

Flexible Security for the WiND Filesystem

Muthian Sivathanu, Omer A. Zaki
{muthian, ozaki}@CS.Wisc.EDU

Abstract

Due to the unending increase in scalability and performance demands, the network attached storage paradigm is being adopted as the solution for large storage systems. This new shift seeks to decentralize storage elements over a network where the idea is to transfer data directly between the storage disks and client machines thereby completely bypassing the fileserver machine bottleneck. The new paradigm opens up a host of novel security issues and protecting the information in transit between the distributed storage entities over a potentially unsecure network becomes essential. In this paper we report on our project wherein we sought to explore the security issues that come up in a network attached storage environment. Our work was done in context of the Wisconsin Network Disks project (WiND). We implemented security features into WFS, an experimental distributed file system built for WiND. Our techniques are performance conscious and do not compromise on the scalability of the system. We allow the user to choose an appropriate security level on a per-file basis. By storing data encrypted on disk, using capabilities, and accomplishing integrity and authentication in a single step, our security features have a minimal impact on system performance.

1 Introduction

Network attached storage is beginning to be accepted as the solution for highly scalable, high-throughput storage systems. The traditional architecture of a single or a few file servers managing the entire file traffic in a distributed file system fails to scale in large systems that have very high bandwidth requirements. These can arise either in the context of a distributed filesystem being shared by thousands of clients, or in the case of a few heavily demanding I/O intensive applications like video servers that seek to achieve high I/O bandwidths by parallelizing access to multiple disks. The problem with traditional architectures is that although the total bandwidth realizable from the disks might be much larger, the CPU and I/O bus at the centralized server tends to very easily become a bottleneck even in moderately sized systems. Network attached storage decentralizes file access traffic by allowing direct communication between the clients and the disks, thereby removing the notion of a centralized file server. By doing this, it potentially permits client applications to utilize the full capacity of the disks without being limited by any other constraint.

This new paradigm comes with its new problems and our work focuses on the security concerns. The clients and disks now communicate over a potentially unsecure network, requiring their communication to be protected from adversaries. Since the main motivation for decentralizing storage is performance and scalability, security at the cost of either of these is unacceptable. Our work was done in context of the Wisconsin Network Disks project (WiND) which seeks to provide a network attached storage infrastructure that is self-managing and scalable. We implemented security features into WFS, an experimental distributed file system built for WiND. The goals of our project were thus to investigate the security issues in WiND and implement security features in WFS without compromising its performance and scalability. Since, security does come at some cost, we sought to make our implementation flexible so that users would have control on how much security they want.

2 Background

This section describes the motivation behind using network-attached storage. It also describes the WFS architecture briefly.

2.1 Move toward network attached storage

Current trends in computing reflect a need for storing and accessing large amounts of data over local area networks. The traditional architecture of a distributed file system built from file servers containing and directly managing storage disks appears to be unable to address such performance demands. File server CPU and the I/O bus tend to become bottlenecks while the bandwidth realized from the disks continues to increase. To alleviate this situation the network attached storage paradigm was proposed wherein the idea is to transfer data directly between the storage subsystem and the client machine thereby bypassing the file server machine bottleneck. The network attached secure disk (NASD) project at CMU pioneered this approach in a system wherein the disks are distributed over a network and clients are allowed to directly access the disks, without having to go through a centralized file server. This contrasts with what is done traditionally in distributed filesystems like NFS or AFS. Since the file server bottleneck is avoided this paradigm is attractive for providing scalable storage as clients can potentially realize the complete bandwidth of multiple disks. In NASD a requesting filesystem client first contacts a central file manager to obtain information and permission in the form of a cryptographic capability to access the stored data. It then presents the capability to the NASD drive and engages in data access directly with the drive. Such an architecture has several advantages [Gibson97][A-D01] which can be summarized as follows:

- *Scalability*: The contention for the CPU or the I/O bus of a central file server no longer limits the growth of the storage system.
- *Availability*: Since multiple data paths exist to the stored data, failure of a file server will no longer render all its data inaccessible. This makes the storage system more fault-tolerant.
- *Extensibility*: The storage system using NASD is not constrained by single machine hardware and attaching storage directly to the network presents opportunity for incremental growth.
- *Specialization*: The distributed nature of NASD allows the storage system to adapt and cater to varying requirements. If more storage is desired additional drives can be added and similarly for higher processing requirements additional nodes can be attached.

2.2 WFS

The WiND [A-D01] project aims to develop an adaptive and a self-managing scalable storage system and its storage architecture is based on the NASD work mentioned above. However, the focus and scope of these two projects differ. NASD deals with providing disk level semantics that give the illusion of a single disk to upper software layers such as the filesystem while hiding the details of the distributed storage. It provides an infrastructure on which any type of a filesystem can be built and hence the functionality it supports is kept generic. WiND on the other hand is more specialized and the storage system is tailored for a companion file system (WFS). This way WiND provides a single file system and hides the details of its distributed nature. This tight coupling presents opportunities to specialize the storage for the file system which makes the goal of adding adaptation and self-management to the entire storage system more tractable. Our objective was to provide security for the storage system and this coupling permitted us to do so while not compromising on performance and flexibility. For instance, in NASD a file system is treated as one object [Gibson97] and as a consequence a uniform security policy has to be enforced for all constituent files of that file system. In WiND we are able to tune the security policy on a per-file basis.

WFS Architecture

A distributed filesystem provides access to shared storage that is organized logically under a single name space. The fundamental operations supported by all distributed filesystems are operations for storing and retrieving data, and manipulating the namespace. In a traditional domain usually a single file server takes care of all these operations and hides the low-level details like disk layout from the clients. The file server also enforces other file management properties like access control. Clearly the file server has the potential of becoming a bottleneck. WFS adopts a different architecture along the lines of NASD. It consists of three main components: WFS client, filemanager (FM) and, the network attached drives as shown in figure 1.

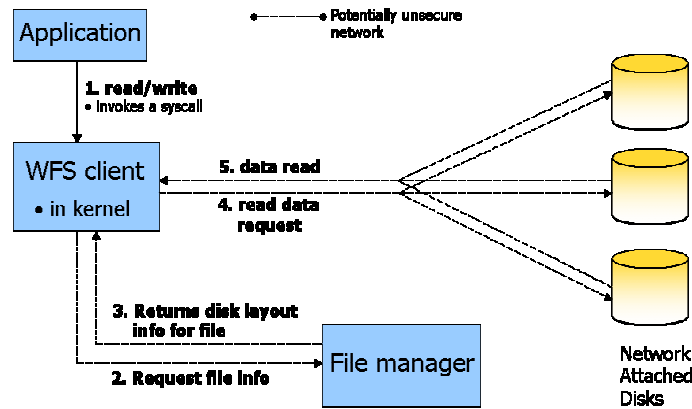


Figure 1: WFS architecture

We will now consider each of these components individually.

- *WFS client*: The WFS client runs in the operating system kernel of the user's machine. It projects a Virtual File System (VFS) interface that allows it to hide implementation details from upper layers of software like user programs. For instance, it transparently takes care of striping the data over the network-attached drives. User programs interact with WFS through system calls (e.g., write, open, stat, link). Other parts of the kernel like memory management functions may also give requests to WFS. To support adaptation in WiND, WFS can do adaptive striping based on the relative performance of the drives. To satisfy user requests WFS contacts the file manager and obtain physical details about the file of interest. Using this information it begins the data transfer directly with the relevant drives. For instance, on a read request from a particular file the WFS client first contacts the FM to obtain layout information for the file. It then uses this information to request the relevant drive for the part of the file that it stores.
- *File manager*: It maintains meta-data for the file system. It is the only entity that has a hierarchical view of the filesystem and stores the mapping between a logical file and its constituent objects that may be striped over several drives. It furnishes this information to a requesting WFS client that makes a request on behalf of some user. In addition to this FM takes policy decisions for the file system. For instance, it handles access control wherein it decides which user is authorized to access which file. However, the enforcement of these decisions is done by the drives to which it conveys its decisions. This way file management duties are shared between the FM and the drives. In our implementation the FM is a user-level process that runs on a machine.
- *Network attached drives*: These take care of the storage of data and provide an object level interface which is yet another level of abstraction. The low level details of mapping the object regions to disk level sectors are taken care by the drive and hidden from the other components of WFS. While the FM takes policy decisions the drive actually enforces them. It allows access to a certain object only if the user is authorized to access that object. In the case of the vanilla WFS (that is devoid of security), this

functionality may not be that impressive, however, in the case of secure WFS this is very important. The drives are implemented as user-level processes.

3 Security for WFS

This project addresses security issues that arise when a distributed filesystem exists in a network attached storage environment. Conventional distributed filesystems rely on a central server that manages all its disks directly. In such a setup, there are no security concerns as regards to getting the data from the disks. As long as the channel between the fileserver and the client is secure the security risk is minimal. In WFS on the hand, drives are physically distributed over a potentially unsecure network requiring all channels between with the WFS clients and FM to be secure in order to frustrate the adversaries. In this section we discuss the threat model that we are catering for. We also present the security issues that came up and eventually influenced our implementation and protocols.

3.1 Threat Model

In WiND the WFS client, FM and the drives communicate with each other over a potentially unsecure network. We thus, have to guard against the standard threats wherein an adversary may add, drop or modify a message sent over the network. Furthermore there is a privacy requirement wherein we do not want the adversary to know what communication is taking place as this may lead to replay attacks amongst other serious compromises.

As mentioned before, the file management work is partitioned between the FM and the drives. The FM takes policy decisions and it is each drive that actually enforces it. This poses a need to communicate the FM's decisions to the drive in a secure manner. For our project security is not the only requirement: the method used must also scale and be efficient. Lets consider the case of access rights. If a user wants to access a file it makes a kernel call to the WFS client which then makes an access request to the FM on the user's behalf. The FM then makes its decision to grant or deny access to WFS client's request. If it has decided to grant access this information must be conveyed to the relevant drives. We use a cryptographic capability mechanism to accomplish this. What actually ends up happening is that this cryptographic capability is returned to WFS client which then uses it to carry out the desired operation on the drives.

As far as the drives are concerned they do not trust the machine running the WFS client. This is because a given user has physical control over the machine and can potentially insert malicious software into it and cause the WFS client to lie about its privileges or capabilities.

The users on the other hand trust the machines they use. We believe that this is a reasonable assumption to make since the machine's kernel will have knowledge of the user's password. The machine also decrypts and encrypts messages and data on behalf of the user, which requires it to have access to the user's private key.

However, the FM is trusted by both the drives and the WFS clients as it has access to all the users' and drives' private keys. We assume that adequate security arrangements are made for the physical security of the FM as any exception to this would expose the entire storage system to an adversary.

To address the above-mentioned threats we present three classes of issues that we need to take care of. These are:1) access control, 2) authentication, and 3) integrity and privacy. The exact mechanisms we adopt must take the scalability and performance requirements into account i.e. they must not impede scalability and/or have a deleterious impact on performance.

3.2 A Straightforward Approach

A possible approach that can handle our threat model will require a secure channel between all entities of WFS: WFS client, FM and network attached drives. This can be done via a secret session key that is exchanged using some standard protocol like Kerberos. Authentication is based on this session key.

Message exchanged between WFS entities can also be encrypted with the session key. Each end involved in the message exchange does encryption and decryption. For integrity we again use the session key. Each end computes a MAC of the message to be exchanged.

Clearly a naïve approach will not work for WFS due to the performance and scalability constraints that we have. For instance, encryption and decryption on every read/write would degrade performance to an abysmal level. Studies and measurements by [Gobioff992] reveal that managing encryption at disks has an adverse impact on disk bandwidth. Using Kerberos is not feasible as it is very heavy weight. It requires encryption & decryption even for authentication.

3.3 Our approach

We propose an approach in which data is stored encrypted on disk. Here the disks don't have to deal with costly encryption and decryption and the WFS client takes care of encrypting data with suitable keys. The goal is to offload work from the disks as much as possible, which we believe will result in scalability and high throughput. We use an efficient hash based scheme for authentication.

To convey the FM's decision to the drives we use a cryptographic capability mechanism. The cryptographic capability is a digest of the object information and operation the WFS client wants to perform.

We also provide a mechanism by which a user can tune the security level on a per-file basis. Performing encryption and decryption at the WFS client for every read or write may be too much of a concern for a user who wants very high performance but doesn't care if the data comes through an insecure channel. Also, a mechanism exists to automatically infer the security level of a file based on the file's permissions.

We now discuss in detail each class of security issues (Access control, Authentication, Privacy & Integrity) that were addressed to deal with the threat model. While describing the security issues we also report on the solution we implemented.

3.4 Authentication

Among the security issues all filesystems must address are authorization (users are allowed to access only those files that they are authorized to access) and authentication (users are identified in a way so that one user cannot impersonate another). Although these problems exist in a local filesystem also, the solution is quite straightforward since all access control decisions are centralized at the kernel. Here, the *userId* of the process making the file access request securely and reliably indicates the user on whose behalf the request is initiated. However, when the entity initiating the request and the entity granting access are separated across machines over an insecure network, the issue of authenticating the initiator is no longer trivial. Traditional network filesystems like NFS or AFS solve this problem by using a separate third-party authentication mechanism like Kerberos, which takes care of ensuring that only authenticated users gain access to the file server. Such protocols securely identify the initiator of a file access request to the fileserver. The problem now reduces to the local filesystem case since the external authentication mechanism supplies a secure notion of a *userId* to the file server, which can now act as if it was a request emanating from the local machine by a process running on behalf of the specified *userId*.

In WFS there is yet another dimension to the problem of authentication. Unlike traditional network filesystems like AFS, where the file server application and the disks reside on the same physical machine thereby guaranteeing a physically secure path between the file server process and the disks, WFS provides for the FM (which can be thought of as an equivalent of the fileserver in a traditional architecture) to reside on a separate machine from that of the disks. In other words, the FM now needs to communicate over an unsecure network to enforce its access control decision at the disks. Also, to avoid becoming a bottleneck the FM must be used as sparingly as possible. By doing just the bare minimum work at the FM and diverting most of file traffic directly between the WFS client and the drives, we seek to achieve scalability. Hence, the notion of every file access (read/write) going through the fileserver and being authenticated is

no longer a reasonable approach. This for all practical purposes rules out the possibility of the FM taking these authentication decisions on a per-file operation basis. There is a need for a solution where the FM takes decisions on a macro level and leaves it to the drive to look into micro-level enforcement of those decisions.

This issue of authentication in a network attached storage domain has been partially addressed in [Gobioff992]. There is however a fundamental difference between NASD and WFS in terms of the layer in which they operate. NASD was designed to operate at a storage system level and was aimed at serving as an extension to the local disk in a client machine: the interface provided to the client filesystem is in the form of a remote disk device driver. This means that whole filesystems can be installed on NASD rather than the local disk. A conventional local disk partition would now be a set of NASD objects striped across multiple disks. Consequently, NASD delegates much of the actual authentication tasks to the filesystem level where they have to be implemented in a way that best suits the policies and requirements of the specific filesystem. WFS, on the other hand, aims at providing a highly scalable distributed filesystem using the network storage paradigm, and therefore many of these issues left to the implementation of the file system, need to be addressed. Also, in WFS, the disks and the client filesystem are very closely tied in that there is no clear demarcation between the storage layer and the filesystem layer (disks can be accessed only through WFS). In view of this close coupling, optimizations can be performed with the extra information available.

3.4.1 Per-User Authentication

Identifying the authenticating entity is an essential part of any authentication mechanism. There are two plausible alternatives for the authentication model. The first is a per-user model, where the entity authenticated is a user logged on to some machine. This model is used by popular distributed authentication protocols like Kerberos. A one-way function is applied to the user's password and this is used as the key and forms the basis for authentication. This way, every file *read* or *write* from/to a disk must be authenticated for every user who accesses it.

However, such a model is not applicable in a distributed filesystem context as it makes the sharing of the file cache among multiple users difficult. Since the protocol requires that every user get itself authenticated before getting access to data, it would mean that each user has to go to the disk for authentication and object access, regardless of whether the same object is cached locally on behalf of some other user or not. This clearly is inefficient in terms of performance as the cache at the client is being used sub-optimally. The cache is being partitioned on a per-user basis which indirectly increases the load on the disks. This runs contrary to our goal of providing a high-throughput scalable storage system.

3.4.2 Per-Client-Machine Authentication

A simple alternative to per-user authentication is to perform authentication on a per-client machine basis. The client machine authenticates itself with the disks, and once it gains access to a set of objects on disk, is free to decide the set of users that can be granted access to the objects. The client kernel thus, becomes the authenticating entity for the users on its system. Now multiple users can share the file cache and it can be thought of as being *owned* by the kernel.

The per-client approach has a few limitations. Authentication requires maintenance of access control information on a per-machine basis. For instance, we need to have rules like: *Machine A is authorized to access file foo*. These are not intuitive as files have been traditionally considered as being *owned* by a user rather than by a specific machine. Also, this may impose a constraint that an object can only be accessed from specific machines. This may be a serious shortcoming for a distributed file system. Another limitation is that the model assumes that a client machine will always be able to securely identify itself. This is not a reasonable assumption given that our threat model assumes that disks do not trust client kernels. This demands a scheme where the lack of physical security of a client machine does not compromise the authentication mechanism.

3.4.3 Hybrid approach to authentication

Having discussed the two possible authentication models, we now present our solution, which is a hybrid of the two models. We do per-user authentication at the disks, similar to the first approach. As before, the user's password is transformed by a one-way function into the client's private key. The FM authenticates users based on the access control list it maintains for each file. Client machines always talk to the FM on behalf of some user. So, if file *foobar* is being shared by users *A*, *B* and *C*, no client machine can access the file unless at least one of *A*, *B* or *C* are logged on. This way we ensure that a client machine compromise does not lead to a authentication mechanism compromise.

From the perspective of the client machine, however, the scenario is quite different, much like the second authentication model discussed. Once the client machine gains access to a file on behalf of some valid user, it gets to decide which other users to give access to. It uses its copy of the access control list obtained during *file lookup* to determine the set of users who can be granted access to the file. This is reasonable as users trust the machines on which they log on.

From the viewpoint of the FM, it grants access to a particular user and then allows the client machine to use its cached copy to serve other users. Again, this is plausible because the client machine was able to gain access to the file on behalf of some valid user. This implies that the machine is under the control of a user with legal access rights to the file. This is sufficient ground to trust the client machine with the file. If this assumption is wrong, the maximum we stand to lose is that the particular file is compromised, but this is bound to happen if the user is careless about giving her password to untrusted machines. A point to note is that the degree of compromise is limited to the access privileges of the user on whose behalf the trust was established between the FM and client, and hence, the damage is limited to the indiscreet user.

3.4.4 Authentication at disks

As mentioned above, the issues of authentication and access control in WFS are considerably different from that of traditional network filesystems as the FM and disks are physically separated and communicate over an unsecure medium.

We adopt a mechanism similar to the NASD model where the access control task is distributed between the FM and disks. The FM takes access control decisions based on its state information, and the disks just enforce the decisions communicated by the file manager. By having this clear separation between access control decision-making and access control enforcement, we obviate the need for replicating access control state at each of the disks. High-level sequence of steps taken to read/write a file in WFS are given as follows:

- 1) To access a file, the WFS client makes a *lookup* call to the FM on behalf of the user requesting access to the file.
- 2) FM sends back layout information of the file. This contains a set of disks and corresponding object ids of objects that contain the striped file. For each (disk, object) pair, FM also returns read and write capabilities for the drive if the given user is authorized to receive them. These capabilities are the means by which FM communicates its access control decisions to the disk, which enforces them.
- 3) The client now contacts the disks directly with the read or write request, quoting the capability received from FM.
- 4) The disk verifies the capability and grants access to file data if permitted.

To achieve scalability this model aims at minimizing traffic at the FM as much as possible. The FM is contacted only during *lookup* of the file path (which normally takes place during a *file open*). Moreover, capabilities and file layout information are cached at the client kernel as part of the in-core inode of the file. Other clients that open the file later can use the same capability. To provide for this reuse of capabilities, the capability generation mechanism does not take *userId* into account.

Protocol

The following are the notations we follow while describing the various protocols in this paper:

K_C	\rightarrow	<i>private key of user</i>
K_D	\rightarrow	<i>private key of disk</i>
K_{cap}	\rightarrow	<i>capability key</i>
K_F	\rightarrow	<i>private key of FM</i>

Each user has a private key that is derived from a one-way function applied to its password. This key is known only to the FM and the user. Each disk has a secret key that is known only to the FM and the disk. Thus, the FM has access to the private keys of all the users and disks. This creates a potential single point of security failure but we expect the FM to be physically secured. This, we believe, is reasonable considering that we need to secure just a single machine. Most authentication models have such centralized points of failure, which need to be addressed accordingly. For example, in Kerberos, if the authentication server is attacked and compromised, the intruder has access to the private keys of all clients and services. We argue that by reducing the placeholder of sensitive information to just a single machine, i.e. the FM, we make the task of physically securing the machine much more realistic and cost-effective.

The WFS client in the kernel keeps track of the private keys of all users who are currently logged into it and accessing WFS. Whenever a process running on behalf of a user invokes a file access routine, the client first tries to get the layout information of the file and the set of capabilities to gain access to the object. To do so, it first sends a lookup request to the FM with the *userId*, file name to be looked up, and the parent directory in which to *lookup*:

<i>Client to FM</i>	\Rightarrow	<i>lookup (dir, fname, clientId)</i>
<i>FM to client</i>	\Rightarrow	<i>driveId, objId, {K_{cap}}K_C</i>
		$K_{cap} = \text{HMAC}_{K_d}(\text{driveId}, \text{objId}, \text{operation})$
<i>Client to drive</i>	\Rightarrow	<i>{request_arg, request_signature}</i>
		$\text{request_arg} = \{\text{driveId}, \text{objId}, \text{operation}, \text{TS}\}$
		$\text{request_signature} = \text{HMAC}_{K_{cap}}(\text{request})$

HMAC is a keyed message authentication code, built on top of MD5, a message digest algorithm. HMAC maps an input string of arbitrary length into a fixed length signature, with the property that given a message x and $\text{HMAC}(x)$, it is computationally infeasible to compute another message x' such that x and x' converge to the same signature. It is also infeasible to compute the key based on which the digest was generated.

The capability key is generated by applying a HMAC, keyed by the private key of the disk (shared between disk and FM) to the *driveId*, *objId* (Object Id) and, *operation* that can either be a read or a write. Since it requires K_d , only the FM and the disk can generate K_{cap} . K_{cap} is transmitted encrypted in the private key of the user on whose behalf the request was initiated so that no eavesdropper can recover the capability key. The client machine receives the encrypted capability, decrypts it with the user's private key and recovers K_{cap} . The client now frames its request arguments to each disk. The request arguments contain *driveId*, *objId* in the drive, and a timestamp. It then computes a HMAC keyed by K_{cap} of all the request arguments to generate *request_signature*. Then it transmits the request and signature to the drive. Since K_{cap} was sent encrypted in the user's private key, no adversary can change the request arguments without harming the correspondence between the arguments and the signature.

The drive obtains request arguments, takes out (*driveId, objId, operation*) portion of the request and computes an HMAC on it using its private key K_d . This should give K_{cap} , the capability key for access to this object, since the FM uses the same computation to generate K_{cap} on a file *lookup*. Now that the disk has K_{cap} , it computes HMAC on the entire request arguments keyed by K_{cap} to check if it converges to the same signature transmitted. If it does, two conditions are guaranteed: a) the request arguments were not modified in transit, and b) the same valid capability key was used to generate the signature, which serves to authenticate the user. If the signatures do not match, one of these conditions is false and request is rejected. Thus we find that this mechanism achieves both integrity of arguments and authentication in a single step. The disk always ensures that the operation performed originates from someone authorized to perform the operation and that it is doing exactly the operation that the originator wanted it to do.

The timestamp field in the request serves to guard against replay attacks. By embedding the timestamp in the signature computation process, we ensure that it cannot be changed without invalidating the request. At the drive, we check for freshness of the message - here we assume that clocks are loosely synchronized upto a skew threshold. The FM keeps track of the time offsets between itself and each of the disks and this time offset is securely communicated to the client as part of the *lookup* process, so that the client knows which timestamp value to use in the request. Preventing replay attacks is crucial in this environment because we do not want an adversary, for example, to undo the effect of a recent write by replaying an older recorded write call.

We thus achieve authentication and integrity of reads and writes between the client and FM in such a way that the FM is given a negligible role to perform. What is also interesting in this scheme is that we have achieved this without employing absolutely any encryption or decryption at the disks. The low CPU power at the disks makes this a very attractive option since costly encryption/decryption at every read/write has the potential of drastically lowering the throughput of the disk. By using a scheme that utilizes digests alone (which is much cheaper than encryption), we almost get authentication for free since to ensure integrity, we should have computed the digests anyway. Moreover, since the capability key is not random but rather always the same for a particular group of arguments, it can be cached for frequently accessed objects at the disk. This way the disk need not compute the capability key on every read or write. This cache will simply be indexed by the object id and the operation contained in the request.

This also explains why we did not choose to use an external authentication protocol like Kerberos in our model. The requirements of the FM having to convey some decision to be enforced by the drive looks very similar to the Kerberos architecture of the authentication server making some decisions to be enforced at the TGS or other services. The reasons for implementing our own authentication protocol are: a) Kerberos is too heavyweight since it uses encryption and decryption on every authentication round. This can be too costly for the disks. b) Kerberos provides a per-service authentication whereas in this case we require a much finer grained authentication at a per-object level. While Kerberos can identify two entities to each other, it does not provide for identifying them to each other *for a specific role*. In our mechanism, since the capability encodes the object id, we automatically get the fine-grained control we require.

3.4.5 Authentication at Filemanager

The above section discussed about the protocol for authenticating read and write requests at the drives. There are certain operations that involve updating the file metadata state available at the FM. These operations include *create, unlink, mkdir, rmdir*, etc. It is crucial that these operations be performed only by authorized entities. Also, the operations should not be modified/distorted in transit over the unsecure network (e.g. An intruder should not be able to modify the file name in a delete request and cause the wrong file to be deleted). So, we need to take care of authentication and integrity for these requests also. The protocol we use to authenticate and preserve integrity of requests at the FM is given below.

Protocol

$$\begin{aligned} \text{Client to FM} &\Rightarrow \{request, signature\} \\ request &= \{filename, operation, userId, TS\} \\ signature &= HMAC_{K_C}(request) \end{aligned}$$

The protocol is similar to the protocol for authentication at disk, except for the fact that it requires just a single phase since the FM need not communicate its decision to any external entity. The request contains the filename, the operation to be performed, the *id* of the user on whose behalf the operation is to take place, and a timestamp. The client also sends a signature of the request that is a HMAC of the request keyed by the secret key of the user. *userId* is required to find the user key to use when computing the signature. If the signatures match, the FM is guaranteed that the request originates from the user it claims to originate from (since nobody else has access to K_C), and that the request has not been modified in transit. The timestamp included guards against replay attacks as mentioned above.

3.5 Data Integrity

To achieve our goal of providing the same security guarantees that a local filesystem provides, integrity of data is an important issue to be addressed. In the local file system, the buffer manager issues a read/write to the device driver which does the operation on the local disk and returns the data. The data that the buffer manager receives during read is guaranteed to be the data that the disk actually has. To ensure further integrity, disks have a CRC mechanism that detects possible loss of data integrity. Since the disks and the clients are on the same machine, this is sufficient to ensure integrity. But in a distributed filesystem like WFS, data could be modified during transit by an intruder. We need to reliably inform the user if the data it sees is actually what the drive wanted it to see (for reads) and vice versa.

We once again use the HMAC signature generation to accomplish data integrity enforcement. The capability key K_{cap} obtained during lookup can be used to generate a HMAC on the data block, and the signature can be transmitted along with the data during a write. The drive applies the same HMAC to the data to check if it converges to the same signature received. If yes, the data hasn't been modified during transit and can go through. Similarly for reads, the disk computes the signature which can be verified at the client end to see if the read data is valid.

3.6 Data Privacy

In a local file system, the path between the disks and the user's memory buffer is an I/O bus which cannot be intercepted by any intruder if the user has physical control over its machine (which it usually does). To extend the same guarantees in the context of WFS, we must prevent any adversary from being able to snoop on file data and get access to valuable information. The simple solution of both communicating entities establishing a secure channel (i.e. encrypting and decrypting every data that is sent) is not a feasible alternative in this context because of the stringent CPU requirements at the drive. Encrypting and decrypting data during every read or write at the disk will bring down the throughput of the disk drastically. Studies in [Gobioff992] have revealed that this brings down performance by around 98%.

Off-loading work from the disks as much as possible is the key to achieving scalability. In a large distributed filesystem like WFS, there can be numerous clients reading/writing to a particular disk and it is important that disks do as minimal work as possible. We believe that it is almost always better in such large systems to push much of the work to the clients, so that the work gets distributed among multiple clients instead of exploiting a single CPU at the server. Going by the same philosophy, we adopt an approach wherein data is stored encrypted on disk, thereby obviating the need to do decryption/encryption at the disk. The disk just reads and writes whatever data it receives, as is, and it is upto the client to encrypt the data properly during writes and decrypt the received data during reads. This does not impose any additional overhead at the client because the client in any case has to encrypt/decrypt data since data has to traverse an

unsecure network. Thus we have relieved the disk of costly crypto operations during reads/writes, while still ensuring privacy of data transmitted across the network.

Each file in WFS has an *access key* which is the key used to encrypt data on the disk objects pertaining to the file. The access key is maintained as part of the file metadata at the file manager, and is transmitted securely to the clients during a *file lookup*. The client uses this key to encrypt all data it writes to disk and to decrypt all data read. This key is implicitly shared by all users that have access to the file.

3.7 Access Control

WFS uses access control lists to provide for flexible access control policies. The *acl* is stored as part of the file metadata at the file manager and is returned as part of the layout information during a *file lookup*. We also provide a system call interface to read or write into the *acl* of a file. A utility *fs* gives a flexible interface to manipulate the *acl* information of a file.

An important consequence of our decision to store data encrypted on disks is that changing *acls* of a file gets complex. If the *acl* is changed to a more restrictive set re-encryption is required. For instance, if an user A, who initially had read access to a file, is deprived of read access, we need to ensure that A can no longer gain access to the file data. It is not sufficient to ensure that the read authentication for the user is rejected at the drive in future. This is because, though A may not be allowed a legal read at the disk, she still has access to the *access key* of the file and therefore can eavesdrop over the network to potentially get access to the entire file contents, which goes against the privacy guarantees we seek to provide. Because of this, we need to re-encrypt the file with a newly allocated access key, whenever there is downgrading of access privileges. Considering the low frequency of *change_acls*, this does not seem to be a major concern.

To re-encrypt the file, the entire file data has to be read, decrypted, encrypted with the new key and written back to disk. If we have a large file spanning hundreds of megabytes, this can take considerable time. Since it would clearly be better for the file to be available during the re-encrypt process, we do not write the data in place, but rather allocate a separate set of objects into which the new data is to be written. While the re-encrypt is in progress, we allow read-only access to the old file, and once the operation completes, the layout information is modified to reflect the new set of object, and the old objects are freed.

We had to make a design choice in choosing the entity (WFS client or the FM) that would do the re-encryption of the file. When the client sends a *change_acl* to the file manager, the FM could perform the entire task and return control to the client. However, this goes against our philosophy of trying to minimize the work at a central entity. The FM is especially a critical resource and overloading it with a task such as re-encryption has the potential to be of detriment to system scalability and availability. Thus, the approach we follow is to delegate the work to the WFS client that initiated the *change_acl*. The FM only allocates a new set of objects and grants capabilities for the new set along with a new access key. The WFS client then takes care of reading the data from the old set of objects and writing into those indicated by the new layout information. After the WFS client is done, it sends a *change_acl_commit* request to the FM which then updates the metadata to point to the new set of objects.

To carry out this operation securely, we implemented a separate protocol for *change_acl*. The FM maintains the set of *change_acl* requests that are pending commits. There must be a reliable way of ensuring that a commit is securely matched with the corresponding *change_acl* request. The protocol we follow is described below.

Protocol

Client to FM \Rightarrow *request, signature*

request = {*filename, new_acl, userid, TS*}

signature = $HMAC_{K_c}(\text{request})$

$FM \text{ to client} \Rightarrow new_key, new_cap, change_acl_id, id_signature$
 $new_cap = \text{capability to new objects}$
 $id_signature = HMAC_{K_f}(change_acl_id, filename)$

$Client \text{ to FM} \Rightarrow request, signature$
 $request = \{change_acl_id, id_signature, filename, TS\}$
 $signature = HMAC_{K_c}(request)$

The *id_signature* field in the FM response ensures that the *id* reported back by the WFS client is a valid *id* that the client received from the FM. In other words, the client cannot send a spurious *id* and cause commit of a *change_acl* that is still in progress. By generating the *id_signature* through K_f , the private key of the FM, FM ensures that the correspondence between *id* and *filename* matches and the client is permitted to send only genuine commits. When the FM receives a commit (the final message), it computes $HMAC_{K_f}(change_acl_id, filename)$ and checks if it matches with the *id_signature* received. If yes, this is a genuine commit. Timestamps again serve to prevent replay attacks.

3.8 Flexibility in Security

One of the important goals of our work was to ensure that in the process of providing foolproof security, we do not penalize applications that do not need the extent of security we provide. As a typical example, consider a web server using WFS to store its documents. It needs absolutely no privacy for its publicly accessible web data, and it would be a pity if WFS brings down the performance of the web server drastically by doing something that is totally unnecessary. This tradeoff between performance and security is a classical tradeoff in computer security and we strongly believe that filesystems do not have sufficient information to decide the optimal tradeoff. What secure filesystems must provide is a mechanism by which users are empowered to choose the ideal tradeoff they require for their files. Users have more knowledge on the semantics and importance of a file and they are the best candidates to decide what level of security their file should come under.

WFS presents a flexible interface by which the user can choose the security requirements of its file on a per-file basis. This security classification can also be dynamically changed by running a simple utility. We currently support three security levels: 1) no security, 2) data integrity, and 3) data integrity and privacy.

It can be seen in the performance section that follows that these three levels have widely varying performance and if the filesystem had been conservative and imposed strict security on all files the maximum performance realizable would be the worst performance reported. But now, we permit the user to choose the exact tradeoff between performance and security and we assert that this should be a prerequisite for any secure file system.

Providing flexibility to the user should not turn into a burden for the user, where the filesystem essentially requires the user to apply his mind on how to classify each of his files, when he actually does not want to. To facilitate this, we have default security policies in the filesystem that get assigned to files. A conservative solution would try to assign the maximum security to all files left unspecified by the user. Though this would be correct and safe, we sought to explore other approaches. At this point, WFS tries to infer the minimal security level that would be reasonable to assume for a file by default. It does this by taking hints from the *acl* of the file. For a file that has read permissions for *anyuser*, we for sure do not require privacy because the user does not care if anyone else looks at the data. Hence, we turn off privacy for such files. Similarly, for files with write access to *anyuser*, the user does not care if anyone writes into it, so enforcing integrity on the file data would be of little use, since modification by an intruder can be thought of as being similar to a write made by *anyuser*. So such files will have integrity turned off by default.

4 Implementation and Performance

The performance results we took are meant to be illustrative of the overhead incurred for the various levels of security that we provide. The wide disparity in performance among the various policies, as exemplified by the performance numbers, serve to argue why flexibility is a paramount facet of any secure filesystem. We took the measurements on three machines connected by a gigabit ethernet. The machines are P-III 550 MHz with 896 MB of RAM. We measured the read and write bandwidth for the three security policies: 1) *no security*, 2) *data integrity*, and 3) *data integrity and privacy*. We observe that performance drops markedly along the security axis, as we strive for better security. For example, in reads (figure 3), while the normal bandwidth realizable was 17.5 MBps, encryption and integrity computation brought it down to 4.23 MBps.

Security level	Bandwidth (MB/s)	CPU Utilization (%)
No security	13.50	25
Integrity	9.60	60
Integrity & privacy	4.23	84

Figure 2: Performance data for writes

Security level	Bandwidth (MB/s)	CPU Utilization (%)
No security	17.50	41
Integrity	9.92	15
Integrity & privacy	4.23	8

Figure 3: Performance data for reads

An encouraging observation we make is that the authentication mechanism during every read and write did not bring down the throughput greatly. The peak read bandwidth obtained in WFS without any security features is 18.5 MBps, and we find that doing per-read access control enforcement at the drive is not much of an overhead. This is in spite of the fact that we do not perform any caching of capability keys at the drive, which means that the drive is computing its capability key for every read or write operation. We hope that with caching of capabilities and a little performance tuning, we can achieve authentication and integrity of request arguments at the drive at a negligible cost.

We also report the CPU utilization observed for each of our tests. For writes (figure 2), the CPU utilization for *privacy and integrity* is much greater (84%) than *data integrity* (60 %), which is again higher than that for *no security* (25%). The high CPU utilization during enforcing *privacy and integrity* implies that a client machine cannot realize the benefit of multiple disks if it requires strong privacy. It might be interesting to see how this utilization varies across less strong encryption and decryption schemes. This also argues for our earlier conclusion that encryption and decryption at disks is a performance degrading choice, since it replicates the same CPU bottleneck at the disk too. One seemingly non-intuitive result is the pattern in which the CPU utilization varies for reads. This is because the decryption and integrity check are done in the RPC callback function which runs in the context of the *rpciod* daemon. Hence the CPU cycles get accounted in the name of *rpciod*, giving a false CPU utilization statistics to the test application.

5 Related Work

The issues in security for network attached storage devices have been explored in [Gobioff992] as part of the CMU NASD project. The NASD project however, considers security at the storage level rather than at the filesystem level and this we believe limits its scope. Storage devices in NASD for instance, are meant to be used by any filesystem and hence cannot exploit knowledge about a specific filesystem. A shortcoming of the project is that it does not look at efficient means of enforcing privacy. Although a privacy scheme was implemented, it incurred an unacceptably high performance penalty. In our project by providing the user the flexibility to choose the level of security on a per-file basis we permit a fine-grained trade-off between performance and security. Moreover, our strategy of inferring the minimal security level of a file by taking hints from the file's *acl* helps achieve this tradeoff without much botheration to the user.

The Security architecture applicable in a network attached storage environment has been discussed in [Gobioff97]. [Gobioff99] looks at some performance optimizations like pre-computing hash digests for data blocks to reduce per-request overhead at the disks. We plan to adopt a similar technique to speed up integrity calculation at the disks. The Kerberos [Steiner88] work and the secure file system [Mazieres99] describe issues and mechanisms to tackle authentication and security in file systems.

6 Conclusion

Network-attached storage holds promise due to the obvious performance implications. Its widespread adoption however will depend upon, among other things, the level of data security that can be provided at an acceptable level of performance.

We discuss that a straightforward approach using some standard protocol like Kerberos, and encrypting and authenticating every message will not perform well or scale. Our approach takes this into account. We offload most work away from the disks (e.g. disks don't do encryption/decryption as they simply store encrypted data). Involvement of central entities like the file manager is kept to a minimum by using capability mechanisms. We deal with authentication and integrity in a single step and our results show that the impact is minimal. As was mentioned before, one of our goals was to give the choice of security ultimately to the user. We have an interface wherein the user can choose the security level it wants on a per-file basis.

We have demonstrated an implementation of a performance-conscious and flexible security mechanism for WFS. Our work was illustrative and there are several directions in which we could proceed. Implementing mechanisms for anonymity of all communication in WFS will give stronger security guarantees. Finer grained choice of security levels will allow the user to make a better choice appropriate to her needs. For instance, security could be tailored by using different security algorithms, key sizes or number of rounds.

References

- [A-D01] A. C. Arpaci-Dusseau, R. H. Arpaci-Dusseau, J. Bent, B. Forney, F. I. Popovici, S. Muthukrishnan and O. Zaki. Manageable Storage Via Adaptation in WiND. IEEE International Symposium on Cluster computing and the Grid (CCGRID2001), May 2001.
- [Gibson97] G. Gibson, D. F. Nagle, K. Amiri, F. W. Chang, H. Gombioff, E. Riedel, D. Rochberg, and J. Zelenka. Filesystems for Network-Attached Secure Disks. CMU SCS Technical Report CMU-CS-97-118, 1997.
- [Gombioff97] H. Gombioff, G. A. Gibson and D. Tygar. Security for Network Attached Storage Devices. CMU SCS Technical Report CMU-CS-97-185, 1997.
- [Gombioff99] H. Gombioff, D. F. Nagle, and G. A. Gibson. Embedded Security for Network-Attached Storage. CMU SCS Technical Report CMU-CS-99-154, June 1999.
- [Gombioff992] H. Gombioff. Security for a High Performance Commodity Storage Subsystem. Carnegie Mellon Ph.D. Dissertation, CMU-CS-99-160, July 1999.
- [Mazieres99] D. Mazieres. Separating key management from file system security. 17th ACM symposium on Operating systems principles (SOSP 99) Operating systems review 34(5); 124-139, Dec. 1999
- [Steiner88] J. G. Steiner, B. C. Neuman, and J.I. Schiller. Kerberos: An Authentication Service for Open Network Systems. In Proceedings of the Winter 1988 Usenix Conference. February, 1988 (Version 4).