is used to protect secrets and perform sensitive encryption and decryption operations on behalf of its owner. The general purpose computer communicates with the PSP through a secure protocol. The owner's password protected private key is stored on the PSP along with private keys of groups owned by the user. Once the PSP is activated, it authenticates the owner and encrypts and decrypts keys on behalf of the owner. The PSP is responsible for communicating with the client and server modules residing on the owner's machine. The PSP is also used for generating symmetric keys and computing digital signatures. In the prototype implementation, the PSP operations were "simulated" on a general purpose machine. After the prototype development was completed we discovered that all the functionality of the PSP can be implemented using the attached processor of the Trusted Computing Platform (TCPA) [19]. Relatively small modifications of the current Paranoid implementation could make the code work with TCPA.

2.5. The Paranoid File System

File Access and Group Operations. The Paranoid system is a global file system supporting global user names and file names. The system uses email addresses as global user identities, and for global file names the system uses a URL-like naming convention. Each file name contains a global host name followed by the file name. A protocol similar to HTTP is use for communication. Paranoid files are encrypted using the toolkit developed. The toolkit takes a file, encrypts it and attaches an extended header to it. The header

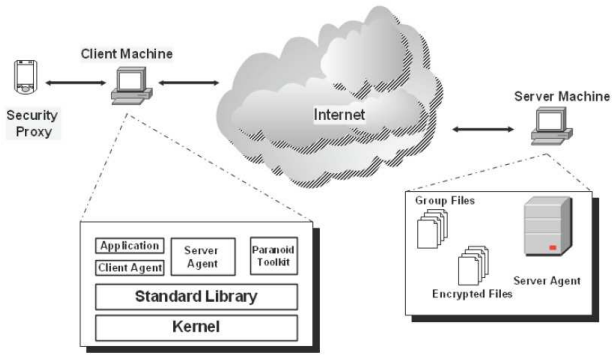

Figure 2. Paranoid system diagram

contains information about the encrypted keys, read access groups, write access groups and digital signatures.

Each access group has an XML group definition file listing members' identities, transform keys, and access privileges. The group definition file is encrypted with the group's public key. It is only accessible to the group owner. Adding or removing group members and changing the access rights for a file or directory are done by the group owner using the toolkit. Shared files are accessed via a file server agent. The file server enforces access rights, checks group members' access rights, applies key transformations and performs write operations on behalf of legitimate users.

File headers, encrypted files, and group definition files are digitally signed. While the system does not provide special protection against malicious file deletion, any tampering with Paranoid files is detected. Operations that modify files require users' PSPs to be active and able to communicate with the file server and client agent.

To globally share files, a file server agent must be running on the group owner's machine. This process authenticates access requests on behalf of the owner, performs key transformations, sends requested files to group members and writes files on behalf of group members. Modifying group access rights is done by adding, removing or modifying a member's entry in the group definition file. Symmetric keys of files that a revoked user has already accessed are lazily re-encrypted - that is, the operation is done at a later stage when the file is next written to. Note that only the group owner can perform these operations.

Adding a member to a group requires adding a new member entry, generating a random private key, computing a transform key, and delivering to the member his private key. The Paranoid system assumes that private key information is sent to members as an out of band operation. These operations can only be done by the group owner.

Paranoid File Read. Table 1 describes the chain of events triggered by an application running on a remote client machine opening a Paranoid file for a read operation. To keep the description simple it only covers a successful file open case. The table has three columns. The first column lists the machine taking the action. There are four computers involved: C, the client machine, C_PSP, the client Personal security proxy machine, S, the group file server, and S_PSP, the group owner's Personal Security Proxy. The second column in Table 1 lists the action taken. The third column provides an explanation and lists steps associated with each action.

Table 1. Sequence of events for reading a remote Paranoid file.

Machine	Action	Tasks and Explanations
C	open_file(r,<file>)	User application opens the file for reading
C	Intercept	Bypass intercepts the open_file system call and passes control to the client module.
C	Remote_Access_Req	The client agent opens a connection to file server, sends request, User_ID, File_name.
S	check_request	The file server checks the user's request: Does the file exist? What is the access group? Is the user in the access group? Does the user have read rights?
S_PSP	key_transform	The server's personal security proxy applies member specific key transformation
S	send_file	The file-server signs the file-header, sends file to client machine.
C	receive_file	The client agent receives the file, parses the file-header.
C_PSP	decode_key	The client's PSP deciphers the symmetric key.
C	decode_file	The client agent decodes the file, returns file pointer to Bypass.
C	return	Bypass returns control to application, passes file pointer to application.

When a client application opens a file for a read operation, the system call to open the file is trapped by the interposition agent created with Bypass. The request is checked to see if its operand is a Paranoid file. If so, the interposition agent invokes a client agent which sends out a read request to the appropriate file server along with the requester's credentials. The file server verifies that the files exist and that the group member has read access privileges. This verification is done using the group definition file. The file server identifies which group the requester belongs to and applies the requester-specific key transformation on the file's symmetric key. The encrypted file is sent back to the client agent along with a signed Paranoid header and the transformed symmetric key. File tampering is caught by verifying the digital signatures. The group member's PSP decrypts the transformed key and sends it to the client agent. The client agent decrypts the file and sends it to the interposition agent, which forwards it to the application.

Table 2. Sequence of events for writing a remote Paranoid file

Machine	Action	Tasks and Explanations
C	close_file	Client application closes a modified Paranoid file
C	intercept	Bypass intercepts the close file operation, passes control to client agent.
C	encrypt_file	The client agent generates a random symmetric key, encrypts the file, encrypts the symmetric key with the group public key, generates a header which includes a hash of the file.
C_PSP	encrypt_key	The client PSP signs the file hash and signs the header hash.
C	send_file	The client agent connects to the file server sending a write request along with the file name and user ID. It sends the file.
S	receive_file	The file server receives the file, parses the file header, checks member authorization.
S_PSP	signature_transform	The server PSP transforms the client signature into a group signature.
S	write_file	The server copies the file into the appropriate directory.

Writing Remote File. Remote file writes are done on whole files. When an application creates or modifies a paranoid file, the client agent performs read and writes on a local copy. Once the file is closed, the client agent does a remote write into the Paranoid file system. The client agent encrypts the file with a new random symmetric key. The symmetric key is encrypted with the member's own private key and the group's public key. The encrypted file is attached to the header and is digitally signed by the writer using his group private key. The file is sent back to the file server. The file server on the group owner's machine verifies the group member's write authorization. Once credentials and signatures are verified, the newer version of the file replaces the old one. The Paranoid file addresses multiple concurrent write back consistency by using a last writer wins policy. However, older versions of files are archived. Remote file and directory creations are the same as file write operations. In case a group member has remote file creation rights, the file server provides a default empty file to the requesting client agent. Newly created files inherit the same access groups as the directories they are created on. Alternatively, default groups can be specified in a configuration file.

Table 2 describes the chain of events that follows a remote client write request. Like Table 1, Table 2 has three columns. The first column specifies the machine, the second column specifies the action, and the third describes the action and its associated sequence of steps.

3. Key Transformation

The Paranoid file system uses a modified version of the *RSA* public key cipher [17]. Each access group uses a different modulus N, but all the members of a group use the same modulus. The modulus and the public exponent of the group key pair are published and the private exponent is only known to the group owner. Each group member is given a random exponent to use as his group private key. Associated with each group member is a *transform key*, known only to the group owner, that can transform a symmetric key encrypted with the group's public key into the symmetric key encrypted by the "public half" corresponding to the member's group private key. Thus each group member can encrypt a symmetric key for group use, but he can only decrypt a symmetric key after his specific transformation is applied to an encrypted key. The transformation step prevents a group member from granting group access rights to outsiders without revealing their group private key. Since this can be easily traced, it is expected to dissuade leakage. In contrast, systems that hand out the group private key to users effectively allow them to add new users by giving the key to others without any accountability. Further, Paranoid's scheme lets the owner remove a user from a group without having to re-encrypt any keys or files.

This section describes the transformation in detail. When a user creates a new group he creates a standard *RSA* modulus N where $N = pq$ where p and q are two large random prime numbers. The group has a public and private key pair, g and g^{-1}, where:

$$g \times g^{-1} \equiv 1 \pmod{\Phi}$$

where

$$\Phi = (p-1) \times (q-1)$$

Every group member M, $M = 1, 2, 3, ...$ is given a large random private key e_m that is relatively prime to Φ. The group owner also computes the inverse of this key e_m^{-1}, such that:

$$e_m \times e_m^{-1} \equiv 1 \pmod{\Phi}$$

Additionally, a transform key τ_m is computed using the following formula:

$$\tau_m = g^{-1} \times e_m^{-1} \pmod{\Phi}$$

The inverse private key e_m^{-1} is discarded and the transform key τ_m is stored in the group definition file together with the member identity M. The group definition file is encrypted and is kept on the owner machine where it is only accessible to the file server.

Assume that a symmetric key K is encrypted with the RSA cipher using the public key $< N, g >$, that is:

$$E(g, K) = K^g \pmod{N}$$

When user M asks to read a file encrypted with K, the file-server computes $E(\tau_m, E(g, K))$ and sends it to the group member M. The group member computes:

$$K = E(\tau_m, E(g, K))^{e_m} \pmod N$$

Proposition:

$$K = E(\tau_m, E(g, K))^{e_m} \pmod N$$

Proof: Since the set of integers relatively prime to Φ is a commutative group under multiplication modulo Φ:

$$\tau_m \times g \times e_m \pmod \Phi$$

$$\equiv g^{-1} \times e_m^{-1} \times g \times e_m \pmod \Phi \equiv 1 \pmod \Phi$$

Therefore, using *Euler's Totient Theorem*,

$$E(\tau_m, E(g, K))^{e_m} \pmod N$$

$$\equiv K^{g^{-1} \times e_m^{-1} \times g \times e_m} \pmod N = K$$

QED.

Please note: Applying the transform key τ_m to a signature H generated by M with his private key e_m transforms the signature H into a group signature generated with g^{-1}. That is:

$$E(e_m, H)^{\tau_m} \pmod N \equiv E(g^{-1}, H)$$

The proof is almost identical to the proof of the proposition above and is left to the reader. Also note that as long as the group owner keeps Φ, g^{-1}, and the transform keys secret, he can use the same modulus N for many different access groups.

3.1. Transformation Security

A primary question concerning the key transformation scheme is how secure is it? The answer is as follows. Since each group member's private key is a large random number, knowledge of the group public key and a group-member's private key does not give an attacker the ability to gain additional capabilities beyond impersonating the member. Any set of collaborating group members could not gain any additional capabilities they don't already have. For example, if group access privileges were taken away from a set of group members, they cannot regain group access by collaborating. Any group member getting hold of a symmetric key encrypted with the group public-key could not decrypt it without knowing the corresponding transform key.

However, the transform keys τ_m *must be kept secret*. Any person that knows both a group-member private-key e_m and the corresponding transform key τ_m can decrypt any symmetric key encrypted with the group's public key g. Thus she can access all the group files, bypassing any access controls. If she was also able to penetrate the server, then she could modify files, forge signatures and alter the group definition file, adding or subtracting members.

```
<FILE name=... path=... hostname=...>
        <OWNER name=.../>
        <SYMMETRIC_KEY key=... iv=.../>

        <READ_GROUPS>
                <GROUP name=...>
                <GROUP_SYMMETRIC_KEY key=... iv=.../>
                </GROUP>

                <GROUP name=...>
                <GROUP_SYMMETRIC_KEY key=... iv=.../>
                </GROUP>
        </READ_GROUPS>

        <WRITE_GROUPS>
                <GROUP name=...>
                <GROUP_SYMMETRIC_KEY key=... iv=.../>
                </GROUP>
        </WRITE_GROUPS>

        <TIME_STAMP value=.../>
        <SIGNATURE lastwriter=... value=.../>
</FILE>
```

Figure 3. XML headers for Paranoid Files

4. Implementation details and results

4.1. XML

Paranoid files are encrypted and stored in XML format. An XML header is prepended to the encrypted data. Binary data, such as encrypted keys, is stored in hexadecimal format for readability. A simplified schema of the XML file header is given below. The header contains the file access information along with the protected decryption keys. The header also contains a list of groups having read or write access rights. An encrypted symmetric key is stored with each group name. The file contains a digital signature of the XML header and the encrypted file.

4.2. Group Files

The XML group definition file is a list of group members and members' transform keys. The skeleton of one is shown in Figure 4.

4.3. Client Modules

Applications communicate with the operating system through the standard library that makes system calls to contact the kernel. Since Paranoid files are kept encrypted, applications can't use them without decryption. Making changes to the operating system or the input and output library on requires super-user privileges on most systems. Paranoid uses interposition agents to make sure existing applications run seamlessly without having any knowledge of the underlying encrypted file format.

```
<GROUP name=...>
<OWNER name=.../>
        <MEMBERS>
                <MEMBER name=...>
                <TRANSFORM_KEY value=.../>
                </MEMBER>

                <MEMBER name=...>
                <TRANSFORM_KEY value=.../>
                </MEMBER>
        </MEMBERS>

        <TIME_STAMP value=.../>
        <SIGNATURE lastwriter=... value=.../>
</GROUP>
```

Figure 4. Paranoid Group Files

We use the Bypass system [20] to implement the Paranoid global encrypted file system. The Bypass system allows us to modify the behavior of a selected set of system calls by replacing each selected system call with code that we supply. The Bypass system traps system calls and executes the supplied code in user space. Thus one can modify or enhance the operating system without having to modify the kernel or system libraries themselves.

A modified version of the **C** input/output (I/O) library, implemented in the client agent, insulates applications from Paranoid file system's underlying details. The modified I/O library supports transparent access to remote cryptographic files. Relevant I/O system calls, such as *open* and *close*, are intercepted. Then the client agent executes code to fetch remote files, communicate with the PSP, and perform cryptographic operations on the files. For example, when the *open* system call is issued by an application, the remote Paranoid file is fetched and its content is decrypted. Control is then returned to the running application once the file's integrity is verified. A local decrypted copy of the file is created its descriptor is passed to the application in lieu of encrypted file's descriptor.

4.4. Global Files

Paranoid makes it as easy to use a remote encrypted file as it is to use a local unencrypted file. Paranoid file names extend the usual UNIX file naming convention. Like in UNIX, a file name is a string with substring fields separated by **/**. The first substring is the keyword **/paranoid** and the second is **[userID@]host[:port]** where **userID** is an optional user identity specifying the owner of the file, **host** is the hostname or IP address and **port** is an optional port number where the Paranoid server agent is running. The default user identity and port number are specified in a configuration file **.paranoidrc** in the user's home directory. For example, to edit a Paranoid file with **vi**, a user could type (in an interactive shell): **vi /paranoid/abc.cs.xyz.edu/file.txt**.

The client agent opens a TCP socket connection to a file server process running on the group owner's machine which in this case is $abc.cs.xyz.edu$. Since no user identity or port were specified, the defaults are read from the configuration file. A simple handshake protocol is used to send the remote user credentials, request the file, and transfer the file to the client machine. Note that requests and files are sent in the clear since files' contents are already encrypted, users are implicitly authenticated, and responses are signed.

File Size	1Mb	4Mb	8Mb	16Mb	64Mb
Verification	0.41	1.2	2.09	3.97	16.09
Decryption	0.65	2.47	4.95	10.08	41.42
Encryption	2.56	4.32	6.69	11.41	43.33
Transfer	0.26	0.8	1.51	2.97	12.93

Table 3. Latency observed (in seconds) to *open* and then *close* a Paranoid file on a local host.

File Size	$1Mb$	$4Mb$	$8Mb$	$16Mb$	$64Mb$
Transfer (S→C)	1.64	5.59	11.34	21.95	89.86
Verification	0.24	0.94	1.77	3.55	15.34
Decryption	0.6	2.38	4.78	9.7	42.25
Encryption	2.56	4.33	6.74	11.53	44.97
Transfer (C→S)	0.4	1.54	3.09	6.28	25.42

Table 4. Latency observed (in seconds) to *open* and then *close* a Paranoid file on a remote host.

5. Performance

The overhead of trapping a system call using Bypass is between 9 to 28 μs [20]. Paranoid incurs a large cost for encrypting, decrypting, signing and verifying files. The use of cryptographic operations in the critical path of file operations has the potential to create a significant adverse impact on overall performance. However, we argue that this overhead is acceptable in the context in which Paranoid is to be used since the file operations are dominated by the latency introduced by the network transfers. Below we provide measurements of the time it takes to open and close Paranoid files to illustrate the effect it has on performance.

The measurements were made using two 300 MHz Intel Pentium II machines connected through the network. The benchmark program invokes the *open* and *close* system calls a number of times over a range of files of different sizes. The tables shows the mean results over 10 runs each for 5 file sizes between 1MB and 64MB. Table 5 shows a client and server located on the same machine with the client opening and closing a Paranoid file. Table 5 displays times for a client and server located on different machines.

The first transfer time is the measurement when the file is retrieved from the server by the client. This includes the time to effect the transform key on the server. Thus this time is much greater than the second transfer time which occurs when the file is returned to the server after it has been closed.

6. Related Work

Several previous projects have proposed the use of encryption to lock data stored in files. The Cryptographic File System (CFS) [1, 2], created at AT&T Bell Laboratories, was one of the early realizations of such a scheme. However, CFS was designed as a local file system. Therefore, the only way a file could be shared was by explicitly distributing file keys to other users. CFS used symmetric keys for all protection. This meant that the keys were left unprotected in memory while in use. Such a scheme is vulnerable when an attacker gains access to the system since they then have access to the keys as well. The use of a public key scheme like that of Paranoid reduces this exposure. Further, the granularity for file accesses in CFS is per directory. Paranoid can be used to provide per file read or write access and per directory create permissions.

The Transparent Cryptographic File System [3] is similar to CFS but it moves the functionality from user space to kernel space for performance and ease of use. Cryptfs [21] uses a stackable file system infrastructure to provide similar functionality. TCFS, Cryptfs and [8] have the same weakness as CFS, which is that the symmetric keys are unprotected. This can only be resolved through the use of a public key cipher in the protocol.

Network of Attached Secure Disks [7] and Secure Network Attached Disks (SNAD) [4] store data remotely and operate at block level. Data is unprotected on the server in the former with data servers cooperating with a single group server for access control, making it a central point of failure. SNAD uses certificates for authentication. However, since access is at the block, neither system can provide the end-to-end security semantics that Paranoid can.

The Self-certifying File System (SFS) [11], from MIT, addressed the problem of mutually authenticating servers users. This was done in order to prevent an adversary from spoofing the server. SFS achieves this through path names which embed the public key. SFS-Read Only [6] extended SFS to address the problem of securely sharing read only data across the Internet.

Cepheus [5] focuses on the separation of storage and group server functionality. It uses session keys to protect communication between the server and clients. The storage server does not need to be aware of the access control operations which are handled by the group server. A shortcoming of the system is the fact that group members are given the private key of the group. Paranoid's transform keys prevent a group member from granting his group access privileges to an outsider without revealing his own private key. In contrast, possession of Cepheus' group private key effectively allows a user to add new users to the group without revealing who effected the delegation. By restricting access to the group key to only the group owner, such direct leakage of rights is not possible with Paranoid. The transform keys of Paranoid force a user to divulge their personal private key allowing the source of such leakage of rights to be uniquely identified.

Plutus [10] uses a client based key distribution scheme. It focuses on using file groups to reduce the number of keys exchanged between users. Plutus, from HP Labs, provides group sharing by explicitly sharing the secret with all the group users. This suffers from the same problems as Cepheus described above. [16] compares several related cryptographic filesystems.

The Encrypting File System of Windows 2000 [12] uses symmetric keys to encrypt files. These are then encrypted with a public key cipher for rights management. Since they are stored on the host, rather than with a PSP, they are exposed in the event of a system compromise. Further, Paranoid's transform keys extend the scheme to enable cryptographic group access control.

The Secure File system, developed at the University of Minnesota [9] uses a protocol similar to Paranoid. However, a key difference is that access control is arbitrated by a group server rather than the end user. This does not have the end-to-end security semantics guarantees of Paranoid. In the event that a security compromise is detected in Paranoid, only the currently active files are at risk. In the Secure File System scheme, there is no way to prevent the attacker from accessing all the remaining files that the group server is responsible for but are not currently being used, if the system is compromised.

The Trusted Computing Platform Alliance [19] is an alliance of industry leaders in hardware and software. It aims to build a trusted computing environment on top of trusted hardware. The IBM 4758 Cryptographic Co-processor [18] is a high security, programmable PCI board which can be used to provide data and cryptographic processing to implement TCPA functionality. It contains tamper detection sensors, circuitry of cryptographic operations, a microprocessor, memory, and a random number generator. It aims to provide security even in the face of a physical attack on the device. Its high cost and weak processing power has hampered widespread adoption. Palladium [13] provides lower assurance security than such a trusted co-processor but is cheap enough to be commercially feasible for commodity desktop systems. Paranoid performs privileged tasks on the PSP. Data is decrypted into the client's volatile memory and assumed to be secure if stored there temporarily. The PSP's

functionality could instead be implemented using the IBM 4758 or Palladium.

7. Conclusions

This paper presents the Paranoid file system, an encrypted global file system that implements peer-to-peer transparent file sharing with UNIX-like access controls. Each user can define access groups and grant group access rights to peers outside their protection domains without the need for any intervention by an administrator. A novel public key transformation scheme is used to facilitate low cost revocation of access rights. Performance measurements show that the implementation overhead for the encrypted global file system is low enough to make it practical. While the implementation described is Linux specific, the Paranoid system could easily be ported to other operating systems by using dynamically loaded libraries that redefine file I/O. All applications that use dynamic linking would then be able to use the Paranoid file system. Implementing Paranoid using the Trusted Computing Platform (for secure management of cryptographic keys) can further reduce system vulnerability.

References

[1] M. Blaze. A cryptographic file system for unix. *Procedings of First ACM Conference on Computer and Communications Security*, 1993.

[2] M. Blaze. Key management in an encrypting file system. *Proceedings of the USENIX Summer Technical Conference*, 1994.

[3] G. Cattaneo, L. Catuogno, A. D. Sorbo, and P. Persiano. The design and implementation of a transparent cryptographic file system for unix. In *Proceedings of the FREENIX Track: 2001 USENIX Annual Technical Conference*, 2001.

[4] W. Freeman and E. Miller. Design for a decentralized security system for network-attached storage. *Proceedings of the 17th IEEE Symposium on Mass Storage Systems and Technologies*, 2001.

[5] K. Fu. Group sharing and random access in cryptographic storage file systems. Master's thesis, MIT, 1999.

[6] K. Fu, F. Kaashoek, and D. Mazieres. Fast and secure distributed read-only file system. *Proceedings of the 4th USENIX Symposium on Operating Systems Design and Implementation*, 2000.

[7] G. A. Gibson, D. F. Nagle, W. C. II, N. Lanza, P. Mazaitis, M. Unangst, and J. Zelenka. Nasd scalable storage systems. *Proceedings of USENIX Extreme Linux Workshop*, 1999.

[8] P. Gutmann. Secure file system, 1996.

[9] J. Hughes and C. Feist. Architecture of the secure file system. *Proceedings of the Eighteenth IEEE Symposium on Mass Storage Systems*, 2001.

[10] M. Kallahalla, E. Riedel, R. Swaminathan, Q. Wang, and K. Fu. Plutus: Scalable secure file sharing on untrusted storage, 2003.

[11] D. Mazieres. *A Self Certifying File System*. PhD thesis, MIT, 2000.

[12] Microsoft Corporation. *Encrypting File system in Windows 2000*, 1999.

[13] Microsoft Windows Trusted Platform Technologies. *Microsoft Palladium: A Business Overview*, 2002.

[14] National Bureau of Standards, U.S. Department of Commerce. *Data Encryption Standard*, number 46-1 in FIPS, 1988.

[15] National Institute of Standards and Technology. *AES Proposal: Rijndael*, 2000.

[16] E. Riedel, M. Kallahalla, and R. Swaminathan. A framework for evaluating storage system security. *Proceedings of the 1st Annual Conference on File and Storage Technologies*, 2002.

[17] R. Rivest, A. Shamir, and L. Adleman. A method for obtaining digital signature and public-key cryptosystems. *Communications of ACM*, 21, 1978.

[18] S. W. Smith and S. Weingart. Building a high-performance, programmable secure coprocessor. *Computer Networks*, 31(9), 1999.

[19] http://www.trustedcomputing.org/.

[20] D. Thain and M. Livny. Multiple bypass: Interposition agents for distributed computing. *Journal of Cluster Computing*, 2001.

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