Hardening Access Control and Data Protection in GFS-like File Systems

James Kelley[1], Roberto Tamassia[1], and Nikos Triandopoulos[2,3]

[1] Brown University, Providence, Rhode Island, USA
[2] RSA Laboratories, Cambridge, Massachusetts, USA
[3] Boston University, Boston, Massachusetts, USA

Abstract. The Google File System (GFS) is a highly distributed, fault-tolerant file system designed for large files and high throughput batch processing. We consider the first complete security analysis of GFS systems. We formalize desirable security properties with respect to the successful enforcement of access control mechanisms and data confidentiality by considering a threat model that is much stronger then in previous works. We propose extensions to the GFS protocols that satisfy these properties, and provide a comprehensive analysis of the extensions, both analytically and experimentally. In a proof-of-concept implementation, we demonstrate the practicality of the extensions by showing that they incur only a 12% slowdown while offering higher-assurance guarantees.

1 Introduction

As more companies adopt the cloud computing framework, an increasing amount of sensitive and mission-critical data will be placed in the cloud. Thus, it is necessary to develop and deploy strong security controls in the underlying cloud framework to protect this data. This necessity is underscored by the work several researchers have done demonstrating various weaknesses in current commercial cloud offerings (e.g., [23,24]).

The Google File System (GFS) is the file system developed in-house by Google to support their storage needs [12]. GFS is a distributed file system utilizing a single server for managing file metadata and (up to) legions of data servers for storing file data. A file is split into blocks (typically tens or hundreds of megabytes in size) which are spread out over the data servers. The servers are assumed to be running on commodity hardware, and the system is meant to scale to thousands of machines. So, machine failures are assumed to be a frequent, and entirely normal, occurrence.

The paradigm ushered in by GFS has since seen deployment in cloud computing infrastructures—notably, HDFS in Hadoop [14]—as the underlying storage mechanism for the large quantities of data. The architecture of GFS lends itself to supporting a MapReduce computing framework, and, indeed, they were developed together. As such, GFS sees use in large data centers for performing computations on enormous data sets (e.g., tens and hundreds of terabytes or more) with already great efficiency and several efforts to further improve its

S. Foresti, M. Yung, and F. Martinelli (Eds.): ESORICS 2012, LNCS 7459, pp. 19–36, 2012.

performance (e.g., [5,8,15]). The usefulness of the MapReduce framework has made the use of data warehouses highly desirable, but potentially costly—due to the large setup and maintenance costs. Given this, companies are increasingly outsourcing these MapReduce needs to the cloud (such as Amazon's EC2 and Elastic MapReduce services). GFS has thus inspired several copy-cat cloud-centric implementations, including the Hadoop File System, CloudStore, and TPlatform [4,6,22], all of these falling under the banner of *GFS-like file systems*.

1.1 Security Issues and Challenges

In a GFS-like system, files are broken up into blocks which are replicated and distributed across multiple *data servers* to achieve fault-tolerance. The system is managed by a central *metadata server* that handles all metadata operations and tracks the placement of blocks, seeking to balance the load across all servers and maintain enough replicas of blocks. Figure 1 shows the basic architecture in GFS-like systems. For example, to create a file, a user contacts the metadata server who records the metadata and replies with a list of data servers; then, the user contacts the data servers to upload the data. Also, the metadata server manages access control information for each file, but here things become problematic.

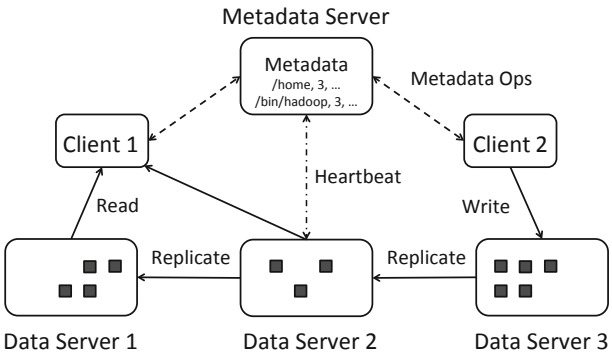

Fig. 1. Basic architecture of a GFS-like system

The design of GFS assumes a benevolent environment: the users are assumed to behave well and not interfere with each other. For example, in the Hadoop File System (HDFS) (and others, see [6,22]), by default the data servers service any request from any user. An assumption of total user benevolence has little justification in the real-world. Thus, it is necessary to integrate security controls into GFS-like file systems to make attacks by malicious users much more difficult.

As an example attack, when accessing a file, a user will contact the metadata server M to learn the location of the file blocks and then contact the individual servers to read the blocks. Since access control checks are performed only at M, they can be bypassed by contacting data servers directly. In a more sophisticated attack, the attacker could register their own machine as a data server. GFS does not authenticate any data server registrations, so any machine may complete the

registration protocol. Once registered, the attacker can simply wait to be given users' data blocks. Ideally, a GFS-like system should achieve a holistic security posture guaranteeing that no attacker can compromise the secure functionality of the system with respect to the integrity and confidentiality of its stored files, nor compromise the system's access control mechanisms.

A key step in adding robust security to these file systems is to extend responsibility of access control checking to the data servers themselves. In addition to stronger access control enforcement, all access control checks, and data structures holding the access control information, must be very efficient. GFS-like file systems are performance oriented, with data intensive applications and hundreds to thousands of parallel tasks. A related concern is that one must also endeavor to avoid putting an undue burden on system administrators. Complex security controls are much more likely to be ignored if they make administration much more difficult and/or negatively impact the job-efficiency of normal users.

The initial paper describing GFS states that no security was built into the system, other than rudimentary checks at the metadata server: no access control checks at servers and no protection of data in flight [12]. Yahoo! has instrumented the Hadoop File System (HDFS) with additional access controls to address some of the security concerns of its users [29]. Their architecture uses Kerberos for user authentication and message integrity, and uses a token-based access control scheme (similar to Kerberos tickets). As with GFS, there are no protections for network traffic and no method to prevent unauthorized servers from registering as data servers. An outline of many more attacks against the Hadoop MapReduce framework, of which HDFS is a part, is given in [1]. Some work was done in [7,28] to harden Hadoop against a worst-case-scenario adversary, putting HDFS on top of the least-authority file system Tahoe, but this resulted in rather severe performance penalties. Moreover, the system does not protect against an attacker bypassing the metadata server to read a block directly from a data server.

1.2 Our Contributions and Approach

In this work, we present the first formal definitions of security for a GFS-like file system. The adversary considered in the work is also a great deal stronger than in previous work and the first to be formally defined. The work by Yahoo! in [29] had an adversarial model, but the adversary was given in terms of its abilities relative to system privileges (e.g., could not be root) rather than any general abilities. Our main result is in proving that our modified GFS architecture is secure, given our formal definition of security, against our more powerful adversary.

From a practical perspective, the contributions of this work are several. Yahoo!'s work on securing HDFS relied on integrating it with Kerberos to provide message authentication and integrity, but not confidentiality. In this work message integrity, authentication, and confidentiality are built into the GFS protocols themselves, without adding a central key distribution center. Another contribution is the integration of stronger, pervasive access control enforcement. It is worth noting that Yahoo!'s work also has pervasive access control enforcement via tokens; however, forging the tokens becomes trivial if the adversary is

permitted to have root access on their machine. There is no such restriction on our adversary. Finally, we deployed a proof-of-concept implementation of these protocols using the Hadoop File System as our starting point. We then performed several experiments to show the practicality of our architecture, showing an overall slowdown of 12%: a reasonable price for stronger, provable security.

In our approach, the metadata server has an asymmetric key pair, with the public key distributed to all data servers. The system administrator also has an asymmetric key-pair which is used to authenticate the start up of each data server; the public half is known to the metadata server. Next, the data servers are brought online. Each data server generates two random keys to be shared with the metadata server for authenticating and encrypting subsequent messages. The keys are encrypted using the metadata server's public key, grouped with some registration information, and the bundle is signed by the administrator. Upon receiving the message, the metadata server verifies the signature, then decrypts and saves the keys while recording the registration information and then replies to the server with some start-up information. All future messages between these two servers are authenticated via a MAC using one of the keys sent during registration. The server periodically sends a heartbeat to the metadata server to attest to its liveness. Once the data servers have started and registered, the cluster can begin to service clients. When a client first starts to use the cluster, it must create a session with the metadata server. The client creates two random session keys and sends them to the metadata server encrypted with public key of the server. The metadata server stores the keys and replies with an acknowledgment of the registration. All further communication will be authenticated by using a MAC with one of the keys associated with the client.

Access control for the files is maintained through tokens: a token (similar to a Kerberos ticket) is issued by the metadata server to a client to read/write a file block. Each token is specific to the server that is holding the block. When returning the token and block location to the client, the metadata server also sends a pair of random secret keys to be used in authenticating messages to/from the data server and encrypting any file data transmitted. These keys are only valid for the duration of the client's request to the data server. The metadata server also prepares and returns a ciphertext to be passed to the data server by the client which contains the information needed for the data server to interact with the client. When a client is finished with their session, they inform the metadata server, who then deletes the session keys. If the client crashes, the session information expires after a given interval of inactivity.

The rest of this paper is organized as follows. Section 2 presents our formal security definitions. In Section 3, we describe our security-enhanced GFS protocols and their asymptotic efficiency. Section 4 provides the security analysis. Section 5 reports on an experimental evaluation on our proof-of-concept implementation in Hadoop, comparing it to the insecure default Hadoop. Section 6 discusses related work and Section 7 presents concluding remarks.

2 Definitions and Model

The GFS protocols have a fixed set of roles that can be assumed by principals: (*a*) the metadata server M, (*b*) a data server D, (*c*) a client C, (*d*) or the system administrator SA. A principal not conforming to exactly one of these roles when executing a protocol will produce an invalid execution of the protocol and all messages sent in the protocol will be ignored by the other principal.

In the following security definitions, we consider a polynomially bounded adversary: bounded in both time and space. The adversary is also subject to a few more restrictions, detailed below. We will later prove that, subject to standard cryptographic assumptions and the definitions in this section, the adversary has only a negligible probability of successfully violating the security guarantees.

At a high level, GFS is a collection of protocols implementing a network-facing file system API. We will define this API to be synonymous with the collection of protocols and it will be this collection that we will secure.

Definition 1. *The* GFS API *is the suite of protocols covering all client–server and server–server communication in the GFS file system.*

As a first step in setting up a GFS-like system, the system administrator SA must determine which machines are to constitute the cluster. The metadata server M is chosen as part of the configuration of the cluster by the administrator, so we assume M to be given and fixed. The first property we wish the cluster to have is that only those servers chosen by SA to be data servers can become part of the cluster. That is, we want to guarantee that SA has full control over which servers may be data servers. Moreover, we want to ensure that, with overwhelming probability, a data server can only possess the data blocks that have been assigned to it by M.

Definition 2. *A data server D exporting the GFS API is* authorized*, if D has successfully completed the registration protocol with M at the behest of the system administrator SA. Moreover, we say that a data server D is* authorized with respect to a block b *if D is authorized and M chooses D as a location for b.*

In a similar vein to the above definition, we next define correct behavior for a client. A client's interactions with the cluster revolves around reading and writing blocks. Naturally, we would like to restrict clients to only accessing blocks for which they are "authorized" (defined below). We define *accessing a block* to be either a read or a write operation on the block.

Definition 3. *A client C of the GFS API is* authorized *to access a block b, if that C is permitted, by the access control policy P associated with b, to access b.*

The policy P could be any type of access control policy (e.g., capability-based, mandatory access control, etc.). In GFS-like systems, the enforcement of an access control policy is performed solely at the metadata server with data servers blithely servicing any arriving request. This work provides a secure means to

extend policy enforcement to the data servers via unforgeable (except with negligible probability) access tokens. Note, however, that the initial policy check is still only performed at M.

As a final, basic term we define the notion of "completing a protocol." A protocol is considered complete if each principal believes that the other is authorized and if all messages are received and verify correctly. For example, if a client tries to read a block from a server, the client has completed the protocol when they receive the block from the server and the message containing the block passes all security checks ("verifies"). The server has completed the protocol upon sending the data block and having it verify at the client. If either party sends an incorrect/malformed message, and it is detected by the recipient, then they have *not* completed the protocol.

We begin the security definitions by first defining what it means for the GFS API to be secure with respect to server-to-server interactions. Following that definition, we define security with respect to clients and passive adversaries. The term "server" will be used as a shorthand for a data server and/or the metadata server. Whenever a statement applies to only one of the two, the type of server will be made explicit.

Definition 4. *The GFS API is* server-secure *if, with overwhelming probability, only authorized servers can complete the server-to-server protocols. Moreover, the GFS API is* client-secure*, if with overwhelming probability, only authorized servers and clients can complete the client–server protocols.*

That is, the GFS API is *server-secure* if for any unauthorized server U, it cannot successfully complete any of the server-to-server protocols, except with negligible probability (similarly for *client-security*). Later, we will show that our modifications achieve these properties. Note that, here, "authorized" covers both meanings of a server being authorized: authorized to be a data server, and with respect to a block. These definitions encompass the behaviors of an active adversary, but we must not neglect a passive adversary.

Definition 5. *The GFS API, is* passive-secure *if, with overwhelming probability, an adversary A, given a polynomially bounded number of messages from GFS protocol instances, cannot learn the contents of any data block.*

Later we will prove that our modifications to the GFS protocols achieve this property. We can now define what it means for a GFS API to be "secure."

Definition 6. *The GFS API, G, is* secure *if, with overwhelming probability, it holds that G is: (i)* server-secure*, (ii)* client-secure*, and (iii)* passive-secure*.*

No current GFS implementation achieves any of these properties, except against much more limited adversaries than the one considered here. For example, the work done by Yahoo! is *client-secure* for adversaries that cannot read arbitrary network traffic, but, since no file is encrypted, it is not *passive-secure*.

We assume that the adversary is polynomially bounded in both space and time and is allowed start any protocol, at any time, with any party that recognizes

that protocol. The adversary may try to impersonate another user or server, or use his own identity. The adversary *cannot* subvert known "good" servers or clients. As an additional power, we allow the adversary to observe any instance of any protocol at will (i.e., read arbitrary data on the network). As an example, a malicious user that has obtained root access on their machine fits our model; a malicious user breaking into another user's machine (or a data server) does not. Denial of service attacks are beyond the scope of this work.

We consider only the GFS protocols, all other communication is considered out-of-band. We also assume that there is some reliable, secure mechanism available to the metadata server (but not necessarily to data servers) for determining a user's identity (e.g., Kerberos). Finally, we will assume reliable message delivery, but the adversary is permitted to manipulate messages while in transit.

3 Proposed Architecture

In securing GFS-like file systems, we modify the constituent protocols to be provably secure against the adversary defined above. Messages between clients and servers, and among servers, must be authenticated to protect integrity and, in some instances, encrypted to maintain confidentiality of data. The data servers will need to register with the metadata server and clients will need to start sessions with the metadata server. The proposed architecture uses public-key cryptography to bootstrap itself to a place where it can use symmetric cryptography for greater efficiency. This section contains high-level descriptions of the secured protocols along with an asymptotic analysis of each protocol. Exact message parameters are omitted both for brevity and clarity. For notation, symmetric keys are denoted with a lower case k and the public and private halves of an asymmetric key used by principal P are denoted PK_P and SK_P, respectively. A message authentication code created with a key k will be denoted m_k. Variables that represent a name are capitalized. The metadata server will be denoted by M. It is assumed that M can securely determine the identity of a client, but data servers do not have this ability.

Client–Metadata Server. When a client C first interacts with M in a session, C sends the server two keys k_1 and k_2 (along with a nonce) encrypted with the public key of M and a MAC appended for integrity, created with k_1. The key k_1 is used to authenticate all subsequent messages while k_2 is used to encrypt some of the responses from M (e.g., an encryption key for sending file data). Upon receiving the message, M decrypts the keys and verifies the MAC, then M replies with the nonce and a MAC of the nonce using k_1. Efficiency-wise, this protocol requires $O(1)$ asymmetric encryption operations, each on input of size $O(1)$, and $O(1)$ symmetric operations, each on an input of size $O(l)$ and requiring $O(l)$ time, where l is the length of the message. All subsequent metadata requests and replies simply contain the request/response, a nonce, and a MAC.

When the client wishes to read/write a file block, they must first contact M to find the location(s) of the block. Along with the block's location, say server D,

M also sends back a ciphertext containing the access token and ephemeral keys for encrypting and authenticating the messages between C and D. Each key and token is valid for a single request. A copy of the ciphertext is encrypted with the long-term encryption key shared between the M and D, to be passed along to D by C. If there are multiple locations for the block, then there is a ciphertext and token for each location, as the client could potentially access any or all of them. If the replication factor is r and the file has n blocks, M must create rn tokens and send a message of size $O(rn)$. Note, according to [27], while files can be quite large, at Yahoo! the average file has 1.5 blocks. With a replication factor of 3 this gives an average of 4.5 tokens created when opening a file.

For a write request, the client contacts M each time it wants to add a block to the file. M picks locations for the r replicas and determines the "pipeline" of servers: where the client sends the data to the first server, who forwards it to the next, etc., rather than have the client communicate with each server individually. As noted above, when writing a block, the receiving data server will need to receive from the client an access token and a ciphertext created by M. Thus, M creates r tokens and r ciphertexts, one for each data server in the pipeline. Each ciphertext contains the necessary information (and ciphertexts) for the corresponding data server to continue the pipeline (detailed below). Note that each subsequent encryption is performed on an incrementally longer message. For simplicity, assume that the increment i is fixed. Then we have that $i + 2i + \cdots + ri = O(ir^2)$ bytes must be encrypted to produce a ciphertext of length $O(r)$. Overall the cost to M is $O(ir^2)$. Note, however, that the message generated by M is typically just a few hundred bytes; so these operations are not a significant cost. The overall message size for writing a block is $O(r)$, as in the original GFS.

Client–Data Server. The interactions between clients and data servers consist entirely of requesting and serving read/write operations. The response from the metadata server to the client (when it starts a request) contains two ephemeral keys k_a and k_e that will be used to authenticate and encrypt (respectively) messages between C and D.

Read and write requests, though similar, require slightly different protocols. For a read request, C receives a list of tuples L from M containing all of the information needed by C to read any block of the file (e.g., access token, encryption key, etc.). Suppose C wishes to read the block b located at data server D. C contacts D and sends the read request along with a nonce and the ciphertext c from b's tuple in L. C also creates and sends a ciphertext c' containing the nonce, the access token t for b, and the client's identity C, encrypted with k_e. Once D receives the request, it decrypts c to obtain the keys and C's identity. D then decrypts c' and checks the access token t. If t is valid, D sends back b encrypted with the same ephemeral key k_e. Both messages are authenticated via a MAC computed with k_a. Note that only $O(1)$ encryption and MAC operations are performed in both sending and receiving b, but the computational cost for each is proportional to the length of the message.

To write a block, C first sends the request to M who replies with the name of the server D (who will hold the block), an access token, two ephemeral keys

(as above), and a ciphertext c_D constructed for D. Here, c_D contains the information needed by D to verify the client's identity and authorization, as well as information about the next server in the pipeline so that D can forward the data. M also sends a ciphertext c for C containing essentially the same information as c_D. Each message between M and C, again, is authenticated with a MAC.

C then contacts D to write the block. First, C generates a nonce and encrypts it along with the block data d, using k_e. C then constructs a message containing the write request, the encrypted data, the nonce, the ciphertext c_D, and a (newly created) ciphertext c_2 containing the access token t, all of which is authenticated with m_{k_a}. D decrypts c_D to obtain the ephemeral keys and verify m_{k_a}. Following this, D decrypts c_2 to obtain t and verifies it. D subsequently forwards the data to the next server, D', then decrypts and writes the data to disk. Note that the data is not reencrypted with a new key as each server in the pipeline has a copy of k_e, given to it in the ciphertext created by M. Finally, D replies with final status s of the write, authenticated with m'_{k_a}.

Related to efficiency, we see that the initial message sent by C to D is of size $O(r + l)$—as in the original GFS—where r is the replication factor and l is the length of the block. Part of the message sent is a ciphertext of size $O(r)$, which was constructed by M for D. The encryption requires $O(l)$ time and the MAC computation takes $O(r + l)$ time. Each data server in the pipeline verifies the MAC of the data and then forwards it to the next node in the pipeline before decrypting it—avoiding a possible decryption-reencryption bottleneck. Thus, each data server needs to perform a MAC calculation on a message of size $O(r + l)$ and a single decryption operation on a ciphertext of length l.

Data Server–Metadata Server. An essential part of maintaining data security and integrity is preventing a malicious user from spoofing or manipulating any communication between data servers and M. The first step in ensuring security is to prevent any spurious data servers (i.e., those started and controlled by the attacker) from registering as data servers. To effect this, the system administrator possesses an asymmetric key pair $(\mathrm{PK}_A, \mathrm{SK}_A)$, with M possessing the public half. M itself has its own asymmetric key pair $(\mathrm{PK}_M, \mathrm{SK}_M)$, which will be utilized by the data servers.

When a data server D starts, it seeks to register with M. Part of the registration message is a pair of symmetric keys k_a^D and k_e^D to be shared with M. The key k_a^D is used to create a MAC for each subsequent message between D and M, as well as for creating the access tokens for blocks hosted by D. The key k_e^D is used for encrypting messages from M to D. The keys themselves are encrypted with PK_M, along with a nonce, to produce the ciphertext, which is added to the registration message. The administrator then signs the message and D sends it to M. Upon receiving the message, M verifies the signature, decrypts c, then saves the keys k_a^D and k_e^D. M then sends some start-up information to D, authenticated with $m_{k_a^D}$. Note that the efficiency of the registration protocol is near optimal, as there are $O(1)$ symmetric and asymmetric cryptographic operations. The asymmetric operations are all on $O(1)$-sized input, while the symmetric operations require $O(l)$ time, where l is the length of the message.

After registration, D periodically sends heartbeat and "block report" messages (usually combined together) to M. The heartbeat attests to D's liveness while the block report is simply an update on any block state changes (e.g., added or deleted). When receiving either of these, M replies with a (possibly empty) list of commands for D to execute. The heartbeat message is typically a fixed size and so D requires $O(1)$ time to compute the MAC. But, with the block report, if the report is of length l', then the MAC takes $O(l')$ time to compute (but still only requires $O(1)$ space).

Data Server–Data Server. Data servers must also interact with each other, but only in limited circumstances: as part of a pipeline when writing a file block and transferring blocks during load balancing. In both situations, the sender appears to the receiver to be just another client writing a block. Thus the sending data server must have enough information to emulate a client in the client–data server protocol for writing blocks.

Suppose we have a pipeline of n servers, $D_1, \ldots, D_n$, where D_i is the i-th server in the pipeline. The D_i will need to forward the data to D_{i+1}. D_n simply receives the data and does not forward it further. For each D_i, the metadata server M creates a ciphertext c_{D_i} containing the information necessary for D_i to continue the data pipeline. The c_{D_i}'s are nested within each other, so that c_{D_1} contains c_{D_2}, which contains c_{D_3}, etc. Each server D_i removes the i-th layer of encryption and obtains, along with other information, the ciphertext $c_{D_{i+1}}$. The "other information" includes: a nonce, two ephemeral keys, an access token, and the identity of D_{i+1}. The keys and access token play the same role here as they do in the client–data server protocol. Transferring blocks during load balancing is essentially identical to the client–data server protocol for writing a block. For more details, see the above section describing client–data server interactions. Note that these inter-server interactions have the same efficiency as the client–data server protocol for writing a block.

As part of increasing the security of GFS-like file systems, we have the data server become a point of enforcement for the access controls. Suppose a client wants to access a file consisting of blocks $b_1, \cdots, b_n$. The metadata server M first checks that C has access rights, then creates a token t_i for each block b_i. Each t_i is valid only at the corresponding data server that holds a copy of b_i, call it D. The token itself is a simply a MAC created from the token information and the long-term key k_a^D (described above). When a request to operate on b_i arrives, D will check the token t_i before servicing the request.

4 Security

To prove the security of the protocols, we will define a "game" for the adversary to play. The game simply encapsulates a standard cryptographic reduction: we will reduce the security of the protocols to the security of the cryptographic primitives used (i.e., MACs and signatures). The setup for the reduction is a bit unusual, but, as shown below, the formulation is equivalent to the standard

reduction framework. We assume that the encryption schemes are semantically secure and the MAC and signature schemes are existentially unforgeable under chosen-plaintext attacks. All keys are assumed long enough to be computationally infeasible to brute-force.

4.1 Security Game

We wish to accurately model the adversary, the system, and their interactions with each other, while giving the adversary as much flexibility as possible. We define a *message-creation game* where A has access to a simulator S that maintains a simulation of the cluster. A dictates all the events in S. Each event details a protocol to be executed with principals and parameters chosen by A. A may submit each message in a protocol as separate events with an arbitrary (but polynomially bounded) number of events inbetween. We do not allow parallel executions of the protocols, e.g. multiple instances of server registration initiated by the same server. Cryptographic keys are chosen by A only when the adversary's role in the protocol generates the keys. Otherwise the keys are generated and maintained by the simulator and are hidden from A.

After an event e is submitted to S and the internal state of S is updated, S outputs a transcript of the (full or partial) protocol execution dictated by e. The adversary wins the game if, after some polynomial number of steps, he produces a message that is unique, well-formed, and correctly verifies at the intended recipient (i.e., a principal in the simulator). We restrict the output message such that it must be for a protocol of which A is *not* one of the principals—otherwise A can win trivially. Note that each protocol consists of exactly two messages: an initiation message and the response. If the output of the adversary is a response message, then, for A to win, there must have been an event detailing the initiation message for that protocol. This setup gives A much more power over the cluster than would be possible in the real world. However, we will prove that the protocols are secure against even this more powerful adversary.

4.2 Security Proofs

The following proofs will use the game described above to reduce the security of the protocols to the security of a cryptographic primitive: whether it is a digital signature or a message authentication code. The registration protocol is the only protocol that involves an asymmetric signature for integrity and authentication; all other protocols use MACs to provide the same protections. As such, the security of the registration protocol's initial message reduces to the security of the digital signature, while the security of every other message reduces to the the MAC. The next two parts give outlines of formal proofs demonstrating these reductions.

Data Server Registration. Assume there exists a probabilistic polynomial-time adversary A, taking as input the public key of the metadata server PK_M,

and the public key of the system administrator PK_{SA}, who can win the message-creation game with non-negligible probability. Moreover, assume A's output is the initial message of the registration protocol. We will construct an algorithm B that uses A as a subroutine to break the signature scheme. B takes as input the public key PK of the signature oracle $\mathcal{O}$ and the security parameter 1^k.

To use the adversary A, B will need to emulate the simulator. For each event e output by A, B will run the protocol with the given parameters, update the state s of the cluster, and return a transcript t to A. Whenever e dictates the registration of a new data server, B forms the registration message m in the usual way and then queries $\mathcal{O}$ on m to get the signature σ. The signature σ is used in place of the administrator's signature. All other protocols are executed normally with B exactly mimicking the simulator. Eventually, A outputs a message m. If the message is anything other than the initial message of the registration protocol, B fails. Otherwise, B extracts the signature $\tilde{\sigma}$ and the data d that was signed and outputs the pair $(d, \tilde{\sigma})$. If A won the game, then m verifies at its intended recipient: the metadata server. This implies that $\tilde{\sigma}$ was a valid signature for d even though A had no access to the key, i.e. A produced a forgery.

Since B outputs, essentially, the output of A, B succeeds exactly when A succeeds. Thus, if the transcripts given to A are distributed properly, B inherits the success probability of A. Note that B runs exactly the protocols in GFS, with the parameters and principals determined by A each time. Furthermore, the signature oracle $\mathcal{O}$ outputs signatures using a key that is from the same scheme as the key of the system administrator. Thus, since (almost) all protocols are run exactly as in the simulator and the signatures are from a distribution identical to the expected distribution, we have that the input to A is distributed *exactly* as expected. This implies that if A has a non-negligible probability of winning the game, then B has a non-negligible probability of producing a forgery. However, this contradicts the security of the signature scheme. Thus, it must be that A has only a negligible probability of winning the game when attacking the initial message of the registration protocol.

General Proof of Security. We now prove the security of the remaining protocols as a group. First, it is important to notice that each of the other protocols have the same structure: principal P_1 sends a message μ with a MAC m, and then principal P_2 replies with a message μ' and a MAC m'. We can exploit this structure and use an adversary A that can complete one of these protocols to create an adversary B that can break the security of the MAC scheme. Note that here we are assuming that the protocols, and the confidential values transferred therein, are secure against a passively observing adversary—we will prove this property later. In this reduction, B will have access to polynomially many oracles for the MAC scheme, each independently instantiated (i.e., the key in each oracle is chosen at random). B is successful if it can forge a message for *any* of the instantiated oracles.

Note that having polynomially many oracles is equivalent in power to having a single oracle. Briefly, given a single oracle $\mathcal{O}$ of polynomially-bounded power (e.g., a signature oracle) and an adversary who succeeds against polynomially

many oracles, we can "guess," with non-negligible probability, which oracle will be attacked by the adversary. Using this guess, we can then use $\mathcal{O}$ to satisfy queries to the "to-be-attacked" oracle and simulate the remaining oracles. If the adversary succeeds with non-negligible probability and we made a correct guess, then we succeed with non-negligible probability.

Since we do not know how many oracles will be needed by B, we give B access to a meta-oracle $\mathcal{MO}$ that will manage the oracle instances. $\mathcal{MO}$ has three operations: *start*, *stop*, *query*. The command *start* takes no parameters, instantiates a new MAC oracle with a randomly chosen key, and returns a unique identifier for the oracle. The *stop* operation takes an oracle identifier as input and "destroys" the indicated oracle instance, making further queries under that identifier invalid. The *query* operation takes as input the identifier for an oracle and the input to the oracle, and then returns the output from the selected oracle.

As before, B emulates the simulator as closely as possible when interacting with A. Whenever an event e starts a new protocol instance, B determines whether or not a new oracle must be instantiated or if previously instantiated oracle must be used. For instance, if a client C is reading a block from a data server D, then B must ask $\mathcal{MO}$ to start a new oracle, since a unique MAC key is used in each block transfer. B would use a previously instantiated oracle for, say, a data server sending a heartbeat to the metadata server. However, if A is one of the principals in the protocol, then, since A knows the keys, B must itself compute the MAC for the message, all other MACs are computed by the oracles. Note that this does not affect B's chance of success as A is forbidden from attacking protocols in which it is a principal.

One difficulty in this reduction is what to do when the key for the MAC is sent as part of the message or in a previously executed protocol (e.g., the ephemeral keys for reading a block). Since the oracles are used for (almost) all MAC generation, B does not have access to the keys and cannot include them in any messages. The solution is to choose the keys in the message at random—except for those instances where A is a principal. While substituting in a random key does not perfectly mimic the simulator, we show next that the distribution of messages is computationally indistinguishable from the ideal distribution.

Suppose that A can distinguish the distribution of messages produced by B from the expected distribution, and that we have access to an encryption oracle for the cipher used to encrypt the keys. Then there exists an A' that, given a sample from one distribution or the other, distinguishes the distributions with a non-negligible advantage over $\frac{1}{2}$. Construct C that generates two random keys k_0 and k_1, and then constructs two messages m_0 and m_1 (both conforming to one of the protocols). C then submits m_0 and m_1 to the oracle to get $\mathcal{O}(m_b) = c_b$ for a random $b \in \{0, 1\}$. Once it has c_b, C finishes constructing the protocol message M and computes the MAC using k_0. C submits M with the MAC to A' and outputs whatever A' does. C is correct exactly when A' is correct. Thus C has a non-negligible chance to distinguish the encryptions of m_0 and m_1, contradicting the semantic security of the cipher. Thus, the view of A is computationally indistinguishable from the expected view. Since B succeeds exactly when A

succeeds, if A wins non-negligiblely often, then so does B, contradicting the security of the MAC scheme. Thus, it must be that there *does not* exist an A that can win the game with non-negligible probability. This, combined with the previous result, implies that A cannot win the game for any of the protocols.

Proof of Security of the Access Token. The definition of security for the access token is most naturally existential unforgeability under chosen-plaintext attacks. That is, with overwhelming probability, any token created by the adversary will not verify at any of the data servers. Note that since the token itself is simply a MAC of a few specific parameters, the security of the token is exactly the security of the MAC scheme. Thus, since we assumed that the MAC is secure, we have that the access tokens are also secure.

Proof against the Passive Adversary. To prove passive security, we must ensure that the adversary A cannot learn the contents of any data block. Since A is not interacting with any other principals, the only way for A to learn the contents a block is for A to capture the block intransit. File blocks only travel between and among clients and data servers and, as stated above, the file blocks are always encrypted before being transmitted. It is worth noting that in several instances, the key used to encrypt a file block is also sent with the block. However, the key is also encrypted with a semantically secure cipher. This layer of encryption should stymie the adversary A, unless A can acquire the key(s) or compute a non-negligible amount of information about the key(s).

The semantic security of the cipher implies that the passive adversary, with overwhelming probability, can only learn a negligible amount of information about any transmitted key (likewise for any key used to encrypt the transmitted key). Similarly, since the cipher used to encrypt the block data is also semantically secure and—it was assumed—the key is too long to brute-force in a reasonable amount of time, with overwhelming probability, the passive adversary A can only learn a negligible amount of information about the contents of the block. This is exactly the definition of being *passive-secure*, as desired.

Security Properties Proven. Overall, the above proofs give us the fact that an adversary (as described in Section 2), with overwhelming probability, cannot complete *any* of the protocols in the GFS API, giving us the *server-secure* and *client-secure* properties. Additionally, we demonstrated that with overwhelming probability the system is also secure against a passive adversary. Thus we have that the extensions given in this work give a GFS API that is *secure*.

5 Experimental Results

To demonstrate the practicality of this secured architecture, a proof-of-concept implementation was created by modifying the open-source Hadoop platform [14] to implement the above secured protocols. The changes were made to version 0.20.104.2 of the Yahoo! branch of the code (which has since been merged into

mainline Hadoop). This branch was chosen because it contains all of the Kerberos integration work performed by Yahoo!. This allows a more direct comparison of the efficiency of previous security work with the efficiency of this work.

Our implementation uses 2048 RSA for the asymmetric keys, and `UMAC128` for the message authentication codes [18]. The stream cipher Salsa20/12 from [2] is used for all data encryption—chosen for both its speed and strong security. The experiments were performed on a cluster of 40 Dell PowerEdge 1855s each running a dual-core 2.8GHz Intel Xeon with 8 GB of memory and 300GB of disk space—for a total of 12TB of disk space in the cluster. The operating system used on each is 64-bit Debian Linux. The metadata server was run on a quad-core Intel Core2 Q6600 at 2.4GHz with 4GB of memory. While the processor has 64-bit instructions, the OS was 32-bit Debian Linux with PAE.

We used standard benchmarks of Hadoop: Gridmix2, NNThroughputBenchmark, and TestDFSIO. Gridmix2 is a mix of various MapReduce jobs designed to stress HDFS in a number of ways while emulating a real-world workload and is regarded as the standard macro-benchmark for Hadoop clusters. NNThroughputBenchmark is used to test the throughput, and hence scalability, of the metadata server (called the NameNode in Hadoop). The TestDFSIO utility measures the raw read and write speed of the cluster. We summarize the results in Table 1.

Table 1. Comparison of our work against default Hadoop. The first column is in seconds, the second and third in MB/s and the remaining in operations per second.

	Gridmix2	Avg Read IO	Avg Write IO	Open	Create	BlockReport
Default Hadoop	23997s	58.6 MB/s	20.2 MB/s	45871	324	8333
Sec-Hadoop	26819s	27.9 MB/s	10.8 MB/s	6711	331	7821
% Slowdown	11.8	52.4	46.5	85.4	-2.1	6.1

Overall Performance. The Gridmix2 column in Table 1 shows that, overall, this work produces a 12% slow down of Hadoop. The work by Yahoo! in comparison achieves a 3% slowdown of the Gridmix2 benchmark, but none of the file data is encrypted. The remaining columns give the average IO rates for reads and writes when creating 40 files of 2048MB each with a replication factor of 3. Average IO is defined as the average the individual IO rates for the created files. We can see that the average IO rates for the secured Hadoop are a bit less than half of the rates for the default Hadoop. While this is a significant drop in performance, the effect of this is attenuated by the fact that cluster performance is not solely IO-bound. For example, even though our work has half the read/write performance, the overall impact was just a 12% slowdown for the cluster.

Scalability. GFS-like file systems are designed to rapidly scale upward, but growth is often limited by the capacity of the metadata server. Shvachko in [27] performs a detailed estimation of the practical limits of a Hadoop cluster assessing memory and computational costs. Looking at the same metrics, the memory overhead in our work is at most in the tens of kilobytes as only a few dozen

bytes are stored per server and client. The real cost of our modifications is computational: an increase in both the time spent processing messages from data servers and handling metadata operations from clients. Table 1 shows that the throughput of the metadata server decreases between 6.1% and 85.4%, depending on the action performed. While this reduces the scalability of the cluster, the limit would only affect very large Hadoop deployments. In particular, our rather modest metadata server is still able to handle several thousand operations per second. Thus, a secured cluster could easily scale to hundreds of servers and even to a few thousand. But, as the "Open" metric shows, a secured Hadoop will have trouble scaling past a few thousand nodes.

6 Other Related Work

Yahoo! has released their own version of Hadoop, an open source implementation of the Map-Reduce framework, including a security-enhanced HDFS [29]. This version incorporates Kerberos authentication into all communication: all servers and users are registered as principals in the Kerberos database and must authenticate before sending any messages. Their work provides message integrity and authentication, but not confidentiality. Recent work has been done on distributed file systems that operate as the underlying cloud storage. However, security is rarely, if ever, mentioned. The efforts in [11] give a file system that is similar to GFS but uses a collection of metadata servers instead of a single central server and finer-grained resource control. User authentication is the only security feature. The work in [17] provides a flexible and modular cloud storage system where components can be swapped in/out to provide customized levels of reliability, efficiency, and consistency semantics, but security is not discussed.

Previous work on security in GFS-like file systems is sparse. Airavat modifies Hadoop to support mandatory access controls and store the security labels with the blocks [25]. However, MAC policies are often unwieldy, difficult to set up, and time-consuming to maintain. Also, the implementation results in a slow-down of up to 25%. TPlatform [22] has the same access control limitations as the original Hadoop. CloudStore, another implementation of GFS, does not have any access controls [6]. Another effort by [16] builds fine-grained access controls on top of the Hadoop file system (HDFS), but it assumes that HDFS is inherently secure.

SUNDR is a network file system that seeks to reduce the amount of trust clients must give to the file servers—the converse of our goal: reducing the trust given to clients—and implements fork-consistency [19]. GPFS is another distributed file system that provides efficient, fault-tolerant storage [26]. Access control checks are performed at the storage servers and users are assumed to be relatively benevolent. The Panache file system is designed to be fully parallel in all read/write operations, utilizing GPFS to store file data and metadata and uses parallel NFS on the client-side for reading/writing data [10]. SFS aims to provide a secure file system over an untrusted network (e.g., the Internet) using "self-certifying paths" via public-key based client-server authentication [20]. Related work on the integrity verification of outsourced file systems includes authenticated data structures (e.g., [13,21]) and proofs of data possession (e.g., [9]).

7 Conclusion and Future Work

This work demonstrates the feasibility of greatly enhancing the security of GFS-like file systems, while maintaining a reasonable overhead. However, a 12% slowdown is not insignificant and could be improved through various avenues. One avenue would be to add more flexibility in the architecture (e.g., choosing to encrypt block data but not use a MAC) so that administrators can more finely tune the trade-off in security and efficiency. Additional experimentation with other cipher suites and MAC schemes could be helpful to reduce the overhead from the security. Another avenue to explore would be utilizing the work of [3] to provide transport-level encryption for all traffic, transparently to the Hadoop cluster itself. One weakness of our secured system is the lack of confidentiality protections for file metadata. While the data itself could not be pilfered, metadata such as file names can contain sensitive information. Protecting metadata is a logical next step in increasing the assurance of GFS-like file systems.

Acknowledgments. Research supported in part by the National Science Foundation under grants CNS–1012060, CNS–1012798, and CNS–1012910 and by a NetApp Faculty Fellowship. We thank James Lentini for useful discussions.

References

1. Becherer, A.: Hadoop Security Design: Just Add Kerberos? Really? (2010), http://media.blackhat.com/bh-us-10/whitepapers/Becherer/BlackHat-USA-2010-Becherer-Andrew-Hadoop-Security-wp.pdf
2. Bernstein, D.J.: The Salsa20 Family of Stream Ciphers. In: Robshaw, M., Billet, O. (eds.) New Stream Cipher Designs. LNCS, vol. 4986, pp. 84–97. Springer, Heidelberg (2008)
3. Bittau, A., Hamburg, M., Handley, M., Mazières, D., Boneh, D.: The case for ubiquitous transport-level encryption. In: USENIX Security, pp. 26–42 (2010)
4. Borthakur, D.: HDFS Architecture, http://hadoop.apache.org/hdfs/docs/current/hdfs_design.html
5. Borthakur, D., Gray, J., Sarma, J.S., Muthukkaruppan, K., Spiegelberg, N., Kuang, H., Ranganathan, K., Molkov, D., Menon, A., Rash, S., Schmidt, R., Aiyer, A.: Apache Hadoop goes realtime at Facebook. In: SIGMOD, pp. 1071–1080 (2011)
6. CloudStore, http://code.google.com/p/kosmosfs/
7. Cordova, A.: MapReduce over Tahoe–a least-authority encrypted distributed file system (2009), http://www.cloudera.com/videos/hw09_mapreduce_over_tahoe
8. Dittrich, J., Quiané-Ruiz, J., Jindal, A., Kargin, Y., Setty, V., Schad, J.: Hadoop++: Making a yellow elephant run like a cheetah (without it even noticing). PVLDB 3(1), 518–529 (2010)
9. Erway, C., Küpçü, A., Papamanthou, C., Tamassia, R.: Dynamic provable data possession. In: CCS, pp. 213–222 (2009)
10. Eshel, M., Haskin, R., Hildebrand, D., Naik, M., Schmuck, F., Tewari, R.: Panache: A parallel file system cache for global file access. In: USENIX FAST (2010)
11. Fesehaye, D., Malik, R., Nahrstedt, K.: A Scalable Distributed File System for Cloud Computing. Tech. rep., University of Illinois at Urbana-Champaign (2010), http://www.ideals.illinois.edu/handle/2142/15200

12. Ghemawat, S., Gobioff, H., Leung, S.: The Google file system. In: SOSP, pp. 29–43 (2003)
13. Goodrich, M.T., Papamanthou, C., Tamassia, R., Triandopoulos, N.: Athos: Efficient Authentication of Outsourced File Systems. In: Wu, T.-C., Lei, C.-L., Rijmen, V., Lee, D.-T. (eds.) ISC 2008. LNCS, vol. 5222, pp. 80–96. Springer, Heidelberg (2008)
14. Hadoop, http://hadoop.apache.org
15. Jiang, D., Ooi, B.C., Shi, L., Wu, S.: The performance of MapReduce: An in-depth study. PVLDB 3(1-2), 472–483 (2010)
16. Kantarcioglu, M., Khan, L., Thuraisingham, B., Gupta, A., Vyas, M., Khadilkar, V., Mishra, N.: Fine-grained Access Control using HIVE (September 2010), http://cs.utdallas.edu/secure-cloud-repository/Hive-AC/hive-ac.html
17. Kossmann, D., Kraska, T., Loesing, S., Merkli, S., Mittal, R., Pfaffhauser, F.: Cloudy: A modular cloud storage system. PVLDB 3(2), 1533–1536 (2010)
18. Krovetz, T.: UMAC: Message Authentication Code using Universal Hashing. RFC 4418 (Informational) (March 2006), http://www.ietf.org/rfc/rfc4418.txt
19. Li, J., Krohn, M., Mazières, D., Shasha, D.: Secure untrusted data repository. In: USENIX OSDI, pp. 91–106 (2004)
20. Mazières, D., Kaminsky, M., Frans Kaashoek, M., Witchel, E.: Separating key management from file system security. In: SOSP, pp. 124–139 (1999)
21. Papamanthou, C., Tamassia, R., Triandopoulos, N.: Authenticated hash tables. In: CCS, pp. 437–448 (2008)
22. Peng, B., Cui, B., Li, X.: Implementation Issues of a Cloud Computing Platform. IEEE Data Engineering Bulletin (2009)
23. Ristenpart, T., Tromer, E., Shacham, H., Savage, S.: Hey, you, get off of my cloud: Exploring information leakage in third-party compute clouds. In: ACM CCS, pp. 199–212 (2009)
24. Rocha, F., Correia, M.: Lucy in the sky without diamonds: Stealing confidential data in the cloud. In: IEEE/IFIP DNSW, pp. 129–134 (2011)
25. Roy, I., Ramadan, H.E., Setty, S.T.V., Kilzer, A., Shmatikov, V., Witchel, E.: Airavat: Security and privacy for MapReduce. In: USENIX NSDI, pp. 297–312 (2010)
26. Schmuck, F., Haskin, R.: GPFS: A shared-disk file system for large computing clusters. In: USENIX FAST, pp. 231–244 (2002)
27. Shvachko, K.V.: HDFS scalability: the limits of growth. USENIX; Login 35(2), 6–16 (2010)
28. Wilcox-O'Hearn, Z., Warner, B.: Tahoe: The least-authority filesystem. In: ACM StorageSS, pp. 21–26 (2008)
29. Yahoo! Distribution of Hadoop, http://developer.yahoo.com/hadoop/