Toward Securing Untrusted Storage Without Public-Key Operations

Dalit Naor
IBM Haifa Research Lab
Tel-Aviv, Israel
dalit@il.ibm.com

Amir Shenhav
School of Electrical Engineering
Tel-Aviv University
amirshen@eng.tau.ac.il

Avishai Wool*
School of Electrical Engineering
Tel-Aviv University
yash@eng.tau.ac.il

ABSTRACT

Adding security capabilities to shared, remote and untrusted storage file systems leads to performance degradation that limits their use. Public-key cryptographic primitives, widely used in such file systems, are known to have worse performance than their symmetric key counterparts. In this paper we examine design alternatives that avoid public-key cryptography operations to achieve better performance. We present the trade-offs and limitations that are introduced by these substitutions.

Categories and Subject Descriptors

D.4.6 [**Operating Systems**]: Security and Protection—*Access controls, Authentication*; E.3 [**Data Encryption**]: Public key cryptosystems

General Terms

Security

Keywords

secure file systems, network attached storage

1. INTRODUCTION

1.1 Motivation

Network based storage solutions, such as Storage Area Networks (SANs), provide users with the opportunity to outsource storage management (e.g., SUN's SSP), and to achieve good performance when accessing the data. However, crucial security problems arise when the storage environment is no longer trusted. A secure storage system needs to provide confidentiality, data integrity, authenticity, freshness guarantees and access control. Recent works that have addressed these challenges presented cryptographic file systems that provide secure, shared storage without trusting the file system itself. These file systems use cryptographic access control and encryption of data at rest performed by the client side. However, the low performance of such file systems is one of the main reasons to limit the adoption such solutions.

It is common knowledge that public-key cryptography algorithms are orders of magnitude slower than their symmetric key equivalents. Hence, the fact that these works use public-key cryptography, motivates a careful examination of the reasons of its usage, while looking for symmetric key alternatives.

Public-key cryptography is used in secure file systems for the following reasons:

Key Distribution

The cryptographic algorithms involved in securing the file system require the use of several keys for different operations. Systems such as [15, 7, 5] use two types of keys: *user keys*, that are bound to each user's identity, and *file keys*, that are assigned to each file, group of files or even to a block in a file, and are handed to the users that share this file. Therefore, a mechanism for key distribution is required. This mechanism may be in-band, which means that the file system manages or participates in the distribution process, or out-of-band, assuming an existing key distribution infrastructure. Most of the systems use public-key cryptography to allow secure, confident and authenticated key distribution.

Digital Signatures

Cryptographic file systems use digital signatures to achieve three goals: data integrity, user authentication and differentiation of readers from writers. The differentiation is an outcome of the asymmetry of public-key signatures between the signer and the verifier. Users who are only allowed to read the file are handed only the public key and thus cannot change the file without being noticed.

1.2 Related Work

Storage security has attracted growing interest in recent years. As the storage world advances, it becomes more complicated to secure, yet more vulnerable to attacks. In [18], a framework that defines the taxonomy of storage security is presented, together with casting previous works onto this framework. Another survey describing current research in

*Supported in part by an IBM Faculty Award.

StorageSS'05, November 11, 2005, Fairfax, Virginia, USA.

this area is presented in [20]. These works emphasize the different assumptions of system architecture, the trust model and security goals in the different works that are reviewed. Earlier works as CFS [1] addressed relatively simple confidentiality issues. Most others trust the file server but wish to protect against malicious users using or snooping the network [11, 21]. The most advanced systems try to avoid trusting either the file server or the storage facilities [5, 7, 15]. In these file systems the cryptographic operations are done at the client side to provide encryption of data at rest and cryptographic access control that do not depend on the reliability of the file server or the storage. However, not trusting the file system entirely can be computationally expensive. Therefore, some variants of [15] relax the assumption: the file system or disks are untrusted regarding confidentiality issues, but are trusted to ensure integrity and access control. Another important work is the SUNDR [12, 10] system that addresses the problem of consistency of untrusted file server, pointing out the difficulty to cope with rollback attacks in such systems.

Our work follows the model of untrusted server storage as in the systems SiRiUS [5], Plutus [7] and SNAD [15]. SiRiUS and Plutus can be viewed as complementary works: SiRiUS handles key distribution issues but operates as an add-on that does not change the underlying file system; Plutus does not refer to key distribution but presents a new design for the file systems itself, providing efficient random access, filename encryption and revocation. SNAD, like SiRiUS, uses in-band key distribution, but in contrast to SiRiUS, suggests that keys refer to users and not to files. Both SNAD and Plutus are ambivalent concerning the trust they have in the file system. They both require the server or disk to perform checks before reading or writing the data as an access control measure that is effective only if the file server has not been compromised.

SiRiUS, Plutus and SNAD rely on the public-key cryptography in their design except that file or block encryption is done with symmetric-key algorithm. In SNAD a symmetric HMAC is suggested as an alternative to signatures—but then the user must rely on the file server to handle the access control and to differentiate readers from writers.

1.3 Contributions

Our work suggests methods to improve the performance of cryptographic file systems like Plutus and SiRiUS by replacing the public-key cryptography with symmetric key algorithms. The main issues we address are in-band key distribution using symmetric key methods and providing data integrity and cryptographic access control without public-key signatures. For key distribution problem, we suggest the methods of Leighton-Micali and Blom. For replacing public-key signatures we use MACs and one-time signatures that are most adequate for scenarios of either few read-only users or of one publisher that addresses many readers. In addition, we discuss ways to improve the method of revocation without re-encryption that was introduced in Plutus.

Some alternatives to public-key cryptography require the consideration of new parameters and trade-offs that are introduced to the system. Such trade-offs can be a larger space consumption or the introduction of new constrains to the file system. We discuss these trade-offs and suggest scenarios which encourage the adoption of the proposed ideas.

As a work in progress, this paper introduces the basic ideas and architecture with preliminary trade-off analysis and recommendations. Detailed implementation including benchmarks will be issued in future work.

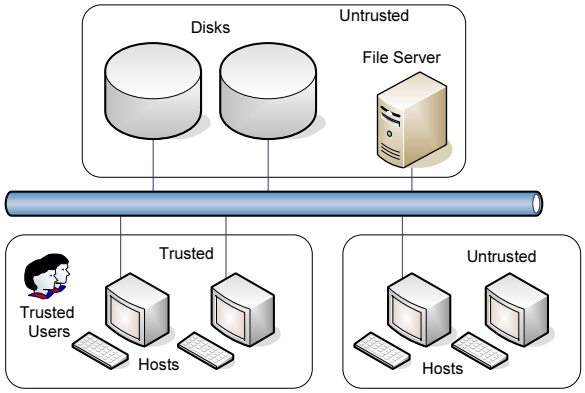

Figure 1: System and Trust Model

Paper Organization

In the next section we define the problem and underlying assumptions. In section 3 we examine alternatives to in-band key distribution based on symmetric key distribution algorithms. Section 4 addresses the problem of providing both file integrity and read/write access differentiation. The problem of efficient revocation without re-encryption is described in section 5. In section 6 we present our conclusions and future work.

2. DEFINITIONS AND ASSUMPTIONS

Our work refers to a networked storage system where clients access their data using a remote file system. The general architecture is that of a SAN where the network includes users, hosts, disks and a file server as described in Figure 1.

Within the system each user can function in one of the following roles, on a per-file basis[1]:

- ***owner*** - The owner of a file is able to read, modify and delete the data. The owner handles the access policy of the file, i.e., the owner provides permissions to other users that allows them to read or write the file. The owner can also revoke users' privileges.
- ***writer*** - The writer is privileged by the file owner to read and modify the data of a file.
- ***reader*** - The reader is privileged by the file owner only to read the data of a file.

We assume that the file server and the disks are untrusted in the sense that users hope that server and the disks will store the data properly, but they do not rely on them to maintain data confidentiality and integrity. The network is also regarded as untrusted, therefore all the data is encrypted and signed on the host before sent to the file server. Availability issues are not addressed in our work. The system assumes that it cannot prevent an adversary from reading, changing or destroying data on a compromised file server or disk. Instead we require that data secrecy is maintained

[1]The definitions in this section are based on the taxonomy of [18].

by encryption of data at rest, and we require the system to provide the users with the ability to check the integrity and authenticity of each file. The system is also required to prevent legal readers of the file from changing it, such that the other users that have read or write permissions can detect such unprivileged change.

The notion of a ***rollback attack*** was introduced by [12]. In this sort of attack an adversary replaces the current file with a valid, earlier version. In [5], protecting against a rollback is also referred to as a ***freshness*** requirement. Our system does not address this threat and assume independent treatment with means like out-of-band communication, some small secure storage or with the technique presented in [10] that provides fork consistency.

The system is required to provide distribution means for all the keys that are used in the system. We refer to this requirement as ***in-band key distribution***. In other words, we do not rely on any other external key distribution infrastructure. The key distribution is a part of the meta-data of the file. Throughout this paper we will use the term meta-data to describe the added information that is used to secure the file system regardless of any other meta-data that is used in the file system.

Revocation of users by the owners of the files is required to be a ***lazy***: access permissions are changed immediately but an unchanged or cached file can be read by the revoked user until the file is updated.

3. KEY DISTRIBUTION

3.1 In-Band Key Distribution

In order to share an encrypted and signed file with other users we need a certain method of key distribution. Plutus [7] relies on an external out-of-band key distribution mechanism. We argue that such infrastructures are not common and may not provide the required performance. Instead, we prefer the in-band key distribution suggested in SiRiUS [5] using the file's meta-data. Each file has its own encryption (*FEK*) and signature (*FSK*) keys that are generated by the owner when the file is created. In SiRiUS, these keys are encrypted using the public keys of any user to whom the owner grants access. The encrypted keys are saved as part of the meta-data of the file. When a user needs an access to a file he reads the relevant meta data and decrypts the *FEK* and *FSK* using his private keys. The meta-data is also signed with the owner private key, thus the user needs also to get the owner's public key to verify his signature on the meta-data. Therefore, a database that contains the users' public keys should be part of a SiRiUS system, as well some mechanism for handing the private keys to the users.

In the next sections we describe two different methods that achieve the same security goals as the ones achieved by SiRiUS, but using efficient symmetric-key cryptography and without an external key distribution mechanism.

3.2 Leighton and Micali's Method

The work of Leighton and Micali [9] introduces methods for key agreement without public-key operations. One scheme in the paper enables secret key agreement between any two parties using a public database. Below is a brief description of the protocol (see also [19]):

A trusted agent randomly generates two *master secret keys* K and K'. Using these keys the trusted agent generates and distributes each user i a *key exchange key* K_i and an *individual authentication key* K'_i such that:

$$K_i = h(K, i) \qquad K'_i = h(K', i)$$

where $h(\cdot)$ is a pseudo-random function. In practice $h(\cdot)$ can be chosen to be implemented with HMAC algorithm. The trusted agent publishes a public data base of two matrices $\mathcal{P}$ and $\mathcal{A}$ that includes *pair keys* and *authentication keys*:

$$\mathcal{P}_{i,j} = h(K_i, j) \oplus h(K_j, i) \qquad \mathcal{A}_{i,j} = h(K'_i, h(K_j, i))$$

When user i wants to encrypt a message to user j he reads the public values $\mathcal{P}_{i,j}$ and $\mathcal{A}_{i,j}$ and calculates his common secret key $K_{i,j}$ as

$$K_{i,j} = \mathcal{P}_{i,j} \oplus h(K_i, j) = h(K_j, i)$$

and verifies the authenticity of this key by verifying that:

$$h(K'_i, K_{i,j}) = \mathcal{A}_{i,j}$$

Clearly, user j can also calculate $K_{i,j}$ since he has is own private K_j and knows the identity of user i. Thus j can decrypt the message.

The Leighton-Micali scheme can be used for key distribution of file encryption and signature keys instead of the public-keys used in SiRiUS. The matrices $\mathcal{P}$ and $\mathcal{A}$ are stored on the file system. Note that this does not imply trust in the server: the $\mathcal{P}$ and $\mathcal{A}$ matrices are not secret. The file owner i can get a symmetric key $K_{i,j}$ for any user j to whom he wants to give access. Using $K_{i,j}$, the owner can encrypt the file keys *FEK* and *FSK*. A user j who wants to access the file needs only to read the owner ID number from the meta-data and re-calculate his decryption key $K_{i,j}$ to reveal the *FEK* and *FSK*. Note that the user does not even need to access the key matrices to do so.

As mentioned, to verify the integrity and authenticity of the meta-data, SiRiUS requires the owner to sign the meta-data using his private key. This signature provides the users with the ability to detect any illegitimate changes to the meta-data. If we naively replace the signature with a MAC (*Message Authentication Code*), then a system-wide shared MAC key is be needed, and also all the users can change the meta-data. Instead, by using the Leighton-Micali method, we suggest the following solution:

- Rearrange all the meta-data in a per-user block fashion, so each block contains all the meta-data relevant for that user (encrypted keys, owner's identity, filename, etc.).
- Calculate an unkeyed MDC (*Message Digest Code*) separately on each block.
- Encrypt the block for each user with the user's key $K_{i,j}$ as suggested above.[2]

Besides lengthening the meta-data negligibly, this method has the drawback that each user is aware only of the meta-data relevant to him. If other parts of the meta-data are corrupted or changed, for example, if a whole block of one of the users is deleted, the other users are unaware that something suspicious happened—except the owner that can calculate and save the MDC over all the blocks. However, given that untrusted storage availability is an unsolved problem, we believe this is an acceptable limitation.

Note that the Leighton-Micali method has similar requirements for distribution of the users' keys as the public-key

[2]See [13] for a discussion on using an MDC and encryption to provide integrity and authenticity.

approach of SiRiUS, i.e., accessing a data-base to get authenticated public data and handing a secret key to each user in advance.

3.3 Blom's Method

As an alternative to the Leighton-Micali scheme, we suggest using Blom's method [2]. His scheme enables any two users that hold secret pre-distributed key information to establish a common symmetric-key. However, the scheme presents a new parameter k that determines the size of an adversary coalition that can break the scheme. This coalition size k exhibits a trade-off with the size of the secret that each user stores. Here is a short description of Blom's scheme:

1. A trusted agent generates a random secret $k \times k$ symmetric matrix D over a finite field $\mathbb{F}_q$.
2. Each user i is assigned a public identification number $s_i \in \mathbb{F}_q$ for $i = 1, \ldots, n$.
3. Each user i is assigned a secret vector $\mathbf{c}_i$ of length k:
$$\mathbf{c}_i = D \cdot [s_i^0, \ldots, s_i^{k-1}]^T$$
4. To establish the key $K_{i,j} = K_{j,i}$ user i uses j's identity s_j to calculate:
$$K_{i,j} = [s_j^0, \ldots, s_j^{k-1}] \cdot \mathbf{c}_i$$

The parameter k that determines the size of the secret vector $\mathbf{c}_i$ is the number of users that have to collude in order to compromise the keys of all n users. In [3], Blundo et al. provide a generalization of Blom's scheme for the establishment of group keys instead of pair-wise keys.

If such a threshold on the maximal size of an adversary coalition can be accepted, then we can achieve a more efficient scheme than Leighton-Micali. Consider applying the same technique to encrypt the file keys *FEK* and *FSK* as described above but with keys between the owner and the readers and writers that are evaluated using Blom's scheme. If the identification number of each user is known, then there is no need for any public database. This result resembles the use of Identity Based Encryption (IBE) referenced in SiRiUS, but with symmetric key efficiency.

4. INTEGRITY AND ACCESS CONTROL

4.1 Public-key Signatures

The most frequent public-key operation in our referenced systems [5, 7, 15] is a signature on the hash of a file (or the root of a hash tree of a file). This signature provides both file integrity verification and proof that the file was written by a user that has a write permission. Thus, the asymmetry between the private and public key distinguish between readers (that only have the public key) and writers (that only have the private key). Data integrity can be also attained by using a symmetric MAC key. However, the reader-writer distinction cannot be achieved with such a MAC, since all the parties that hold the MAC key can calculate it. Another property of a public-key signature is the notion of *non-repudiation*. We assume that non-repudiation is less relevant in file systems and thus we ignore it. In this section we suggest two schemes to replace public-key signatures with symmetric key techniques, and consider the relevant trade-offs between the two options.

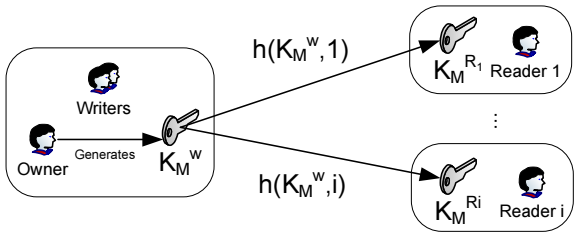

Figure 2: MAC Keys

4.2 Using MACs

Our first scheme uses MACs. The scheme is most suitable for the scenario of few readers and many writers.

If all the users that share a file have write permission, a MAC with a single shared key is sufficient, since user has equal right to change the file. Now assume that we add a single user with a read-only permission. We want this reader to be able to verify the file integrity against an external adversary, yet to prevent him from changing the file without being detected by the writers. To do this, we require each writer to calculate two MACs on every file update; one with a key shared only by the writers, and another with a key that is common to all the writers and the single reader. The reader can change the second MAC, since he holds the key, but this will buy him nothing as he is the only user that checks this MAC. The reader cannot modify the first MAC, since only the writers hold its key.

To extend the scheme to support more readers, we require the writers to calculate a different MAC for *every* reader. For each of these MACs the writer uses a designated key known only to the writers and the appropriate reader.

This approach encounter three drawbacks: (i) key management, (ii) time added to each write operation to calculate all the MACs, and (iii) the overall space consumed.

For the problem of key management we suggest the following scheme:

- During file creation the owner generates a random *file master MAC key*, denoted as K_M^w, that is handed only to the writers.
- Each reader receives a private *file reader MAC key*, denoted as $K_M^{R_i}$, where i is the reader's identification number.
- The keys of the readers are *derived* from the master key using a one-way function, i.e., $K_M^{R_i} = h(K_M^w, i)$.

Thus, there is no need to store multiple reader keys. When the file is updated by one of the writers, he can derive all file reader MAC keys, on the fly, from the file master MAC key. The scheme is illustrated in Figure 2.

A second drawback of this scheme is that the time overhead grows with the number of readers of the file. The overhead includes the time needed to derive the readers' MAC keys from the file master MAC key, and for each update the time required to calculate all the MACs. To lower the first overhead we suggest to derive the reader keys at file opening. The keys can be saved locally as long as the file is open. The MAC calculations can be made more efficient with the *"hash and MAC"* technique used in NASD [21]: instead of calculating the keyed MAC over the entire file,

several message digests are first calculated then the keyed MAC is calculated on much smaller amount of data.

The third drawback of this scheme is space. We argue that for a very small number of readers the space overhead can be similar to a public-key signature requiring about few hundreds of bytes. It becomes a significant problem as the number of readers grows.

To address the problems that result from a large number of readers we suggest using a combinatorial approach presented by Pinkas et al. [4], where the case of multicast authentication is discussed. Instead of adding another key for any additional user, the owner derives a *fixed* number of keys from his file master MAC key. These keys will be later used by the writers to calculate a *set* of MACs using each one of the keys. The owner gives each reader a *subset* of the derived keys. To verify the file integrity the reader calculates the MACs using his subset of keys and compares them to the corresponding MACs. However, this method allows a group of malicious readers to collude and share their keys. Therefore the security of the rest of the readers is probabilistic and depends on the size of the set of keys, the size of the subset of keys handed to each user, and the coalition size. Pinkas et al. also suggest to reduce the overall MACs size by taking only a few bits of each MAC, which also effect the overall forging probability.

¿From the results presented in [4] it can be seen that for a set of approximately 200 keys, with a reader subset of about 20 keys, taking 10 bits from each MAC, we can provide security of 10^{-3} against coalitions of up to 10 corrupted readers. The performance is about 50 times better than computing an RSA signature. Signature verification time is about the same as RSA verification if the public exponent $e = 3$, but about 500 times better than DSS signature verification. The size of the overall MACs is less than twice RSA signature size.

4.3 Using One-Time Signatures

A different approach is to look for an efficient signature scheme that presents better performance than regular public-key schemes. An approach that was used for multicast authentication [17] is to use one-time signatures that are based on symmetric-key primitives such as one-way hash functions. Here we introduce a scheme that is useful for scenarios where the file's permission profile consists of one publisher and many readers.

A *one-time signature* is based on a set of public *commitments* to secrets that the signer randomly generates. Some of the secrets are exposed according to the message to be signed. These secrets serve as a signature and can be validated against the public commitments. However, each set of such committed secrets can be used to sign only one (or a few) messages. In contrast to MACs, one-time signatures provide the asymmetry between signer and verifier which can give us the ability to distinguish readers from writers. One-time signatures also provide non-repudiation. The most well known one-time signature scheme is that of Merkle [14].

The main drawback of a one-time signatures scheme is clearly its "one-time-ness": it requires a mechanism to commit a large number of signatures in advance yet leaving this number finite. To handle a large number of commitments efficiently, Merkle [14] introduced the concept of a *hash tree*, in which the root serves as a commitment to all the leaves of the tree. Another challenge of one-time signature schemes is the size of the signatures that can be 10 times larger than public-key signatures.

In [16], we provide a practical analysis for using one-time signatures. We show that a commitment for 2^{20} signatures can provide signatures that are about 5 times faster than the equivalent RSA scheme, and uses space of few kilobytes to maintain the current state of the signer.

For the scenario of a publisher that serves a large number of read-only users, this approach can be very efficient. The publisher, being the owner of the file, gives each reader the root of the hash tree in an authenticated manner, similar to handing the file signature public-key in SiRiUS. Whenever the one-time signatures run out, the owner generates a new tree of commitments and publish the new root[3].

If multiple writers are required, we argue that this method becomes difficult to implement as the internal state of the scheme (hash tree representation and the number of the current signature) must be shared securely between the owner and the writers.

5. REVOCATION WITHOUT RE-ENCRYPTION

Users revocation requires the change of the file's access permissions. *Lazy revocation* avoids re-encryption of the file with a new key until the next update is performed, thus allowing the revoked users to read the file until the update event occurs. In Plutus [7] a technique of *key rotation* is introduced for convenient implementation of lazy revocation. A dedicated private key is used by the owner to generate a new file encryption key from the current one. All other users that are handed this new key can find the previous key using the dedicated public key.

A symmetric key substitute that is mentioned in Plutus is to use the concept of a hash chain (see [8]). The hash chain provides the same properties as the public-key rotation with better efficiency. However, it introduces a limit on the number of revocations. In this method, the symmetric file encryption keys are generated by the owner from an initial random key using a one-way hash function such that $K_i = h(K_{i+1})$, where K_i denotes the file encryption key after the i-th revocation. The first key to use, K_0, is the last one on the hash chain. When revocation occurs, the owner provides users with the next key, from which they can easily find the previous one using a hash computation, as long as the file does not change.

To handle the hash chain, the owner can store the whole chain or only the initial key as part of the meta-data of the file. Since the scheme has a limited number of revocations that equals the length of the hash chain, we want the chain to be reasonably long. However, this leads to either very large space consumption or to an inefficient computation done by the owner each time the next key in the chain has to be evaluated. To address this problem and thus make this preposition practical, we suggest using the *fractal hash chain traversal* presented by Jakobsson in [6]. This method allows the owner to find the next key using $O(\log n)$ hash operations where storing only $O(\log n)$ values out of the whole chain. Note, though, that the whole chain must be calculated by the owner when the file is created.

[3]The new root can be signed using the last signature of the current hash tree.

6. CONCLUSIONS

In this work we showed that secure untrusted storage with encryption of data at rest and cryptographic access control can be achieved using efficient symmetric key primitives. We demonstrated several promising methods for key distribution, integrity, access control and revocation. Our suggestions strictly adhere to the assumption that the file server is untrusted. We believe that since our methods only use symmetric-key cryptography, they will exhibit a lower security overhead than earlier systems.

Our design is based on the pre-key distribution schemes of Leighton-Micali or Blom for key distribution, and on MACs or one-time signatures for data integrity and reader/writer distinction. We believe that our design is practical, at least for some secure file-systems scenarios. However, a real performance evaluation requires the implementation of this design within a file system, and is left for future work.

7. REFERENCES

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

[2] R. Blom. An optimal class of symmetric key generation systems. In *EUROCRYPT*, pages 335–338, 1984.

[3] C. Blundo, A. D. Santis, A. Herzberg, S. Kutten, U. Vaccaro, and M. Yung. Perfectly-secure key distribution for dynamic conferences. In *CRYPTO*, pages 471–486, 1992.

[4] R. Canetti, J. A. Garay, G. Itkis, D. Micciancio, M. Naor, and B. Pinkas. Multicast security: A taxonomy and some efficient constructions. In *INFOCOM*, pages 708–716, 1999.

[5] E.-J. Goh, H. Shacham, N. Modadugu, and D. Boneh. SiRiUS: Securing remote untrusted storage. In *NDSS*. The Internet Society, 2003.

[6] M. Jakobsson. Fractal hash sequence representation and traversal. In *IEEE International Symposium on Information Theory*, 2002.

[7] M. Kallahalla, E. Riedel, R. Swaminathan, Q. Wang, and K. Fu. Plutus: Scalable secure file sharing on untrusted storage. In *Proceedings of the FAST '03 Conference on File and Storage Technologies*, 2003.

[8] L. Lamport. Password authentification with insecure communication. *Commun. ACM*, 24(11):770–772, 1981.

[9] F. T. Leighton and S. Micali. Secret-key agreement without public-key cryptography. In D. R. Stinson, editor, *CRYPTO*, volume 773 of *Lecture Notes in Computer Science*, pages 456–479. Springer, 1993.

[10] J. Li, M. N. Krohn, D. Mazières, and D. Shasha. Secure untrusted data repository (SUNDR). In *OSDI*, pages 121–136, 2004.

[11] D. Mazières, M. Kaminsky, M. F. Kaashoek, and E. Witchel. Separating key management from file system security. In *Proceedings of the 17th ACM Symposium on Operating System Principles*, pages 124–139, 1999.

[12] D. Mazières and D. Shasha. Don't trust your file server. In *HotOS*, pages 113–118. IEEE Computer Society, 2001.

[13] A. J. Menezes, P. C. van Oorschot, and S. A. Vanston, editors. *Handbook of Applied Cryptography*. CRC Press, 1996.

[14] R. C. Merkle. A digital signature based on a conventional encryption function. In C. Pomerance, editor, *CRYPTO*, volume 293 of *Lecture Notes in Computer Science*, pages 369–378. Springer, 1987.

[15] E. L. Miller, D. D. E. Long, W. E. Freeman, and B. Reed. Strong security for network-attached storage. In *Proceedings of the FAST '02 Conference on File and Storage Technologies*, pages 1–13, 2002.

[16] D. Naor, A. Shenhav, and A. Wool. One-time signatures revisited: Have they become practical? Manuscript, 2005.

[17] A. Perrig. The BiBa one-time signature and broadcast authentication protocol. In P. Samarati, editor, *Proceedings of the 8th ACM Conference on Computer and Communications Security*, pages 28–37, Philadelphia, PA, USA, Nov. 2001. ACM Press.

[18] E. Riedel, M. Kallahalla, and R. Swaminathan. A framework for evaluating storage system security. In *Proceedings of the FAST '02 Conference on File and Storage Technologies*, pages 15–30, 2002.

[19] A. D. Rubin. Kerberos versus the Leighton-Micali protocol. *Dr. Dobb's Journal of Software Tools*, 25(11):21–22, 24, 26, Nov. 2000.

[20] P. Stanton. Securing data in storage: A review of current research. *CoRR*, cs.OS/0409034, 2004.

[21] D. Tygar, G. Gibson, and H. Gobioff. Security for network attached storage devices. Technical Report CMU-CS-97-185, Carnegie Mellon University, October 1997.