Integrating Security in FreeLoader Distributed File System

Abdullah Gharaibeh

abdullah@ece.ubc.ca

Electrical and Computer Engineering Department

University of British Columbia

***Abstract*—FreeLoader is a high performance read-dominant distributed file system that utilizes free disk space on desktop workstations. The system is designed for maximum scalability and high connectivity by taking into account the write-once-read-many property of specific use cases. While still in the prototyping stage, FreeLoader security has not been addressed until recently. In this work we develop a threat model for FreeLoader following the CIAA (Confidentiality, integrity, availability and authentication) threat modeling process. Based on our threat model, and the threats we decide are worth to mitigate, a set of security requirements are derived; moreover, we make the first attempt to design and implement a security model for FreeLoader.**

I. Introduction

FreeLoader [1] utilizes available storage space on commodity desktops while aggregating network bandwidth to achieve a fast, low cost storage solution that can be used as an alternative to existing expensive storage systems like Storage Area Networks (SAN). Hence, the main FreeLoader's use case is as a cache/scratch space for scientific data sets, where data sets are usually write-once-read-many; in addition, these data sets are shared within the same research group, which often have shared interest on specific data sets. Therefore, means for protecting the system should be provided to facilitate this sharing.

II. FreeLoader Overview

This section provides an overview of the current FreeLoader design as implemented in the latest release.

FreeLoader consists of three main components: A single manager, a set of benefactors and a set of clients that access the system. Figure 1 illustrates the design. Files are divided into fixed-size chunks (typically 1MB) distributed among the benefactors where they are stored as regular files. The manager maintains all file system metadata such as available storage space, file system namespace, file attributes, the mapping from files to chunks and the mapping from chunks to benefactors. From the client side, FreeLoader provides two implementations: A command line client and a traditional file system interface using FUSE (File system User SpacE) Linux kernel module [8].

A typical write operation is performed as follows:

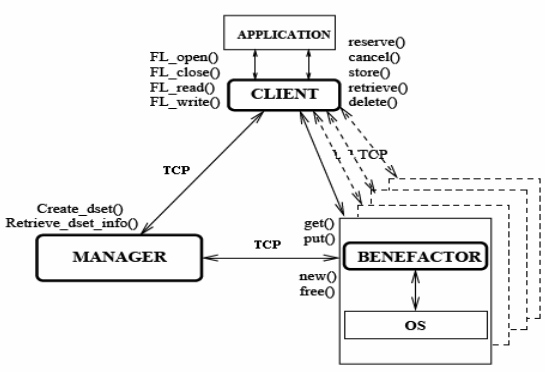

Figure 1: FreeLoader Modules

1. The client asks the manager for a file handle for the new file. A file handle is a 32bit integer identifier unique per file.
2. The manager increments its internal file handle counter and replies with a new identifier.
3. The client asks the manager for a set of benefactors that will provide the disk space for the new file.
4. The manager replies with a list of benefactors that will hold the new file's chunks. This list is named a chunkmap.
5. The client divides the file into chunks and pushes them to the benefactors according to the chunkmap sent by the manager in the previous step.
6. Finally, the client commits the chunkmap back to the manager asserting the write operation. Only in this step the actual storage space status is updated in the manager. Moreover, the manager doesn't have any state information to relate this chunkmap with the one previously sent in step 4.

A typical read operation is performed as follows:

1. The client asks the manager which benefactors hold the chunks of a specific file.
2. The manager responds with the set of benefactors that hold the file's chunks.
3. The client pulls the data from the benefactors directly.

All requests and replies are performed using TCP as a transport protocol. Also, note that the client never read or write data through the manager. One more important point to

mention is that the number of available benefactors has a direct effect on FreeLoader performance. Having more benefactors in the system gives more parallelism, hence increased performance and visa versa. Therefore, when creating a new chunkmap for a write operation (step 4), the algorithm that maps chunks to benefactors will try to balance the load across the benefactors.

III. Threat Model

In this section, we identify a set of possible attacks or threats that can be mounted against FreeLoader distributed file system following the CIAA (confidentiality, integrity, availability and authentication) approach. As mentioned before, we will base our analysis on the current FreeLoader prototype implementation. While the focus of this section is on threat modelling, we do briefly provide references to possible protection techniques.

A. Confidentiality Attacks

Confidentiality attacks are passive attacks that expose confidential data to the view of unauthorized readers [9]. These kinds of attacks don't alter the system content, but affect the security level and distribution of the content. FreeLoader is a distributed system where different components of the system communicate using a shared network; hence, the main confidentiality attack is sniffing the network traffic. FreeLoader traffic can be divided into two main categories: control traffic generated between the manager and both the clients and the benefactors, and data traffic generated between the clients and the benefactors.

In addition to traffic sniffing, any user has access to the donated space in a benefactor machine (which is typically a designated folder), has full access to the chunks stored in that space.

The obvious solution to confidentiality attacks is to use cryptography. For example, instead of using pure TCP connections as a transport protocol, Secure Socket Layer (SSL) [RFC 2246] can be employed to provide secure communication across the three system components. Moreover, the client can encrypt the chunks before pushing them to the benefactors so that clients with the correct key can make it comprehensible. Old encryption algorithms like 3-DES are very expensive; however, newer algorithms like AES allow much better performance and are easier to implement efficiently [10].

B. Integrity Attacks

Integrity attacks attempt to modify information in the system without proper authorization.

A possible integrity attack, which is common to all distributed systems, is a man-in-the-middle attack in which the attacker makes independent connections between any two components in the system and relay messages between them. For example, an attacker can intercept the messages sent by a client to the manager and change its parameters.

Also, as mentioned in the previous section, a user has access to the donated space in a benefactor machine, has full access to the chunks stored in that machine, which means that a malicious user can manipulate with the chunks.

Message Integrity Codes (MIC), which is a short cryptographic checksum, can be used to achieve message integrity. Also, the manager can store per-chunk checksums to increase chunk integrity while stored in the benefactor.

C. Availability Attacks

Availability attacks attempt to make system services unavailable to legitimate users for a period of time.

As mentioned in the overview section, a write operation starts with the client asking the manager for a new file identifier. An attacker may create many fake files to exhaust the file identifier space. Moreover, an attacker can exhaust the storage space by continuously reserving and committing back the chunkmap (steps 3 and 6 in the write operation) without even writing a single byte to the benefactors which means that this attack can be done in no time. Also, a malicious client can hinder the system performance by exhausting the disk space of a specific set of benefactors, thus reducing parallelism. This can be done by asking the manager for a chunkmap (step 4 in the write operation) and repeatedly performing steps 1 and 6 (asking for a new file id and committing the same chunkmap) until all disk space donated by this set of benefactors is exhausted.

The first attack is difficult to defend against because it takes the form of a legitimate content with bad intent. The second one can be eliminated by requiring the client to present a chunk hash signed by the host benefactor for each committed chunk; further, the last attack can be minimized by enforcing a predefined storage quota for each client.

D. Authentication Attacks

Authentication attacks occur when an attacker masquerades as a legitimate end-user. FreeLoader doesn't implement any sort of authentication until now. However, we will discuss few common authentication attacks that should be considered when introducing authentication to the system.

The first traditional attack is brute force attack where the attacker tries every possible combination of characters to meet the password or the key. Dictionary attacks fall in this category also. Another possible attack is replay attacks in which messages are recorded and then retransmitted to trick the user into unauthorized operations.

Selecting the appropriate key size or password strength is vital to increase resistance against brute-force attacks, while replay attacks can be avoided by using timestamps.

Kerberos [11] is a widely deployed approach for authentication in distributed environments. Global Security Services API (GSS-API) [RFC 2743 and RFC 2744] is yet another approach that provides a mechanism agnostic interface for security services. The latter will be discussed in more detail in the next section.

IV. Assumptions, Requirements and Mechanisms

FreeLoader is an ongoing project. First prototypes of the system focused on feasibility and performance. Next phases of the project will focus on robustness and security. In this

section we present our assumptions and requirements about the deployment environment; moreover, we provide an overview to the main mechanism used to develop our secure protocol.

A. Assumptions

Before discussing the requirements, we first introduce our design assumptions and considerations: First, we assume that the manager is a trusted machine, managed by a trusted administrator. Second, FreeLoader is not responsible for data confidentiality, i.e. data will be stored unencrypted in the benefactors; however, the higher level application can always encrypt the data before pushing it to the benefactors. These two assumptions are made (and will most probably preserved in the next versions of the model) as a trade off to simplicity and performance. Third, since Freeloader is open source, we assume that an adversary can, as a client or a benefactor, modify the code for malicious purposes; further, an adversary can passively or actively sniff network traffic to mount spoofing attacks.

B. Design Requirements

The following is a set of requirements based on which the model is developed and the mechanism is chosen:

1. Acceptable performance degradation: Security overhead must be minimized in order to preserve Freeloader's main design goal, performance.
2. Scalability: Introducing security to the system shouldn't impact its scalability. This means that Freeloader should provide acceptable performance while achieving security in large scale deployments.
3. Authentication and Authorization: clients should be authenticated and authorized for accessing the system's resources.
4. Data integrity: The client should always be able to assure that all chunks stored in the benefactors are consistent and correct.
5. Accountability: The solution should be able to determine who is responsible for a corrupted file: the client or the benefactor. However, accountability among clients is not addressed in this work such as the responsibility for last file modification.
6. Usability: Access to FreeLoader shouldn't significantly change. As mentioned before, FreeLoader provides a convenient client interface using FUSE Linux module which support traditional file system API.
7. Easy integration with the Grid environment: We want to be able to run FreeLoader as a service in Grid deployments.

C. Mechanisms

The Generic Security Services Application Program Interface (GSS-API) provides access to security services in a generic fashion. GSS-API defines services and primitives at a level independent of the underlying mechanism and programming language environment. Kerberos [11] is one mechanism that can be used to implement the GSS-API among other mechanisms. The GNU Generic Security Services Library manual [12] defines four major steps to use the API (Figure 2 summarizes the protocol):

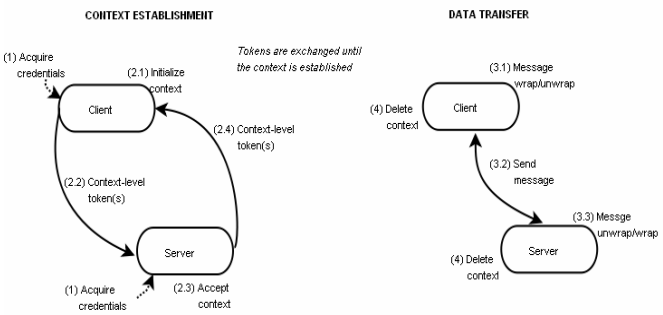

Figure 2: GSS-API Protocol

1. The application acquires a set of credentials with which it may prove its identity to other processes such as X.509 certificate or a Kerberos ticket (Depending on the low level mechanism used to implement the interface).
2. Communication between two applications starts with establishing a security context using their credentials. The security context is established by exchanging opaque messages called tokens that hide the implementation details from the application. These tokens can be exchanged over an insecure channel as the lower level mechanism should guarantee message security. At the end of this step, authentication is accomplished and a security context is established (which is basically a shared symmetric session key). Authentication can be either one-way or two-way (mutual) depending on the parameters specified at the beginning of this step.
3. Once the context is established, both applications can exchange messages with different levels of protection: integrity and data authentication and confidentiality. The API provides two main functions for message protection: gss_wrap which applies the required quality of protection to the application data, and gss_unwrap that converts the message back to application data.
4. At the end of the communication session, both applications delete the security context.

The advantage of using the GSS-API instead of a native implementation of a specific mechanism is that it will be easier to port an application across different mechanisms; even across different versions or implementations of the same mechanism. For example, Kerberos V has several not compatible implementations. However, the GSS-API only standardizes authentication, and not authorization.

Another important advantage is that the Grid Security Infrastructure (GSI) [13], which is used by the Globus toolkit [14] (A toolkit to build computing grids), adopts the GSS-API as a standard interface to access its security services; hence, it

will be easy to integrate FreeLoader with Globus toolkit as a new storage service.

The GSI implementation of the GSS-API is based upon SSL/TLS protocol and X.509 certificates. In this work, we use this implementation to provide two security requirements: user authentication and message protection; in addition, we assume that all parties have certificates issued by a common trusted certificate authority.

V. Security Model

In this section, we describe our modified protocol to accomplish the previously declared set of requirements. The protocol is described in the context of a write operation (Figure 3) and is detailed below:

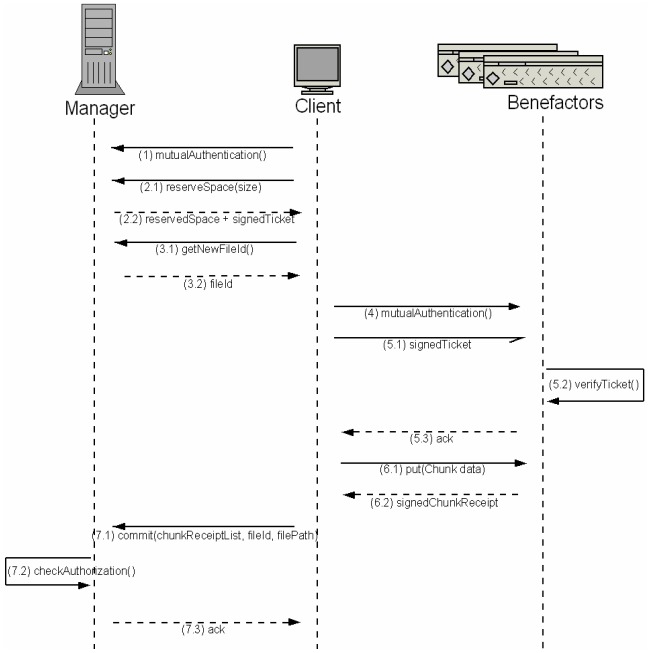

Figure 3: Write protocol

1. The write operation starts with mutual authentication between the client and the manager. At the end of this step, both parties are certain that they know each others' identity and public key; moreover, a security context is established between them. As illustrated previously in the GSS-API section, all messages between the client and the manager in the next steps will be protected within this context (Basically signed and optionally encrypted), hence achieving communication integrity and confidentiality.
2. The client sends a reserve space request to the manager asking for free space from the benefactor pool. Note that this space is not related to a specific file; rather it is a generic space that can be used by the client to store chunks of any file. Typically a client would reserve space once per group of files. As a response, the manager replies with a set of benefactors, which the client can push its data to; along side a signed authorization ticket. Mainly, an authorization ticket contains the client's name and a timestamp that limits the validity period of the ticket to a predefined amount of time.
3. Next, the client asks the manager for a file id for the new file. The manager replies with a new file id exactly as described previously in section II.
4. The client mutually authenticates with the benefactors that are going to host its data. As stated in the first step, a security context is also established with each benefactor and all subsequent messages are optionally protected within this context.
5. Once successfully authenticated, the client sends the authorization ticket previously obtained from the manager to all benefactors selected to store the file. Each benefactor will verify that the ticket is signed by the manager and that it is not expired; further, the client name contained in the ticket is checked against the identity of the currently authenticated client. Once successfully done, a positive acknowledgment will be sent by the benefactors to the client in order to start sending the data; otherwise, the write operation fails.
6. In this step, the client divides the data into fixed-size chunks (which is currently set to 1MB) and pushes the chunks to the benefactors. Note that the chunks are also sent within the previously established security context to achieve different levels of protection. For each received chunk, the benefactor will send back a signed chunk receipt to the client. A chunk receipt mainly contains the benefactor's name, the chunk's hash, and the corresponding file id.
7. In the end, and after pushing all the file's chunks, the client composes a chunkmap for the file. A chunkmap identifies the responsible benefactor and chunk receipt for each chunk. The chunkmap, together with the file id and the intended file path, are sent to the manager in order to commit the write operation. The manager consults its internal policy decision module to check whether the client has the permission to write to the specified path. If authorized, the chunkmap is checked for consistency by verifying the signatures of each chunk receipt against the specified responsible benefactor, and that all chunk receipts belong to the same file id. Once successfully verified, the chunkmap is stored as part of the file's metadata. Otherwise, the manager will reject the commit request.

The read protocol differs from the abovementioned write protocol in three places: First, instead of steps 2 and 3 in the write protocol, the client will send a read request for a specific file path to the manager, and the manager replies with the file's chunkmap signed by its private key. Second, in step 5, the client presents the signed chunkmap to the benefactors instead of the manager's authorization ticket. Third, in step 6 the data flow is from the benefactors to the client and there will be no commit in the end.

VI. EVALUATION

A. Security Analysis

The new protocol provides authentication, authorization, message protection, data integrity and accountability.

Authentication is achieved by requiring any two communicating entities to mutually authenticate before starting a communication session. The details of the authentication protocol depend on the specific mechanism used to implement the GSS-API interface. In our case, we have chosen the GSI implementation of the GSS-API interface, and the details of the authentication protocol is detailed in [16].

Authorization in our system is two fold: First, whether the client has the right to modify (or read) a file's metadata maintained by the manager. Second, whether the client has the right to write (or read) chunks to the benefactors. The first part is achieved by assuming that the manager will consult a policy decision module to check for a request's authority. Our protocol doesn't assume any specific implementation for this module which can be represented as an ACL or an external authorization service such as CAS (Community Authorization Service) [17]. The second authorization part is achieved by requiring the client to present an authorization ticket signed by the manager (or a signed chunkmap in case of a read operation) to the intended benefactor; thus, and as described in the previous section, the benefactor can verify the client's authority by checking the identity specified in the ticket against the identity of the requesting client and that the ticket is not expired.

One result of the authentication phase is the establishment of a security context in which all messages between the communicating parties are protected (signed and optionally encrypted), hence achieving message integrity. In the same vein, we preserve data integrity by having the manager to maintain per chunk fingerprint hash which is received as part of the chunk receipt in the commit phase (step 7 in the write protocol). Therefore, a client can check for data integrity by hashing the read chunk and comparing it with the hash maintained by the manager.

Accountability for a corrupted chunk is accomplished by employing chunk receipts. As detailed in the protocol, a chunk receipt represents a proof that the client has pushed the specified chunk to the benefactor; moreover, since the receipt is signed by the host benefactor, the client can't modify the receipt in case he wants to accuse the benefactor later on for bad chunk preservation; therefore, whenever a chunk hash doesn't match the one maintained by the manager, there is one party to blame, the benefactor.

On the other hand, our protocol doesn't provide explicit countermeasures against availability attacks where we depend on two underdevelopment FreeLoader features that can minimize the effect of these attacks. First, a synchronization protocol named garbage collection between the manager and the benefactors. This protocol synchronizes metadata status stored in the manager with the actual storage usage in the benefactors. Garbage collection is assumed to run once every specific configurable amount of time; hence, any fake commits done by a malicious client are reverted once garbage collection is done. Second, client quota in which each client is allocated a predefined maximum storage space; thus, a malicious client with intent to reduce parallelism (by always streaming to a specific set of benefactors to exhaust their donated space) is limited with his quota. Moreover, this attack has a very limited effect in large scale deployments where hundreds of benefactors exist.

One last point to mention is that accessing FreeLoader is not significantly changed. The manager, the clients and the benefactors should all have valid certificates from a common trusted CA. In a small deployment, an administrator can install a local certificate authority for this purpose such as SimpleCA [15].

B. Performance Evaluation

We evaluate our prototype using a cluster of nine nodes. Each node is a Dell PowerEdge 1950 machine equipped with 1Gbps Ethernet card, a 2.33GHz Intel Xeon Quad-core CPU with a 1333MHz FSB (Front Side Bus) and 4GB of memory. For all configurations we report averages and standard deviations (using error bars in plots) over 15 runs.

Our security protocol provides three main features: Authentication, integrity and accountability. As mentioned before, authentication is achieved by steps 1 and 4 in the write protocol. Channel integrity is achieve by sending all network traffic within the established security context (which basically uses MIC (Message Integrity Code)), while file integrity is accomplished by storing chunk hashes in the manager. Finally, accountability is achieved using chunk receipts in steps 6 and 7.

In this evaluation, we measure the throughput of a write operation while incrementally enabling each of the abovementioned features in order to quantify their performance impact; consequently, we define three different security configurations: (i) Authentication (ii) Authentication and Integrity (iii) Authentication, Integrity and Accountability. Furthermore, we note that FreeLoader's implementation decouples the application write I/O from the actual file transfer over the network that stores the file on the benefactors [18]; therefore, two performance metrics are defined to compare the various alternatives of our security protocol. First, the observed application bandwidth is the write bandwidth observed by the application in which the file size is divided by the time interval between the application-level open() and close() system calls. Second, the achieved storage bandwidth uses the time interval between file open() and until the file is stored safely in the benefactors (i.e., all remote I/O operations have completed).

Figure 4 presents the observed and achieved storage throughput for the three configurations compared to the original FreeLoader protocol. The figure shows a very bad performance where the throughput degradation is in the range of one order of magnitude. This is due to the very expensive mutual authentication operation, especially between the client and the benefactors, which incorporates a great number of public key cryptography operations. For example, a 1 GB file with a chunk size of 1MB requires 1K chunks to be pushed to

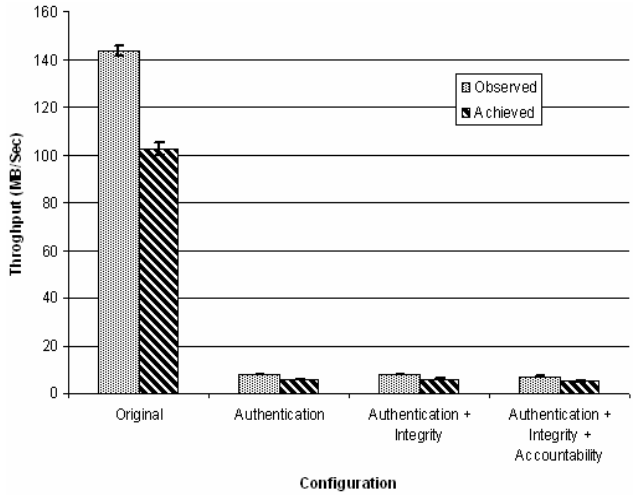

Figure 4: Average observed and achieved write throughput for one client and four benefactors.

the benefactors, which mean 1K mutual authentication operations between the client and the benefactors. Additionally, the experiment shows no differences among the three security configurations. This is also attributed to the mutual authentication overhead which dominates the protocol's overall operating cost.

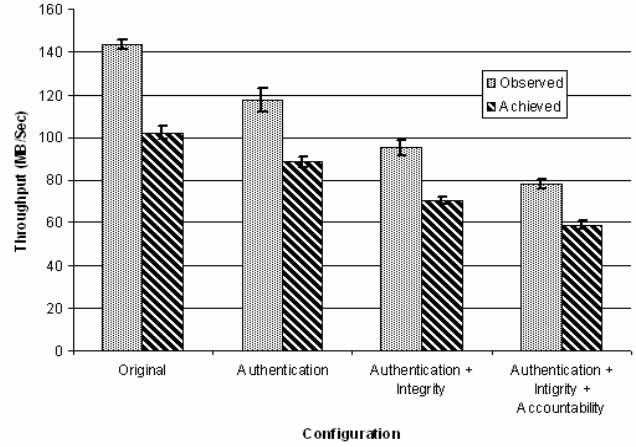

Figure 5: Average and achieved write throughput for one client and four benefactors while caching the context.

To solve this problem, we cache the context established between any two entities after the first mutual authentication. This cache is maintained for a configurable period of time before expiration. This period can be set to an absolute value (half an hour for example) or for the duration of streaming a single file. This optimization enhances greatly the performance of the system as it eliminates a large number of redundant authentications while maintaining the protocol's characteristics. Figure 5 presents the effect of caching on the observed and achieved write throughput.

C. *Protocol Scalability*

To assess the scalability of our protocol, we measure the throughput of FreeLoader with increasing number of benefactors (from one to eight benefactors). Figures 6 and 7 show the observed and achieved throughput respectively. Note that after adding the third benefactor, the original FreeLoader protocol saturates as we are limited with the network card maximum speed (1Gbps). The figures show that the throughput of our protocol ramp-up successfully with increasing number of benefactors as it achieves less than 10% degradation in performance for the three features and around 3% degradation for the authentication feature only.

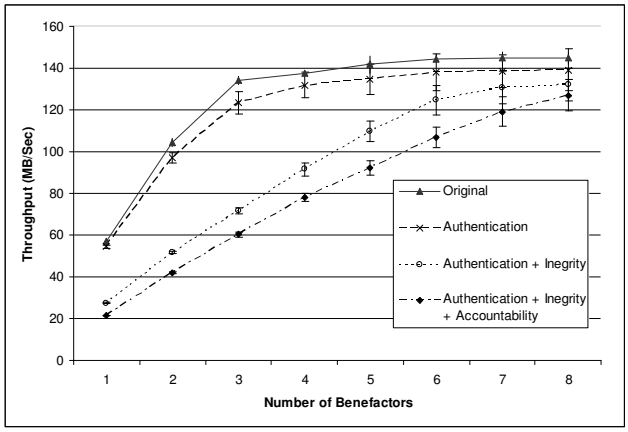

Figure 6: Average observed write throughput with increasing number of benefactors.

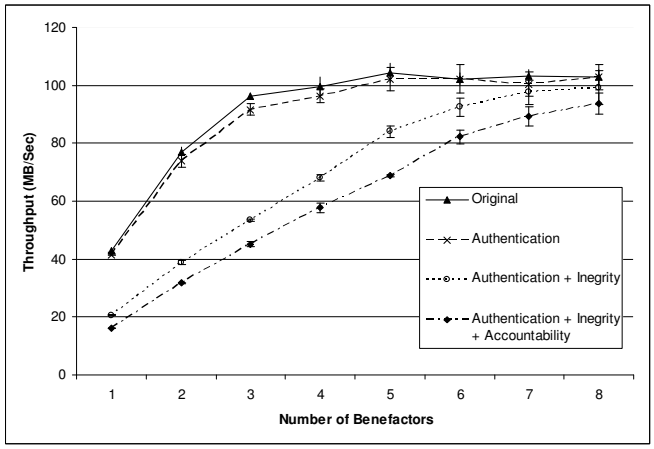

Figure 7: Average achieved write throughput with increasing number of benefactors.

D. *Effect of file size*

Figure 8 shows the effect of file size on the throughput. Note that when adding the accountability feature, the throughput ramps up much slowly because of the need to produce chunk receipts by the benefactors which involves

hashing the chunks and encrypting the hash, among other parameters, using the private key of the benefactor.

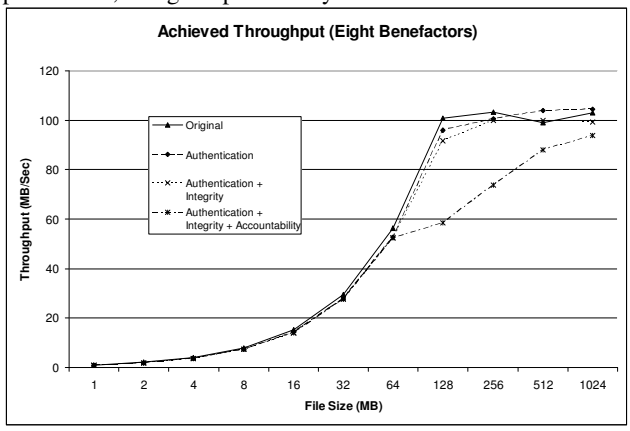

Figure 8: Average achieved write throughput with increasing file size.

VII. Related Work

Networked File System (NFS) [5] is one of the most popular distributed file systems. The early versions of NFS implement a very weak security model. NFS relies on the IP address to authenticate the client hosts making it vulnerable to address forgery; moreover, the system relies on the client machine to authenticate the user making it vulnerable to any user who has compromised the client machine. The new version of the protocol, NFSv4, dictates many new changes to the previous versions. For example, it mandates the use of Kerberos V5 support for user authentication.

Andrew File System (AFS) [6] was developed to provide a scalable file system. AFS uses Kerberos for authentication and implements access control lists on directories for users and groups.

Self-certifying File System (SFS) [7] is a global and decentralized distributed file system. SFS provides transparent encryption of communication and authentication. It is designed to operate securely between separate administrative domains.

VIII. Conclusion

This paper presents a threat model and a design and implementation of a secure protocol for FreeLoader distributed file system. We have presented a security analysis for the protocol and performance evaluation that shows reasonable performance degradation in small deployments, and close to original performance in larger deployments.

References

[1] Vazhkudai, S.S.; Xiaosong Ma; Freeh, V.W.; Strickland, J.W.; Tammineedi, N.; Scott, S.L., "FreeLoader: Scavenging Desktop Storage Resources for Scientific Data," Supercomputing, 2005. Proceedings of the ACM/IEEE SC 2005 Conference, vol., no., pp. 56-56, 12-18 Nov. 2005.

[2] Hasan, R., Myagmar, S., Lee, A. J., and Yurcik, W. 2005. Toward a threat model for storage systems. In Proceedings of the 2005 ACM Workshop on Storage Security and Survivability (Fairfax, VA, USA, November 11 - 11, 2005). StorageSS '05. ACM, New York, NY, 94-102.

[3] S. Myagmar, A. J. Lee, and W. Yurcik. Threat Modeling as a Basis for Security Requirements (SREIS). In Symposium on Requirements Engineering for Information Security, 2005.

[4] F. Swiderski and W. Snyder. Threat Modeling. Microsoft Press, 2004.

[5] B. Pawlowski, S. Shepler, C. Beame, B. Callaghan, M. Eisler, D. Noveck, D. Robinson, and R. Thurlow. The NFS version 4 protocol. SANE 2000, May 2000.

[6] J. Howard. An overview of the Andrew file system. In Proceedings of the USENIX Winter Technical Conference, Dallas, TX, February 1998.

[7] M. Kaminsky, G. Savvides, D. Mazie´res, and M. F. Kaashoek. Decentralized user authentication in a global file system. In Proceedings of the 19th ACM Symposium on Operating Systems Principles, 2003.

[8] FUSE, Filesystem in Userspace, http://fuse.sourceforge.net/. 2008.

[9] Ballardie, T. and J. Crowcroft, "Multicast-Specific Security Threats and Counter-Measures," Symposium on Network and Distributed System Security, February 1995.

[10] Tucek, J.; Stanton, P.; Haubert, E.; Hasan, R.; Brumbaugh, L.; Yurcik, W., "Trade-offs in protecting storage: a meta-data comparison of cryptographic, backup/versioning, immutable/tamper-proof, and redundant storage solutions," Mass Storage Systems and Technologies, 2005. Proceedings. 22nd IEEE / 13th NASA Goddard Conference, pp. 329-340, 11-14 April 2005.

[11] J. G. Steiner, C. Neuman, and J. I. Schiller, "Kerberos: An authentication service for open network systems," in Proc. USENIX Winter Conf., Feb. 1988, pp. 191–202.

[12] GNU Generic Security Service Library, http://www.gnu.org/software/gss/manual/index.html. 2008.

[13] Butler, R.; Welch, V.; Engert, D.; Foster, I.; Tuecke, S.; Volmer, J.; Kesselman, C., "A national-scale authentication infrastructure," Computer, vol.33, no.12, pp. 60-66, Dec 2000.

[14] I. Foster and C. Kesselman. Globus: A Metacomputing Infrastructure Toolkit. International Journal of Supercomputer Applications, 11(4):115-128, 1997.

[15] SimpleCA, A simple certificate authority, http://www.vpnc.org/SimpleCA/. 2008.

[16] http://www-unix.globus.org/toolkit/security/. 2008.

[17] Pearlman, L., Welch, V., Foster, I., Kesselman, C., and Tuecke, S. 2002. A Community Authorization Service for Group Collaboration. In Proceedings of the 3rd

international Workshop on Policies for Distributed Systems and Networks (Policy'02) (June 05 - 07, 2002). POLICY. IEEE Computer Society, Washington, DC, 50.

[18] stdchk: A Checkpoint Storage System for Desktop Grid Computing, Samer Al-Kiswany, Matei Ripeanu, Sudharshan Vazhkudai, Abdullah Gharaibeh, International Conference on Distributed Computing Systems (ICDCS 08), Beijing, China, June 17-20, 2008.