Tahoe – The Least-Authority Filesystem

Zooko Wilcox-O'Hearn
allmydata.com
555 De Haro St.
San Francisco, CA 94107
zooko@zooko.com

Brian Warner
allmydata.com
555 De Haro St.
San Francisco, CA 94107
warner@allmydata.com

ABSTRACT

Tahoe is a system for secure, distributed storage. It uses capabilities for access control, cryptography for confidentiality and integrity, and erasure coding for fault-tolerance. It has been deployed in a commercial backup service and is currently operational. The implementation is Open Source.

Categories and Subject Descriptors: D.4.6 [**Security and Protection**]: Access controls; E.3 [**Data Encryption**]: Public key cryptosystems; H.3 [**Information Systems**]: Information Storage and Retrieval; E.4 [**Coding and Information Theory**]: Error control codes

General Terms: Design, Human Factors, Reliability, Security

Keywords: capabilities, fault-tolerance, open source, peer-to-peer

1. INTRODUCTION

Tahoe is a storage grid designed to provide secure, long-term storage, such as for backup applications. It consists of userspace processes running on commodity PC hardware and communicating with one another over TCP/IP. Tahoe was designed following *the Principle of Least Authority* [21] – each user or process that needs to accomplish a task should be able to perform that task without having or wielding more authority than is necessary.

Tahoe was developed by allmydata.com to serve as the storage backend for their backup service. It is now in operation and customers are relying on the Tahoe grid for the safety of their data. Allmydata.com has released the complete Tahoe implementation under open source software licences [1].

The data and metadata in the filesystem is distributed among servers using erasure coding and cryptography. The erasure coding parameters determine how many servers are used to store each file – denoted N, and how many of them are necessary for the file to be available – denoted K. The default settings for those parameters, and the settings which are used in the allmydata.com service, are $K = 3$, $N = 10$, so each file is shared across 10 different servers, and the correct function of any 3 of those servers is sufficient to access the file.

The combination of cryptography and erasure coding minimizes the user's vulnerability to these servers. It is an unavoidable fact of life that servers can fail, or can turn against their clients. This can happen if the server is compromised through a remote exploit, subverted by insider attack, or if the owners of the server are required by their government to change its behavior, as in the case of Hushmail in 2007 [29].

Tahoe's cryptography ensures that even if the servers fail or turn against the client they cannot violate confidentiality by reading the plaintext, nor can they violate integrity by forging file contents. In addition, the servers cannot violate freshness by causing file contents to *rollback* to an earlier state without a collusion of multiple servers.

2. ACCESS CONTROL

Tahoe uses the *capability access control model* [6] to manage access to files and directories. In Tahoe, a capability is a short string of bits which uniquely identifies one file or directory. Knowledge of that identifier is necessary and sufficient to gain access to the object that it identifies. The strings must be short enough to be convenient to store and transmit, but must be long enough that they are unguessable (this requires them to be at least 96 bits).

Such an access scheme is known as "capabilities as keys" or "cryptographic capabilities" [22]. This approach allows fine-grained and dynamic sharing of files or directories.

The Tahoe filesystem consists of files and directories. Files can be mutable, such that their contents can change, including their size, or immutable, such that they are writable only once.

Each immutable file has two capabilities associated with it, a *read capability* or *read-cap* for short, which identifies the immutable file and grants the ability to read its content, and a *verify capability* or *verify-cap*, which identifies the immutable file and grants the ability to check its integrity but not to read its contents.

For mutable files, there are three capabilities, the *read-write-cap*, the *read-only-cap*, and the *verify-cap*.

Users who have access to a file or directory can delegate that access to other users simply by sharing the capability. Users can also produce a verify-cap from a read-cap, or produce a read-only-cap from a read-write-cap. This is called *diminishing* a capability.

3. ERASURE CODING

All data is erasure coded using Reed-Solomon codes over $GF(2^8)$ [28]. When writing, a writer chooses erasure coding parameters N – the total number of shares that will be written – and K – the number of shares to be used on read.

Using this codec allows an efficient byte-wise implementation of encoding and decoding, and constrains the choice of erasure coding parameters to be $1 <= N <= 256$ and $1 <= K <= N$.

Compared to erasure codes such as *Tornado Codes* [5], Reed-Solomon codes offer optimal storage overheard – exactly b blocks of erasure code data (plus metadata – the index number of each block) are necessary to decode a b-block file. On the other hand, Reed-Solomon codes suffer from asymptotically worse computational complexity – the time to encode or decode a file with a Reed-Solomon codec is in the worst case proportional to N^2 where N is the number of erasure code blocks. Other codes offer asymptotically faster encoding and decoding – $O(N)$ computational complexity – but they impose storage overhead – more than b blocks are necessary to reconstruct a b block file. Also, many of these faster codes are patented.

In Tahoe we have learned that since K and N are small and since our implementation of Reed-Solomon ("zfec") is fast [25], the measured computational cost of erasure coding is low, and in fact is lower than the cost of the encryption and secure hash algorithms.

4. CRYPTOGRAPHY

4.1 Requirements

Tahoe is designed to run in software without requiring "Trusted Computing" (also called "Treacherous Computing" [2]) hardware. Therefore, the only robust way to constrain users or administrators from certain behavior such as unauthorizedly reading or altering files is to make such behavior require secrets, and to withhold those secrets from people who are not authorized to perform those behaviors. Thus, cryptography.

In this section we will describe the access control requirements of the different kinds of capabilities and how Tahoe satisfies those requirements using cryptographic techniques.

We want a verify-cap to enable its wielder to check the integrity of a file, but not to enable its wielder to learn the file's plaintext. This way a user can delegate to some other entity (such as a backup service) the job of checking that a file is still correctly stored, without sacrificing confidentiality.

A read-only-cap to a file has to give the ability to read the plaintext of the file and also to verify its integrity. We also require that a holder of a read-only-cap to a file is able to diminish it to produce a verify-cap to the same file.

A read-write-cap is meaningful only for mutable files. A read-write-cap grants its wielder the ability to write new contents to a file, and to produce a read-only-cap to the same file.

It is essential that these powers can not be gained by someone who has not been granted the appropriate capability. In particular, knowing a read-only-cap to a mutable file must not allow a user to change the contents of that file. This imposes an asymmetric cryptographic requirement on mutable file capabilities: users who hold a read-only-cap but not a read-write-cap must be able to check the integrity of the file

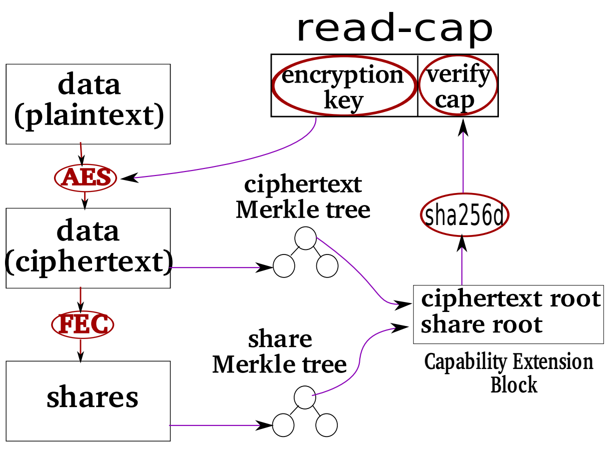

Figure 1: immutable file

without being able to produce texts which would pass the integrity check. This excludes symmetric integrity primitives such as *Message Authentication Codes* [19], and requires us to use the more computationally expensive *digital signatures* [19].

An additional requirement that we impose on read-only-caps and read-write-caps is that they are short enough to be conveniently used by humans, for example by being embedded in URLs. This requirement is not common in cryptography, and satisfying it turns out to require some careful cryptographic constructions.

4.2 Cryptographic Preliminaries

In the description that follows a "secure hash", denoted H, always means *SHA256d* [8]. $SHA256d(x)$ is equal to $SHA256(SHA256(x))$. This construction prevents length-extension attacks [26].

In addition, each time we use a secure hash function for a particular purpose (for example hashing a block of data to form a small identifer for that block, or hashing a master key to form a sub-key), we prepend to the input a tag specific to that purpose. This ensures that two uses of a secure hash function for different purposes cannot result in the same value. This is necessary to ensure that the root of a Merkle Tree can match at most one file, and it is a good hygienic practice in any case [15] [23].

"Merkle Tree" always means a binary Merkle Tree [20] using SHA256d as its secure hash, using one tag for the secure hash function to generate an internal node from its children and another tag for the secure hash function to generate a leaf from a segment of the file that the tree covers.

"Encrypt" always means encrypt using *AES-128* in *CTR mode* [18].

4.3 Immutable Files

An immutable file (Figure 1) is created exactly once and can be read repeatedly. To create an immutable file, a client chooses a symmetric encryption key, uses that key to encrypt the file, chooses erasure coding parameters K and N, erasure codes the ciphertext into N shares, and writes each share to a different server.

Users may opt to derive the encryption key from the secure hash of the plaintext of the file itself, a technique known as *convergent encryption* [7]. Convergent encryption allows two users to produce identical ciphertext in two independent writes of the same file. Since Tahoe coalesces storage of identical ciphertexts this can be a space savings. However, use of convergent encryption can undermine confidentiality in some cases [30], and a measurement of files stored on allmydata.com showed less than 1% of files (by aggregate file size, not file count) are shared between users. To protect confidentiality, Tahoe mixes in an *added convergence secret* which limits the scope of convergent encryption to users who share the same convergence secret.

For immutable files, the verify-cap is derived from the ciphertext of the file using secure hashes. We use a combination of two different secure hash structures to defend against two different problems that could arise in file validity.

The first problem is that if the integrity check fails, the client needs to know which erasure code share or shares were wrong, so that it can reconstruct the file from other shares. If the integrity check applied only to the ciphertext, then the client wouldn't know which share or shares to replace. Instead we compute a Merkle Tree over the erasure code shares. This allows the client to verify the correctness of each share.

However, this Merkle Tree over the erasure code shares is not sufficient to guarantee a one-to-one mapping between verify-cap and file contents. This is because the initial creator of the immutable file could generate some erasure code shares from one file and other erasure code shares from another file, and then including shares from both files in the set of N shares covered by the Merkle Tree. In this case, the reader would see a different file depending on which subset of the shares they used to reconstruct the file [10].

In order to ensure that there is a one-to-one mapping from verify-cap to file contents, we construct another Merkle Tree over the ciphertext itself. Since it is a Merkle Tree instead of a hash of the entire file contents, the reader can verify the correctness of part of the file without downloading the entire file. This allows streaming download, such as to watch a movie while it is downloading. It also allows the client to perform a simple Proof of Retrievability protocol [14] by downloading and verifying a randomly chosen subset of the ciphertext (this protocol is not deployed in the current release of Tahoe, but is a work in progress).

The roots of the two Merkle Trees are kept in a small block of data, a copy of which is stored on each server next to the shares. This structure is called the "Capability Extension Block" because its contents are logically part of the capability itself, but in order to keep the capability as short as possible we store them on the servers and fetch them when needed.

The verify-cap is the secure hash of the Capability Extension Block.

The read-cap to an immutable file is the the verify-cap and the symmetric encryption key.

4.4 Mutable Files

Each mutable file (Figure 2) is associated with a unique RSA key pair [27]. Authorized writers have access to the private key (the *signing key*, or SK) so that they can make digital signatures on the new versions of the file that they write.

However, RSA keys are too large for humans to casually manipulate such as by embedding them in URLs – we use 2048-bit (256 byte) RSA keys. In order to have small read-caps and write-caps we create small (16-byte) secrets cryptographically linked to the RSA keys in the following way.

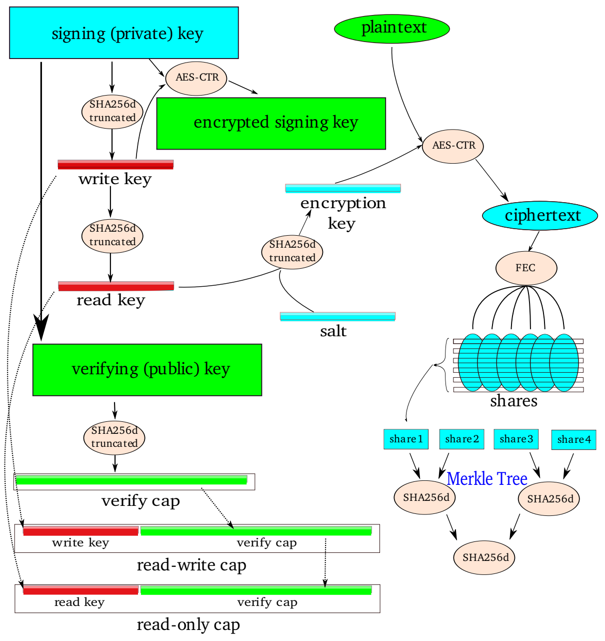

Figure 2: mutable file

To write to a mutable file, a client first computes the *write key*, $WK = Htrunc(SK)$, where $Htrunc$ denotes SHA256d with its output truncated to 16 bytes. It then computes the *read key*, $RK = Htrunc(WK)$. Therefore anyone who has the write key can easily compute the read key, but knowledge of the read key does not confer useful information about the write key.

A client which is writing a file generates a new random 16 byte *salt* and computes the *encryption key* with $EK = H(RK, salt)$.

Using this encryption key, the client encrypts the plaintext to produce the ciphertext, erasure codes the ciphertext to form shares, and writes each share to a different server, computing a Merkle Tree over the shares: $ciphertext = Encrypt(EK, plaintext)$.

Using the signing key SK, the client signs the root hash of the Merkle Tree, as well as the salt, the erasure coding parameters, the filesize, and a sequence number indicating that this version of the file contents supercedes the previous one.

For mutable files, verification involves checking a digital signature. A copy of the RSA public key – the *verifying key* or VK – for each file is stored in the clear along with the erasure coded shares of the file on each server.

The verify-cap must provide the ability to check that the verifying key which is stored on the storage server is the right one. This is accomplished by making the verify-cap be the secure hash of the verifying key: $VC = H(VK)$.

The read-only-cap to a mutable file includes VC, and in addition contains the read key: $RC = VC, RK$.

The read-write-cap to a mutable file includes VC, and in addition contains the write key: $WC = VC, WK$. The holder of a read-write-cap needs to get the ability to produce digital signatures on new versions of the file. This is accomplished by storing the RSA signing key SK, *encrypted* *with* WK on the servers. This somewhat unusual arrangement means that the RSA signing key is encrypted with the truncation of the secure hash of itself.

This is an instance *key-dependent encryption* [3] [12] [11], which is not as well-studied as we would like. However, we think that it would be surprising if there were a weakness in this construction which did not imply a weakness in the underlying standard primitives – RSA, AES-128, and SHA256. See 6.1 for a proposal to replace this nonstandard cryptographic construction with a different nonstandard cryptographic construction with better performance properties.

WK serves as both integrity check on SK (the client verifies that the $Htrunc(SK) = WK$) and access control to the RSA signing key (the client uses WK to decrypt SK in order to use SK to produce digital signatures). It also serves to provide writers with access to RK, since $RK = Htrunc(WK)$.

4.5 Directories

Capabilities can be used as the child links in directories. A directory could simply be a mutable file which contains a set of tuples of (child name, capability to child). If directories were implemented like this, then anyone given read access to the directory would be able to learn the child names and corresponding capabilities, thus enabling them to follow the links. Anyone given write access to the directory would be able to change the set of children.

Tahoe doesn't use this scheme though, because we want something in addition, the property of *transitive read-only* – users who have read-write access to the directory can get a read-write-cap to a child, but users who have read-only access to the directory can get only a read-only-cap to a child. It is our intuition that this property would be a good primitive for users to build on, and patterns like this are common in the capabilities community, e.g. the "sensory keys" of KeyKOS [13].

In order to implement transitive read-only we include two slots for the caps for each child – one slot for a read-write-cap and one for a read-only-cap. All of the directory information is of course encrypted as usual when the directory is written as the contents of a Tahoe mutable file, but in addition the read-write-caps to the children are *super-encrypted* – they are encrypted separately by the writer before being stored inside the mutable file.

5. FRESHNESS OF MUTABLE FILES AND DIRECTORIES

We've shown how our use of cryptography makes it impossible for an attacker – even one who controls all of the servers – to violate the confidentiality of the user, which would allow unauthorized people to learn the contents of files or directories. We've also shown how our use of cryptography makes it impossible for an attacker – even one who controls all of the servers – to forge file or directory contents. However the digital signature does not prevent failing or malicious servers from returning an earlier, validly signed, version of the file contents.

In order to make it more difficult for servers to (either accidentally or maliciously) serve a stale version of a mutable file, Tahoe leverages the erasure coding scheme. Each version of a mutable file has a sequence number. The read protocol proceeds in two phases. In the first phase, the reader requests the metadata about a share from each of $K + E$ servers. K is the number of shares necessary to reconstruct the file, and E is a constant configured into the Tahoe client for the number of "extra" servers to read from. In the current Tahoe client K is set to 3 and E is set to K.

If after querying $K + E$ servers, the client learns that there are at least K shares of the highest numbered version available, then the client is satisfied and proceeds to the second stage. If it learns about the existence of a higher version number but does not find at least K shares for that version, then the client will continue query more servers for their metadata until it either locates at least K shares of the newest known version or it runs out of servers to query.

Once the first phase has ended, either because the client ran out of servers to ask or because it learned about at least K shares of the newest version that it learned about, then it proceeds to the next phase and downloads the shares.

This scheme reduces the chance of incurring a stale read, either accidental or malicious, provided that enough of the servers are honest and have fresh data. Tahoe makes no attempt to enforce other consistency properties than freshness of mutable files. For example, it makes no attempt to enforce linearization of operations spanning more than a single file, as in Byzantine Fault-Tolerant filesystems such as [17] [9].

6. FUTURE WORK

6.1 ECDSA and Semi-Private Keys

A performance problem with the RSA-based write capabilities is the generation of a new RSA key pair for each mutable file. On our modern commodity PC hardware, the process of generating a new 2048-bit RSA key pair takes between 0.8 seconds and 3.2 seconds.

An alternative digital signature scheme which offers fast key generation is DSA [24]. Creating a new DSA key is as easy as choosing an unguessable 96-bit secret s, using a Key Derivation Function to produce a 192-bit private key $x = KDF(s)$ and then computing g^x in some appropriate group for some generator g. On our hardware, generating a new DSA key pair takes less than one millisecond.

Another potential improvement would be to use *Elliptic Curve Cryptography* (or *ECC*) instead of traditional integer-based cryptography for the DSA group. ECC with similar conjectured security to our 2048-bit RSA keys would require only 192-bit ECC keys [16]. Such short keys could be suitable for including directly in capabilities.

However, observe that even with the use of short ECC keys, we cannot simply make the write-cap be the private key and make the verify-cap be the public key, because then the read-cap would be unspecified. We require three, not two, levels of privilege separation – write-caps, read-caps, and verify-caps. We require that the lower levels of privilege can be easily computed from the higher ones (to efficiently diminish capabilities) but not, of course, the other way around, and we require that the write-cap enable its holder to create digital signatures and the read-cap enable its holder to check the validity of such signatures.

This could be accomplished by including a separate secret in the read-cap, but then the read-cap would be too large to casually embed into URLs. A 192-bit secret, encoded in a suitable URL safe base-62 encoding, looks like this:

```
http://hostname/2DM7XkGnnlFjIg8ozYy2pLo4qrkh6EY71
```

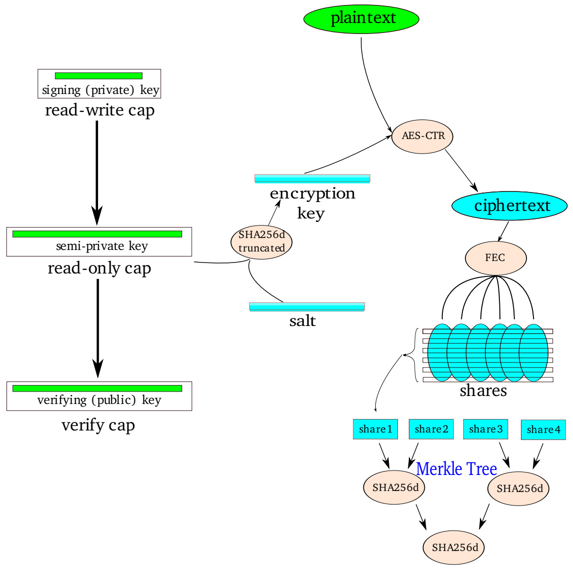

Figure 3: mutable file with semi-private key

Two cryptographic values (a 128-bit encryption secret and a 192-bit hash value) would look like this:

`http://hostname/ArpPguJIBJwtvm1F5JaoR6pWVVTF4eiNa YWL8e5iF78rq2OAdzZL2z`

Therefore we propose a novel cryptographic construction, the *semi-private key*. A semi-private key is one which is more private than the public key, and less private than the private key, and which can be used to derive the public key in order to check signatures.

To implement semi-private keys in DSA, recall that the private key is $x = KDF(s)$ and the public key is the value g^x. Instead, let the private key be $x = KDF(s)$, let the semi-private key be g^x, let $y = H(g^x)$ and let the public key be g^{xy}. The signing key is therefore xy. (Note that it is possible to iterate this process to produce semi-semi-private keys and semi-semi-semi-private keys, for as many levels of privilege separation as desired.)

Using semi-private keys we can implement capabilities which are short, and which have a much simpler design: Figure 3.

Intuitively, it would be surprising to us if digital signatures using this scheme were vulnerable to an attack which did not also defeat DSA itself. It seems closely related to the recent formulation of "The One-Up Problem for (EC)DSA" [4]. However, we have not proven that this scheme is as secure as ECDSA so, like the key-dependent encryption scheme from section 4.4, we currently have only heuristic arguments to believe that this scheme is secure.

7. DEPLOYMENT

The first beta of Allmydata.com was released April 3, 2008 and the first product was shipped May 5, 2008. At the time of this writing, October 17, 2008, allmydata.com operates 64 servers, typically has about 340 clients connected at any time, and typically has about 300 users accessing the filesystem through a web front-end in a 24-hour period. There are about 6.5 million user files stored, with an aggregate size of about 9.5 TB. See Figure 4.

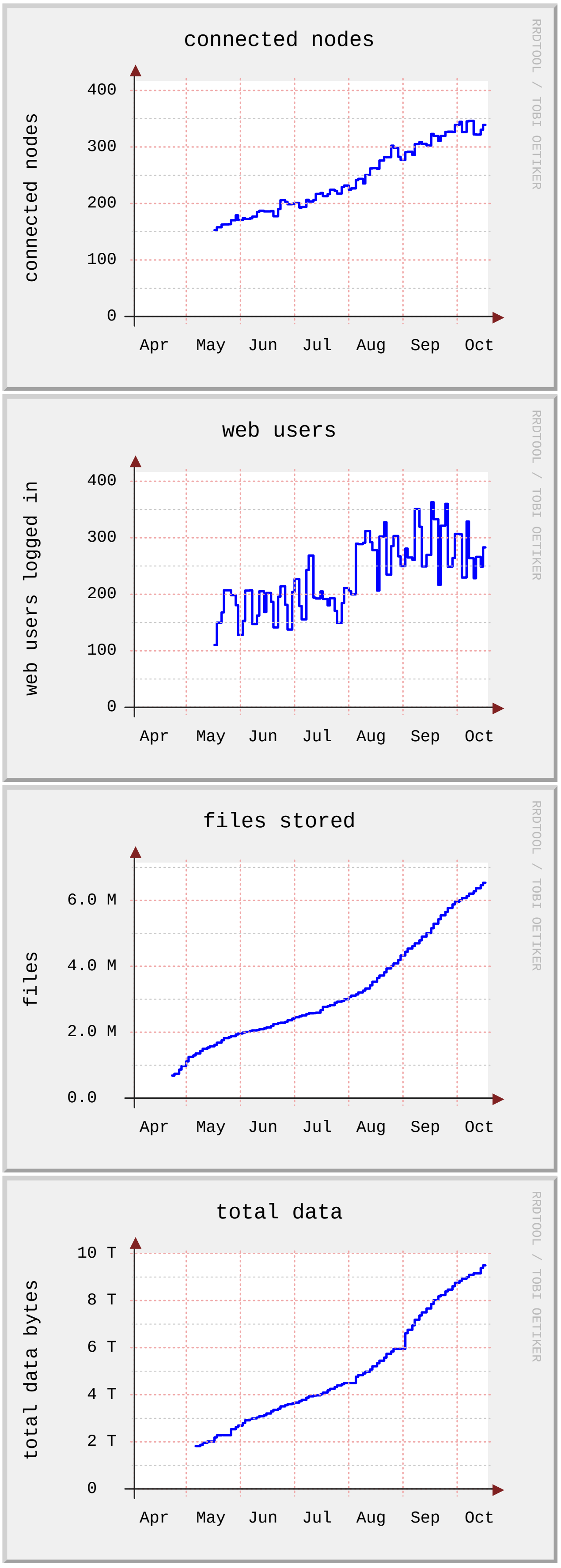

Figure 4: storage grid statistics for allmydata.com

8. CONCLUSIONS

Tahoe, The Least-Authority Filesystem, is a practical secure, decentralized filesystem, using several techniques which have previously been analyzed in the theoretical literature but not widely deployed. Tahoe's security is based on cryptographic capabilities for decentralized access control, which have proven to be flexible enough to serve our requirements so far. It uses Reed-Solomon codes for fault-tolerance, which have been characterized as being asymptotically inefficient in the literature but which are quite fast enough in practice. Some non-standard cryptographic constructions were required to make cryptographic capabilities short enough to fit conveniently into URLs.

Tahoe is open source (http://allmydata.org) and is currently in use in the allmydata.com backup service.

9. ACKNOWLEDGMENTS

Thanks to L. Amber Wilcox-O'Hearn, Kevin Reid, E. Dean Tribble, Ian Lynagh, and Niels Bakker for comments and suggestions. Thanks to Josh Wilcox for comments, suggestions, and diagrams. Thanks to Peter Secor for funding and managing this work, for comments, suggestions, and graphs.

Thanks to Nathan Wilcox, Drew Perttula, and Christian Grothoff for finding security flaws in earlier versions of the Tahoe Least-Authority Filesystem.

Thanks to allmydata.com for the development and open publication of this system.

10. REFERENCES

[1] allmydata.org web site, 2007.

[2] R. Anderson. 'trusted computing' frequently asked questions, 2003. [Online; accessed 26-May-2008].

[3] J. Black, P. Rogaway, and T. Shrimpton. Encryption-scheme security in the presence of key-dependent messages. In *In Selected Areas in Cryptography, volume 2595 of LNCS*, pages 62–75. Springer-Verlag, 2002.

[4] D. R. L. Brown. One-up problem for (ec)dsa. Cryptology ePrint Archive, Report 2008/286, 2008.

[5] J. W. Byers, M. Luby, M. Mitzenmacher, and A. Rege. A digital fountain approach to reliable distribution of bulk data. In *SIGCOMM*, pages 56–67, 1998.

[6] J. B. Dennis and E. C. V. Horn. Programming semantics for multiprogrammed computations. *Commun. ACM*, 26(1):29–35, 1983.

[7] J. R. Douceur, A. Adya, W. J. Bolosky, D. Simon, and M. Theimer. Reclaiming space from duplicate files in a serverless distributed file system. In *ICDCS '02: Proceedings of the 22 nd International Conference on Distributed Computing Systems (ICDCS'02)*, page 617, Washington, DC, USA, 2002. IEEE Computer Society.

[8] N. Ferguson and B. Schneier. *Practical Cryptography.* Wiley Publishing, Inc., 2003.

[9] G. R. Goodson, J. J. Wylie, G. R. Ganger, and M. K. Reiter. Efficient byzantine-tolerant erasure-coded storage. pages 135–144, 2004.

[10] C. Grothoff. Crisp advisory 2008-01: Uris do not refer to unique files in allmydata tahoe, July 2008.

[11] I. Haitner and T. Holenstein. On the (im)possibility of key dependent encryption. Cryptology ePrint Archive, Report 2008/164, 2008.

[12] S. Halevi and H. Krawczyk. Security under key-dependent inputs, 2007.

[13] N. Hardy. Keykos architecture. *SIGOPS Oper. Syst. Rev.*, 19(4):8–25, 1985.

[14] A. Juels and J. Burton S. Kaliski. Pors: proofs of retrievability for large files. In *CCS '07: Proceedings of the 14th ACM conference on Computer and communications security*, pages 584–597, New York, NY, USA, 2007. ACM.

[15] J. Kelsey, B. Schneier, and D. Wagner. Protocol interactions and the chosen protocol attack. In *In Proc. 1997 Security Protocols Workshop*, pages 91–104. Springer-Verlag, 1997.

[16] A. Lenstra. *Handbook of Information Security*, chapter Key Lengths. Wiley, 2004.

[17] J. Li and D. Mazières. Beyond one-third faulty replicas in byzantine fault tolerant systems. In *NSDI*. USENIX, 2007.

[18] H. Lipmaa, P. Rogaway, and D. Wagner. Comments to NIST concerning AES-modes of operations: CTR-mode encryption. In *Symmetric Key Block Cipher Modes of Operation Workshop*, Baltimore, Maryland, USA, 2000.

[19] A. J. Menezes, S. A. Vanstone, and P. C. V. Oorschot. *Handbook of Applied Cryptography.* CRC Press, Inc., Boca Raton, FL, USA, 1996.

[20] R. C. Merkle. A digital signature based on a conventional encryption function. In *CRYPTO '87: A Conference on the Theory and Applications of Cryptographic Techniques on Advances in Cryptology*, pages 369–378, London, UK, 1988. Springer-Verlag.

[21] M. S. Miller. *Robust Composition: Towards a Unified Approach to Access Control and Concurrency Control.* PhD thesis, Johns Hopkins University, Baltimore, Maryland, USA, May 2006.

[22] M. S. Miller, K. P. Yee, and J. Shapiro. Capability myths demolished. Technical report, Combex, Inc., 2003.

[23] R. Needham. Prudent engineering practice for cryptographic protocols. *IEEE Transactions on Software Engineering*, 22:122–136, 1996.

[24] NIST. The digital signature standard. *Commun. ACM*, 35(7):36–40, 1992.

[25] J. S. Plank, L. Xu, J. Luo, C. D. Schuman, and Z. Wilcox-O'Hearn. A performance evaluation and examination of open-source erasure coding libraries for storage. 2009.

[26] B. Preneel and P. C. V. Oorschot. Mdx-mac and building fast macs from hash functions. pages 1–14. Springer-Verlag, 1995.

[27] R. L. Rivest, A. Shamir, and L. M. Adelman. A method for obtaining digital signatures and public-key cryptosystems. Technical Report MIT/LCS/TM-82, MIT, 1977.

[28] L. Rizzo. the feasibility of software fec. Technical Report LR-970131, DEIT, 1997.

[29] Wikipedia. Hushmail — Wikipedia, the free encyclopedia, 2008. [Online; accessed 26-May-2008].

[30] Z. Wilcox-O'Hearn. convergent encryption reconsidered, 2008. [Online; accessed 18-Aug-2008].