Cryptographic Access Control in a Distributed File System

Anthony Harrington Christian D. Jensen
Department of Computer Science
Trinity College Dublin
{Anthony.Harrington,Christian.Jensen}@cs.tcd.ie

ABSTRACT

Traditional access control mechanisms rely on a reference monitor to mediate access to protected resources. Reference monitors are inherently centralized and existing attempts to distribute the functionality of the reference monitor suffer from problems of scalability.

Cryptographic access control is a new distributed access control paradigm designed for a global federation of information systems. It defines an implicit access control mechanism, which relies exclusively on cryptography to provide confidentiality and integrity of data managed by the system. It is particularly designed to operate in untrusted environments where the lack of global knowledge and control are defining characteristics.

The proposed mechanism has been implemented in a distributed file system, which is presented in this paper along with a preliminary evaluation of the proposed mechanism.

1. INTRODUCTION

The widespread availability of networks such as the Internet as a medium for communication has prompted a proliferation of both stationary and mobile devices capable of sharing and accessing data across networks spanning multiple administrative domains. Such collaboration increases user dependence on remote elements of the system which are often outside their control and heightens their need for effective security systems. The shift toward the networking paradigm has resulted in a fundamental re-evaluation of computer security and its place in system design.

Current access control mechanisms invoke a *reference monitor* [1] to verify a principal's right to access a protected object every time the object is referenced. The access rights of all principals in the system can be viewed as an access control matrix [14], which is stored either in the form of an access control list (ACL) for each object in the system or in the form of a capability list for every principal in the system. ACLs require the authenticated identity of the requesting principal to be known before access can be granted, which reduces the ability to define dynamic access control policies. Moreover, ACLs do not scale well, because it is difficult to delegate the right to delegate, i.e. modify the ACL. This means that the ACLs have to be organised into a hierarchy where the right to delegate access rights corresponds to the right to modify the ACL in a specific branch of the hierarchy, Thus, the ACL hierarchy must reflect the structure of the entire organisation.[1]

On the other hand, capabilities provide the necessary flexibility to grant and delegate access rights in large scale open networks, but they only control access to the object on the managing server. The capability does not control access to the object while it is transferred across a potentially hostile network or while it is temporarily cached on an intermediate server.

Reference monitors and access control matrices are fundamental concepts in access control; originally developed in the context of centralized systems. Different extensions have been introduced to extend their use to distributed systems. Strong authentication mechanisms, such as Kerberos [12], X.509 [19] and BAN logic [15] (an extension of BAN logic is used in the Taos operating system [13, 34]), allow ACLs to be used in a networked environment and capabilities can be partly encrypted to ensure the integrity and authenticity of the capability in a distributed system, e.g., this method is used in Amoeba [32]. However, the reference monitor is inherently centralized, as the interpretation of access control state, i.e., the encoded access control matrix, is limited to the server that manages the particular object.

A distributed reference monitor would need a consistent view of access control state, which is theoretically impossible, if failures may occur in the system [8]. Moreover maintaining consistency limits the scalability of the access control mechanism. In addition, a distributed reference monitor enforces a homogeneous security policy on a heterogeneous environment and requires global knowledge on the part of the entity that defines security policy. Finally, the reference monitor only controls access to the protected object on the server that stores the object, additional mechanisms are needed to ensure confidentiality and integrity of data while they transferred across the network or temporarily cached on an intermediate server.

We therefore need to rethink the notion of access control in large-scale open systems, such as the Internet, in a way that separates enforcement of access control policies from the storage of objects in the system and, at the same time, ensures the confidentiality and integrity of data stored and transferred in the system.

In this paper we propose *Cryptographic Access Control*, a novel

[1]This is successfully demonstrated in section 4.3 of RFC 2693 [7].

access control mechanism that eliminates the need for a reference monitor and relies exclusively on cryptography to ensure confidentiality and integrity of data stored in the system. Data are encrypted as the applications store them on a server, which means that the storage system only manages encrypted data. Read access to the physical storage device is granted to all principals (only those who know the key are able to decrypt the data) and write access can be granted to everybody, providing that the modifications do not overwrite existing data, e.g. by using a journaling mechanism that tracks modifications of objects instead of applying them directly to the stored version of objects.

In order to separate read and write access, the cryptographic storage system must implement an asymmetric encryption algorithm. Data are encrypted with the private key, also called the *encryption key* and decrypted with the public key, also called the *decryption key*. Data that are not encrypted with the encryption key cannot be decrypted with the decryption key, so an earlier version must be read from the log. This allow us to selectively grant read-access and write-access to the data, by respectively giving the decryption key, the encryption key or both to another principal.

The advantage of this approach is that it is inherently distributed, i.e., the tokens needed to access data only have meaning where data is used, so there is no need for coordination of access control state among clients and servers in the system. Moreover, no additional mechanism is needed to protect data in transit between users and the servers that stores data, since cryptographically stored data can be transferred across the network "in the clear" because the contents are already encrypted.

By employing security protocols at the application level, the cryptographic access control model respects the End-To-End argument [23]. The targeted operational domain for this access control method, i.e. large-scale open systems characterised by a lack of central authority, is such that the cryptography overhead which must be borne by all applications is justified.

We have implemented cryptographic access control in a file system that allows users to securely access and store files on a server from any machine in the network. This file server implements no access control mechanism, so users can access their files from any user account and any machine connected to the network. The design and implementation of this prototype is outlined in this paper along with preliminary performance measurements.

The rest of this paper is organized as follows. we define Cryptographic Access Control in Section 2. The design and implementation of our prototype file system is presented in Section 3. We present the evaluation of our prototype in Section 4. Related work is discussed in Section 5 and our conclusions are presented in Section 6.

2. CRYPTOGRAPHIC ACCESS CONTROL

The cryptographic access control mechanism is designed to operate in an open network where establishing the identity of the client conveys no information about the likely behaviour of the client and thus, is irrelevant to the secure operation of the server. It is flexible enough to operate across multiple administrative domains with varying security requirements and offers a high degree of resource protection in these untrusted environments. It makes no assumptions about the security of remote elements in the system such as the transport mechanism or file server and only requires clients to place complete trust in their own machine. The cryptographic access control mechanism seeks to reposition the burden of access control from the server to the client.

2.1 Basic Mechanism

The basic cryptographic access control mechanism is based on asymmetric cryptography, where objects are encrypted with the private key, also called the *encryption key*, and can only be decrypted by someone who knows the public key, also called the *decryption key*. A separate key-pair may be associated with every file and should only be distributed to principals who are allowed to access the file's data. Restricting the distribution of keys allows us to export the actual file to everybody. The two keys effectively act as capabilities, where the decryption key corresponds to a read-capability and the encryption key corresponds to a write-capability. However, capabilities are part of an access control mechanism and the semantics can only be interpreted by the object server. The semantics of the encryption/decryption key-pair is universally valid.

Asymmetric cryptography ensures secrecy and authenticity of the stored data, but neither integrity nor availability are guaranteed. Everybody can write the file, so integrity is easily compromised by overwriting the file with garbage. This also denies legitimate users access to the stored data, thus compromising availability.

Overwriting data without knowing the proper encryption key is easily detected because the result of decryption is unlikely to respect the format of the file.[2] However, detection is deferred to runtime and the process may already have modified other data that are now in an inconsistent state. Detection of corrupted data may happen until the last read from disk is completed, so we either have to execute all programs within transactions that can be restarted if corrupt data are detected, or we need to prevent users from overwriting data arbitrarily. Executing long running programs as transactions isn't practical, so we decide to prevent users arbitrary write access to data.

Integrity and availability can be ensured by using a log-structured file system [22, 30, 25, 6], where modifications are written to a log instead of overwriting disk blocks. An attacker may write false log entries, but this is easily detected and corrupt disk blocks can be ignored, i.e. by reading the latest entry written to the log that decrypts correctly.

2.2 The Problem of Log Compression

In order to prevent the log from growing indefinitely and limit fragmentation of user data on the file server, most log-structured file systems implement a mechanism to compress logs and reclaim space allocated to entries that have become obsolete. This means that the server needs some way to distinguish valid log-entries from invalid log-entries created by an attacker. A simple approach is to provide the server with the decryption key, which allows it to identify incorrect log-entries and remove them from the log. However, the decryption key alone does not allow the server to detect invalid log-entries in unstructured binary data, so we require that the client signs log-entries, with the encryption, key before they are sent to the server.

Providing the server with the decryption key may compromise con-

[2]Detection of corruption is difficult in unstructured binary data, such as image bitmaps, but the modifications proposed in 2.2 and 2.3 solve this problem.

fidentiality on compromised servers or servers that are simply not trustworthy. A solution to this problem is presented in Section 2.3.

2.3 A Necessary Optimisation

So far, we have assumed that asymmetric cryptography is sufficiently fast to encrypt/decrypt all data to be stored in the system. In reality, asymmetric cryptography is too slow – several orders of magnitude slower than symmetric cryptography. We therefore propose an extension that uses symmetric cryptography to encrypt and decrypt data, while asymmetric cryptography is reserved for generation and verification of digital signatures.

Data stored on the server are encrypted with a symmetric algorithm and signed with the encryption key as before. The digital signature is used by clients to guarantee integrity of the data and by the server to provide a means for log compression as described in Section 2.2. The addition of symmetric cryptography to the system means that we now have two cryptographic algorithms and three cryptographic keys that are used in the following way:

read clients that are authorized to read the file need the symmetric key to decrypt data received from the server and the decryption key to verify the signature that proves the integrity of the data.

write clients that are authorized to write the file need the symmetric key to encrypt the data before sending them to the server and the encryption key to generate the signature that proves integrity of the data.

log compression the server only needs the decryption key to verify the signature of log-entries in order to weed out corrupt entries and merge correct log-entries with the original file data. It is important to note that the addition of symmetric cryptography solves the confidentiality problem introduced by giving the decryption key to the server.

The modification introduced above means that when a user wishes to store a new file on the server he must first encrypt the data using symmetric key cryptography. Next he must produce a hash or message digest of the encrypted file contents. The message digest is then signed using the encryption key known only to clients with write access to the file. This message digest and the decryption key used to verify the signature are then sent to the server along with the data to be stored. Upon receipt of the data and digest the server will generate its own digest from the encrypted data using the same algorithm. It then decrypts the client-generated digest using the file's decryption key and compares the two digests. If they match, the write request is considered valid. The server stores the decryption key associated with the file and uses this stored key for validating future write requests from the user. If the data originates from a malicious client then the message digest cannot be signed using the valid encryption key and the write operation is considered invalid. When a user wishes to update the contents of a file which already exists the server, he encrypts the data and signs it as before. When the server receives the data from the user, it can retrieve the decryption key associated with the file and validate the data immediately. This means that we no longer need a log-structured file system to ensure integrity, but we may still decide to use one in order to improve performance.

When the client issues a read request the data returned is encrypted with the symmetric key associated with that file. This key is only stored on the machines of clients that are authorized to read the file. The client will possess this key and can thus decrypt the data however it must also ensure that the data has not been tampered with on the server side or in transit. This is accomplished through the use of message digests. When servicing the clients read request the server also returns the hash of the file which was signed with the encryption key associated with this file. The client then creates his own hash of the data, decrypts the hash supplied by the server and compares the two digests. If they match, the authenticity and integrity of the data is established.

2.4 Discussion

It is a fundamental requirement of our system that the user have complete trust in the client machine. All data stored locally are unencrypted and if a malicious agent succeeds in compromising the client machine then data confidentiality is destroyed. The keys necessary to read and update the users files are also stored on the local machine. If an attacker obtains these keys then the system has been fully compromised.

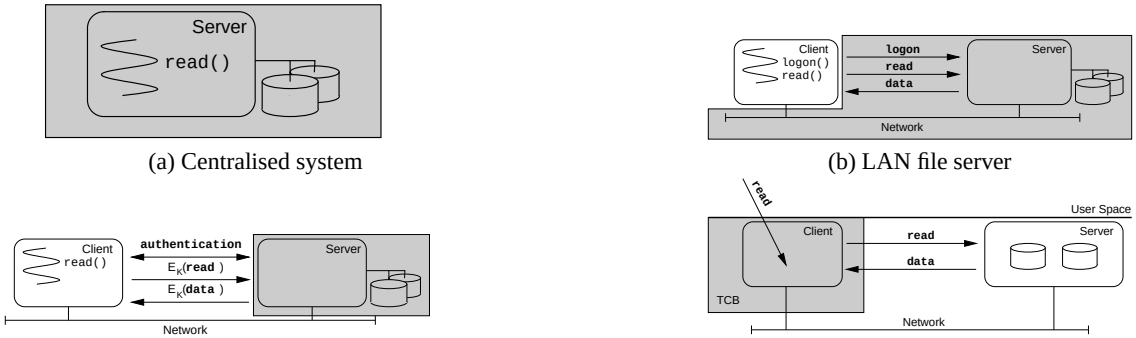
The system is designed to operate over a public unsecured network. All data are encrypted by the client before the remote storage procedures are invoked. This ensures that all data is encrypted prior to being transmitted across the network. All data sent from the server to the client is also encrypted. The use of digital signatures protects against deliberate or accidental changes to the payload of the network packets. This effectively means that data integrity and confidentiality are guaranteed and the user need not place any trust in the network.

In the envisaged operational environment for cryptographic access control the server may be located in an administrative domain outside the control of the client. With this in mind the extent of the trust that the user must place in the server is a critical factor in determining the strength of the system. We have three principal requirements to meet before the storage of data on the server can be considered secure. We must ensure the confidentiality, integrity and availability of the data.

Confidentiality Data is stored in encrypted format and the server does not possess the keys necessary to decrypt the data. An assailant who compromises the server but who does not possess the necessary keys cannot access the data in a meaningful manner. All data stored on the server can thus be considered confidential.

Integrity When a client sends data to the server it also sends the file's public key and a message digest of the data signed with the file's private key. This digital signature is stored with the file data on the server. Future read requests from the client result in the file data and digital signature being sent to the client. If the data has been altered on the server the signed digest will be invalid and the user will be aware that the integrity of the data has been compromised.

Availability The server performs three tasks: it stores data on disk, it provides data to any client requesting a particular object and it updates data modified by authorized clients. There is no direct way for the client to verify that the server actually performs these tasks. A server may receive an update from an authorized client, but decide not to commit it to disk. It may also segregate two collaborating principals by keeping two copies of a particular file and serve each client from a



(a) Centralised system

(b) LAN file server

(c) Distributed file systems

(d) Cryptographic Access control

Figure 1: Trusted Computing Base in different types of file systems

separate copy of the file. These problems are common to all distributed file systems, which is why clients are required to interact with a pre-specified set of trusted servers. However, we believe that these problems can be solved, either through a system of server issued receipts [18], or though a system of client maintained version numbers. Otherwise, clients simply have to select trustworthy servers in the same way that they would with a traditional distributed filesystem, except that they only need to require availability from the server, confidentiality and integrity are guaranteed by the cryptographic access control mechanism, i.e., clients are more free to select servers.

Contrary to traditional access control mechanisms, cryptographic access control implements a client centric view of the trusted computing base (TCB). The trusted computing base of a number of popular filesystem architectures is illustrated in gray in Figure 1.

Figure 1 (a) shows a traditional centralized file system, where all file system components belong to the trusted computing base of the local operating system. Figure 1 (b) shows a traditional file server, e.g., NFS [24, 31], where both the file server and the intermediate network are considered to be secure, but the client must be authenticated, e.g., through a logon procedure. Figure 1 (c) shows an example of a global file system, where the network is considered insecure, while the server is considered secure and generally stores data unencrypted on disk. Consequently, the clients and servers have to authenticate each other. Moreover, data has to be encrypted on the server before it is sent to the client, which decrypts the data, performs the required operations on the data before it re-encrypts them and sends them back to the server. The server then has to decrypt the data before they can be written to disk. Finally, Figure 1 (d) shows the trusted computing base in a system with cryptographic access control, where only the client is trusted. Data can be sent from the server without prior authentication and the client must decrypt the data before it can perform any operations on them. The client then re-encrypts the data before sending them back to the server, which stores the encrypted data back on disk.

We believe that cryptographic access control will improve the performance of file system operations across untrusted networks, because we only need one decryption and signature validation for a read operation and one encryption and signature creation for a write operation as opposed to one authentication (which may include signature validation) one encryption and one decryption for every read operation and one authentication, one encryption and one decryption for every write operation.

3. SYSTEM DESIGN AND IMPLEMENTATION

In the following we present the design and implementation of CNFS [10], a network file system that implements cryptographic access control. A key design goal for CNFS is to minimize the number of trusted components in the system. The principal system components are the client module, the key distribution mechanism, the network connecting the file server and client and the file server itself. Use of the cryptographic access control mechanism requires that the user need only completely trust the client machine and the key distribution mechanism, while only minimal trust is required in the file server.

3.1 System Architecture

The architecture of the CNFS System as illustrated in Figure 2 is based on the traditional client/server model. The server and client have been implemented in user level space. The overhead introduced by the additional context switches was offset by the flexibility offered by a user level solution which does not require root privileges or kernel recompilation to install. The ease of debugging user level applications coupled with the widespread availability of cryptography libraries enabled more rapid development of the prototype.

Client Architecture

All data to be stored on the server is encrypted on the client side prior to transmission across the network and all data to be read from the server is only decrypted on the client side. The full responsibility for ensuring the authenticity and integrity of file data rests with the client. Thus the client must also produce and validate hashes of all data stored on the server.

Server Architecture

The CNFS server is designed to be as simple as possible and the system intelligence should in the main reside on the client side. Ideally the file server would simply serve disk blocks on request from the client. Although the CNFS server codebase is taken from the Linux NFS server there are significant differences in design. The NFS failure model assumes a stateless service and all file operations are designed to be idempotent. These file operations are carried out on data units which match the underlying NFS block size and the NFS server has no concept of the file as a logical unit in itself.

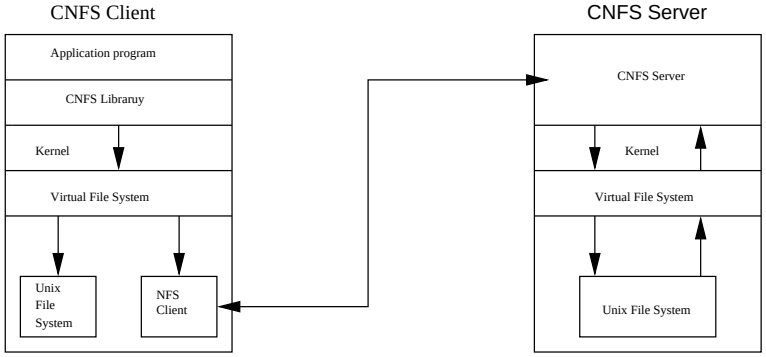


Figure 2: CNFS architecture

However the write validation routines implemented by the cryptographic access control mechanism require the CNFS server to perform digital signature verification on the entire file before the data can be written to disk. Fundamental changes to the NFS write procedure were required to accommodate this functionality.

When dealing with updates to a file which is a multiple of the block size the CNFS server must maintain state information in relation to which clients were updating open files. It is also necessary to buffer data written from the client until the entire file is transferred across the network. The file name and client identity are established from information included in each RPC request header. This information is maintained in memory while a client has a server resident file opened in write mode on his machine.

When the client writes the first block to the CNFS server, this block is validated by the cryptographic access control mechanism. If this operation is successful, state information in relation to file and client identity is maintained. Each subsequent data block can then be buffered until all data is received from the client. The buffer is passed to the validation routines and ultimately written to disk or flushed from memory.

This also has the effect of delaying the write to disk operation and has important consequences for the NFS failure model. Buffering data increases the likelihood of cache inconsistency faults. The window of vulnerability associated with lost updates due to server crashes is also widened.

Use of Cryptography in CNFS

Employing cryptography at the level of the file system itself enables applications and users to access protected data in a transparent fashion. Encryption in the CNFS System is employed at the granularity of the file level. This necessitates a high overhead for file generation operations on the client as a symmetric key and an asymmetric key-pair must be generated for each new file created. However the advantages offered by this approach, namely increased flexibility and security outweigh the disadvantages.

The symmetric and encryption keys for files are never stored on the file server. This is a fundamental requirement in ensuring that compromising the file server will not compromise the data stored there. These keys are to be stored on the client which owns or generated the files originally. Only the decryption key associated with a file is stored on the server.

In the case of CNFS sharing files is inextricably intertwined with key distribution. As files are meaningless without the appropriate keys the file owner must make the key available to whomever (s)he wishes to share his files with. Studies of file system usage properties have noted that file sharing is unusual[26]. It seems logical that sharing protected files among large numbers of users would be rarer still. As the symmetric key and private asymmetric key are created and stored on the client machine, key distribution is at the file owners discretion.

3.2 CNFS Implementation

In this section we present the implementation of cryptographic access control in the CNFS system. The cryptographic access control mechanism was originally designed to operate on a log-structured file system. However, the use of symmetric cryptography as a necessary optimization allows us sufficient flexibility to use the standard Linux NFS codebase.

CNFS Ciphers As stated earlier only the decryption key is stored on the server. The absolute file path is hashed and normalized to produce a reference value for the relevant key. The decryption keys associated with all files on the server are maintained in a binary tree which is sorted by the hash value. When the server wishes to find a file's public key it reconstructs the file path from the filehandle and searches for a match. A lookup operation on this tree is an $O(\log n)$ operation. As each new key is registered with the server an extra node is added to the tree, this information is also written to a file stored on the server so that they keys can be restored if the server fails. Each time the server is started it must load all public keys from this file into memory.

The Twofish cipher was chosen as the symmetric cipher [29]. To date no significant success have been reported in crypt-analysing this cipher. We obtained the reference implementation which was submitted to the AES panel. The key size used in our prototype is 128 bits. This is deemed sufficient for security in the envisaged operational domain.

In the CNFS system asymmetric cryptography is only used to sign and verify the message digests generated from the contents of the encrypted files.

The RSAEuro library we used provides a mature and extensive API with support for RSA encryption, decryption, key generation, and

the generation and verification of Message Digests using MD2, MD4 and MD5. We use the MD5 hashing algorithm to generate a 128 bit digests of the file data which are signed with the private asymmetric key. The asymmetric key length used in the prototype implementation is 1024 bits.

4. EVALUATION

In order to evaluate the prototype we have performed an initial performance evaluation and an analysis of the security properties of the proposed mechanism.

4.1 CNFS Server Performance

We have compared the performance of the CNFS server with the performance of the unmodified user level NFS server. The lack of cryptography in the standard NFS server mitigates against direct comparison with the CNFS server. This evaluation is primarily intended to determine whether the overhead introduced by cryptographic access control is acceptable.

The execution time of the *read()* and *write()* procedures on the server have been measured. TCP stack and XDR operation times are not included. Two file sizes were chosen for the tests. The first size is 8K, the underlying NFS V2 and CNFS data block size. The second file size of 1MB was chosen as an arbitrary large file size.

The tests were carried out on a lightly loaded Pentium Pro 233 MHz machine with 128 MB SDRAM. The user level CNFS server and client were run on the same machine. A local directory was mounted and requests for file operations in this directory were handled by the CNFS server. For the purposes of our tests caching was disabled on the client and server side. All experiments were run 20 times and the arithmetic mean and standard deviation of the results of these experiments are presented in the tables below.

Read Performance

The absence of centralized access control allows the CNFS server to process read requests without validation. There is an obvious performance benefit and the CNFS read operation on the 8 K file is 20% faster than the read operation on the unmodified NFS server.

Access control and request caching functionality were closely integrated in the NFS codebase. The modifications required to the *read()* operation on the CNFS server involved removing the ACL lookup operation. Some weakening of the caching strategy was unavoidable. Compromising the caching strategy results in a read operation on the 1MB file which is 10% slower on the CNFS server. The results of these operations are displayed in Table 1.

Write performance

The write operation for a file of size 8 K on the CNFS server is two orders of magnitude slower than the comparable operation on the unmodified server. This difference is accounted for by the overhead introduced by the digital signature validation. A similar overhead would be introduced by any mechanism that ensures the integrity of data transferred over a network. The buffering mechanism used in CNFS for larger files accounts for the relatively small difference in performance between the two servers for larger file sizes.

4.2 Security Analysis

The design of the CNFS system makes it very difficult for a malicious agent to mount a man in the middle attack. Minimal trust is placed in the server and any interceptor of messages can only inflict as much damage as a malicious server. The strong encryption safeguards data confidentiality and the signature validation routines protect data integrity.

Replay attacks are a more plausible threat. However these can be countered by the use of version numbers or timestamps.

A denial of service attack could be launched by attempting to fill the server with garbage data, however the signature validation routines will counter this attack. The current implementation of the CNFS server validates the first 8K data block of a file and buffers subsequent blocks until all data have been received. A more sophisticated denial of service attack would be to allow the first 8K of a file to be validated and then substitute bogus data for the remainder of the file. In such a scenario the CNFS client will detect the write failure and may then decide to reissue the write operation and sign the first 8K and every subsequent 48K of data, thus reducing the effectiveness of the attack. A sliding scale could be used to determine the optimum size data block to sign.

5. RELATED WORK

The idea of securing stored data through encryption is already well established. It was originally proposed in the context of the cryptographic file system (CFS) [4], which is implemented by a user level NFS server that runs on the user's own machine. The CFS server can use any underlying file system, including NFS, to store data on disk. The Transparent Cryptographic File System (TCFS) [5] provides similar functionality but has migrated the cryptography to the kernel level thus increasing the transparency to the user. CryptFS [35] provides an interesting departure in design in that it has been implemented as a stackable v-node interface and can be used on top of any underlying file system without modification to that system. Similar to cryptographic access control, the use of cryptography in these systems limit the trust required in the server. The exclusive use of symmetric cryptography means that these systems must rely on the access control mechanism of the underlying file system to allow users to separate read and write access to the files.

Many distributed file systems seek to support flexible user defined access control across different administrative domains. AFS [11, 27, 28] is probably the most widely used wide-area file system. It uses Kerberos [12] to authenticate users, which requires coordination of users and access rights among systems in the different administrative domains. Cryptographic access control does not require users to be known to the server that stores data, which means that data can be freely shared among different administrative domains. SFS [9, 17, 16] is a global decentralized file system, which is similar to AFS. However, it does not rely on a single authentication service, but instead, public-key cryptography is used to authenticate all entities in the system. The identity of the trusted third party, which is required to authenticate the user and the client machine is encoded in the file name. SFS is more flexible than AFS, but essentially suffers from the same disadvantages. Other distributed file systems, such as WebFS [33, 2] and Truffles [21], rely on a trusted third party to authenticate users to servers. This means that user can only collaborate if the system administrators have agreed to coordinate the security policies of their respective domains. Moreover, they generally rely on some trusted infrastructure, e.g., authentication in WebFS is based on X.509 certificates [3]. Coordination among different administrative domains requires some degree of global knowledge and may prove an impediment to the scalability of the access control mechanism.

Operation	NFS		CNFS		Overhead
	Mean	SD	Mean	SD	
8K Read	0.79 ms	0.09	0.63 ms	0.02	-20%
8K Write	0.89 ms	0.11	23.062 ms	1480.16	$+2600\%$
1MB Read	792.84 ms	83.11	874.168 ms	150.85	$+10\%$
1MB Write	1187.89 ms	156.30	1480.16 ms	222.16	$+25\%$

Table 1: NFS and CNFS Server Routines Performance

Capability File Names [20] is a mechanism, that facilitates collaboration among users, without requiring additional services for identification and authentication. However, it still requires the server to invoke a reference monitor to authorize clients. Moreover, security of data in transit is not addressed by the mechanism itself, so it requires an additional mechanism to provide a secure channel between client and server.

6. CONCLUSIONS & FUTURE WORK

In this paper we addressed the problems of scalable access control in large open systems, where the authenticated identity of a principal conveys no *a priori* information about the likely behaviour of that principal.

We proposed a novel access control mechanism, called cryptographic access control, that relies exclusively on cryptography to guarantee confidentiality and integrity of objects (files) stored on potentially untrusted servers in the system. This mechanism eliminates the need to invoke a reference monitor (a potential bottleneck) on the server. Instead, the server delivers data to any requesting client. Access control is performed implicitly on the client machine through the principal's ability to encrypt, decrypt and sign data stored in the system. This means that access control can be performed in a decentralized way on the client, without relying on trusted code or tamper proof devices.

We presented the design and implementation of cryptographic access control in a distributed file system called CNFS, which is based on a standard NFS server. We evaluated the performance of read and write operations on the server, which showed that reading a small file was slightly faster than a standard NFS server, while writing a small file was substantially more expensive because the CNFS server has to verify the signature of data before they are written to disk. The cost of signature verification is amortized when larger files are written to disk, so both read and write operations of larger files are comparable to a standard NFS server.

Security policies are defined by the users and enforced by the way that they distribute keys. The current prototype lacks a secure key distribution mechanism, which would facilitate file sharing.

The current prototype is implemented as a user level library and a modified NFS server. The user level library makes the implementation portable at the cost of transparency and performance, but we would like to integrate the client code into the operating system in order to improve performance. Re-implementing the server, using a log-structured file system, would allow us to defer signature validation until the log is compressed, which should significantly improve the performance of the write operation.

The rapid growth of mobile computing and the deployment of large distributed systems which span multiple administrative domains has created a series of new challenges for security systems. Existing access control mechanisms need to be adapted to reflect the changing operational environment. We have described these challenges and have offered a critique of the suitability of the cryptographic access control mechanism for use in modern distributed systems. We have found the cryptographic access control mechanism to be flexible and scalable yet powerful enough to effectively protect resources in insecure environments.

7. REFERENCES

[1] J. P. Anderson. Computer security planning study. Technical Report 73-51, Air Force Electronic System Division, 1972.

[2] E. Belani, A. Thornton, and M. Zhou. Authentication and security in WebFS, January 1997.

[3] E. Belani, A. Vahdat, T. Anderson, and M. Dahlin. The crisis wide area security architecture. In *Proceedings of the 7th USENIX Security Symposium*, pages 15–29, San Antonio, Texas, U.S.A., January 1998.

[4] M. Blaze. A cryptographic file system for UNIX. In *ACM Conference on Computer and Communications Security*, pages 9–16, 1993.

[5] G. Cattaneo and G. Persiano. Design and implementation of a transparent cryptographic filesystem for Unix. Unpublished Technical Report, ftp://edu-gw.dia.unisa.it/pub/tcfs/docs/tcfs.ps.gz,, July 1997.

[6] C. Czezatke and M. A. Ertl. LinLogFS — a log-structured filesystem for Linux. In *Freenix Track of Usenix Annual Technical Conference*, pages 77–88, 2000.

[7] C. Ellison, B. Frantz, B. Lampson, R. Rivest, B. Thomas, and T. Ylonen. Spki certificate theory. Technical Report 2693, Network Working Group, IETF, September 1999.

[8] M. Fischer, N. Lynch, and M. Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM*, 32(2):374–382, 1985.

[9] K. Fu, M. F. Kaashoek, and D. Mazières. Fast and secure distributed read-only file system. In *Proceedigs of the 4th Symposium on Operating Systems Design and Implementation*, pages 181–196, San Diego, California, U.S.A., October 2000.

[10] A. Harrington. Cryptographic access control for a network file system. Master's thesis, Trinity College, Dublin, 2001.

[11] J. H. Howard, M. L. Kazar, S. G. Menees, D. A. Nichols, m. Satyanarayanan, R. N. Sidebotham, and M. J. West. Scale and performance in a distributed file system. *ACM Transactions on Computer Systems*, 6(1):51–81, 1988.

[12] J. Kohl and C. Neuman. The kerberos network authentication service (v5). Request for Comments (RFC) 1510, Network Working Group, IETF, September 1993.

[13] B. Lampson, M. Abadi, M. Burrows, and E. Wobber. Authentication in distributed systems: Theory and practice. *ACM Transactions on Computer Systems*, 10(4):265–310, November 1992.

[14] B. W. Lampson. Protection. In *Proceedings of the 5th Princeton Symposium on Information Sciences and Systems*, pages 437–443, mars 1971. reprinted in *Operating Systems Review, 8, 1* January 1974 pages 18–24.

[15] R. Needham M. Burrows, M. Abadi. A logic of authentication. *ACM Transactions on Computer Systems*, 8(1):18–36, February 1990.

[16] D. Mazières. Security and decentralised control in the SFS distributed file system. Master's thesis, MIT Laboratory of Computer Science, 1997.

[17] D. Mazières, M. Kaminsky, M. F. Kaashoek, and E. Witchel. Separating key management from file system security. In *Proceedings of the 17th Symposium on Operating Systems Principles*, pages 124–139, Kiawah Island, S.C., U.S.A., 1999.

[18] B. D. Noble, B. Fleis, and L. P. Cox. Deferring trust in fluid replication. In *Proceedings of the 9th ACM SIGOPS European Workshop*, pages 79–84, Kolding, Denmark, September 2000.

[19] Telecommunication Standardization Sector of ITU. *Information Technology — Opens Systems Interconnection — The Directory: Authentication Framework*. Number X.509 in ITU–T Recomandation. International Telecomunication Union, November 1993. Standard international ISO/IEC 9594–8 : 1995 (E).

[20] J. T. Regan and C. D. Jensen. Capability file names: Separating authorisation from user management in an internet file system. In *Proceedings of the 2001 USENIX Security Symposium*, pages 221–234, Washington D.C., U.S.A., August 2001.

[21] P. Reiher, T. Page, S. Crocker, J. Cook, and G. Popek. Truffles—a secure service for widespread file sharing, 1993.

[22] Mendel Rosenblum and John K. Ousterhout. The design and implementation of a log-structured file system. *ACM Transactions on Computer Systems*, 10(1):26–52, 1992.

[23] Jerome H. Saltzer, David P. Reed, and David D. Clark. End-to-end arguments in system design. *ACM Transactions on Computer Systems*, 2(4):277–288, November 1984.

[24] R. Sandberg, D. Goldberg, Kleinman S, D. Walsh, and B. Lyon. Design and implementation of the Sun Network File System. In *Proceedings of the Summer 1985 USENIX Conference*, pages 119–130, Portland, Oregon, ¡C9¿tats–Unis, June 1985.

[25] D. S. Santry, M. J. Feeley, N. C. Hutchinson, A. C. Veitch, R. W. Carton, and J. Ofir. Deciding when to forget in the elephant file system. In *Proceedings of the 17th Symposium on Operating Systems Principles*, pages 110–123, Kiawah Island Resort, South Carolina, U.S.A., 1999.

[26] M. Satyanarayanan. A study of file sizes and functional lifetimes. In *Proceedings of the Eight Symposium on Operating System Principles*, pages 96–108, Pacific Grove, California, U.S.A., 1981.

[27] M. Satyanarayanan. Integrating security in a large distributed system. *ACM Transactions on Computer Systems*, 7(3):247–280, 1989.

[28] M. Satyanarayanan. Scalable, secure and highly available file access in a distributed workstation environment. *IEEE Computer*, pages 9–21, May 1990.

[29] B. Schneier, J. Kelsey, D. Whiting, D. Wagner, and C. Hall. Twofish: a 128-bit block cipher, 1998.

[30] M. I. Seltzer, K. Bostic, M. K. McKusick, and C. Staelin. An implementation of a log-structured file system for UNIX. In *USENIX Winter*, pages 307–326, 1993.

[31] Sun Microsystems Inc. Nfs: Network file system protocol specification. Request for Comments (RFC) 1094, Network Working Group, March 1989.

[32] A. S. Tanenbaum, S. J. Mullender, and R. van Renesse. Using sparse capabilities in a distributed operating system. In *Proceedings of the 6th International Conference in Computing Systems*, pages 558–563, June 1986.

[33] A. Vahdat, P. Eastham, and T. Anderson. Webfs: A global cache coherent file system. Department of Computer Science, UC Berkeley, Technical Draft, 1996.

[34] E. Wobber, M. Abadi, M. Burrows, and B. Lampson. Authentication in the Taos operating system. *ACM Transactions on Computer Systems*, 12(1):3–32, 1994.

[35] E. Zadok, I. Badulescu, and A. Shender. Cryptfs: a stackable vnode level encryption file system. Technical report, Computer Science Department, Columbia University, 1998.