AONT-RS:
Blending Security and Performance in Dispersed Storage Systems

Jason K. Resch
Development
Cleversafe, Inc.
222 S. Riverside Plaza, Suite 1700
Chicago, IL 60606
jresch@cleversafe.com

James S. Plank
EECS Department
University of Tennessee
203 Claxton Complex
Knoxville, TN 37996
plank@cs.utk.edu

Abstract

Dispersing files across multiple sites yields a variety of obvious benefits, such as availability, proximity and reliability. Less obviously, it enables security to be achieved without relying on encryption keys. Standard approaches to dispersal either achieve very high security with correspondingly high computational and storage costs, or low security with lower costs. In this paper, we describe a new dispersal scheme, called *AONT-RS*, which blends an All-Or-Nothing Transform with Reed-Solomon coding to achieve high security with low computational and storage costs. We evaluate this scheme both theoretically and as implemented with standard open source tools. AONT-RS forms the backbone of a commercial dispersed storage system, which we briefly describe and then use as a further experimental testbed. We conclude with details of actual deployments.

1 Introduction

Dispersed storage systems coalesce multiple storage sites into a collective whole. Files are decomposed into smaller blocks which are computationally massaged and then dispersed to the storage sites. When a client desires to read a file, it retrieves some subset of the blocks, which are combined to reconstitute the original file. Compared to traditional single-site storage systems, dispersed storage systems offer a variety of benefits. Multiple independent storage sites offer greater availability than a single site, since they have no single point of failure. When sites are physically distributed across a wide area, they offer physical proximity to distributed clients, which can improve performance and scalability. Finally, the massaging of data typically includes adding redundancy in the form of erasure codes or secret sharing, which improves reliability in the face of failures.

There have been many dispersed storage systems developed in the past ten years. Examples include storage systems such as Oceanstore [23], Pergamum [29], POTSHARDS [30], PASIS [9], Gridsharing [31], Glacier [11], Cleversafe [4] and Tahoe-LAFS [32] among others. Related to dispersed storage systems are distributed or peer-to-peer storage systems which use replication rather than coding to achieve reliability. Examples include LOCKSS [14], Google file system [8], Elephant [27], PAST [26] and BitTorrent [5].

A side benefit of dispersal is the ability to provide security without the use of encryption keys. The basic techniques are classics from computer science literature: Shamir's secret sharing [28] and Rabin's information dispersal based on non-systematic erasure codes [21]. Each technique is a (k,n) *threshold scheme*: The storage system transforms a file into n distinct blocks. A client or attacker must retrieve at least k of the n blocks to reconstruct the file. With fewer than k blocks, the client or attacker gets no information. Several of the abovementioned systems [9, 30, 31] use these techniques to achieve security by storing each of the n pieces at a different site, and assuming that an attacker will not be able to authenticate himself to at least k of them. This avoids encryption strategies which require the secure storage of encryption keys, a difficult and dangerous practice (see [30] for a thorough discussion of this problem).

Each technique achieves a different level of security with different performance and storage requirements. If the original file is b bytes in size, Shamir's scheme requires a total of nb bytes, while Rabin's requires $\frac{nb}{k}$. Shamir's requires more computation as well. To compensate for the extra storage and computation, Shamir's scheme is more secure, achieving information theoretic security. Rabin's security is far less, and would be unacceptable in many environments.

In this paper, we describe a further modification to Rabin's scheme that achieves improved computational performance, security and integrity. We achieve this by combining the All-Or-Nothing Transform (AONT) [24] with systematic Reed-Solomon erasure codes [13].

Hence, we call it *AONT-RS*. We describe the technique, evaluate it both theoretically and experimentally and detail how it fits into a commercial dispersed storage system. We conclude with some field data of actual deployments.

2 Dispersal Algorithms

At the heart of all (k,n) threshold schemes (which we heretofore call *dispersal algorithms*) is a matrix-vector product, illustrated in Figure 1. The data to be stored is broken into *words* or *elements* which are w bits in length. A *generator* or *dispersal* matrix G is created, which has n rows and k columns. This matrix is multiplied by a k-element vector D (called the *data* or *message*) to yield a n-element vector C called the *codeword*. Each element of the codeword is stored on a different storage node.

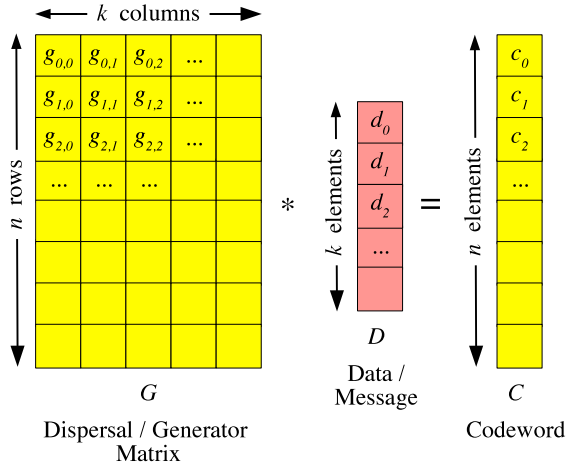


Figure 1: The central matrix-vector product for all dispersal algorithms.

The dispersal matrix is constructed so that all combinations of k rows yield invertible matrices. This gives us a technique to reconstruct D from any k surviving elements of the codeword: each row of G corresponds to the calculation of a codeword element. We create a new $k \times k$ matrix A from the rows of G that correspond to the k surviving elements. We invert A and multiply A^{-1} by the surviving elements to yield D. The construction of G guarantees that A is invertible.

So that elements may fit into computer words, it is convenient that w be a power of two. To achieve this, we employ *Galois Field* arithmetic, $GF(2^w)$, where addition is equal to bitwise exclusive-or (XOR) and multiplication is implemented in a variety of ways either in hardware or software. In this way, dispersal is simply a variant of the well known Reed-Solomon codes [13, 22]. A tutorial on implementing Reed-Solomon codes in this manner is available in [17], and a thorough discussion of implementing Galois Field arithmetic is provided in [10]. There is also a methodology that converts multiplications into XOR's described in [3]. There are open-source implementations of these codes and methodologies in [16, 20, 25, 36].

Shamir's secret sharing algorithm encodes w bits of data in d_0. The remaining elements of D are randomly chosen w-bit words. The matrix G is a Vandermonde matrix, where $g_{i,j} = i^j$, which guarantees that any k rows are invertible so long as $n \leq 2^w$ [28]. Thus, when one uses Shamir's algorithm on a b-byte file, the total storage requirement is nb bytes, and the act of encoding requires $O(knb)$ XOR and multiplication operations (we will characterize this further in Section 6 below). The security guarantees of Shamir's algorithm are very strong — even with an infinite amount of computing power, unless an attacker has possession of k words, he cannot determine anything about the initial data. Moreover, this is done without the necessity of storing encryption keys.

Rabin's information dispersal algorithm (IDA) weakens the security, but improves both storage efficiency and performance. Each element of D now contains a word of data. Thus the storage requirement is $\frac{nb}{k}$ bytes, improving both storage efficiency and encoding performance by a factor of k. Like Shamir, k elements of the codeword are required to reconstruct the original data. However, the security guarantees of Rabin are far less than Shamir. We will analyze this below in Section 5, but attackers looking for known or patterned data can find it more easily from elements of the codeword. To combat this problem, Rabin suggests a technique to generate the rows of G randomly, embed the row id's within each codeword element, then encrypt the codewords [21]. Unfortunately, this requires storing an external encryption key, which does not solve the main problem we wish to solve (providing security without securely storing encryption keys).

In 1993, Krawczyk proposed a blending of Rabin and Shamir, by encrypting the data with a key-based encryption algorithm, and then dispersing the encrypted data with an IDA and the key with a secret sharing scheme [12]. This is called *Secret Sharing Made Short (SSMS)*. Our dispersal algorithm, described in the next section, also enriches Rabin's IDA with security. Unlike SSMS, it does so without secret sharing, and with the integration of integrity checking for corruption.

3 A New Dispersal Algorithm: AONT-RS

We enrich Rabin's IDA in two ways. First, we employ a variant of Rivest's *All-or-nothing Transform (AONT)* as a preprocessing pass over the data [24]. The AONT

may be viewed as a $(s+1, s+1)$ threshold scheme. Data composed of s words of size w_A[1] is encoded into $s+1$ different words so that *none* of the original words may be decoded unless all $s+1$ encoded words are present, or an attacker possesses enough computing power to crack an encryption key. The key, however, is encoded with the data. If a file's size is b bytes, the performance of encoding is $O(b)$. The benefits of the AONT are:

- No external keys are necessary.
- Very little extra storage is required.
- The computational requirements of the attacker may be a parameter of the encoding.
- The performance is good.

The AONT works as follows. The data is composed of s words $d_0, \ldots, d_{s-1}$, each of which is w_A bits in length. A random key K is chosen, and each codeword c_i is calculated as:

$$c_i = d_i \oplus E(K, i+1),$$

where E is a key-based encryption algorithm such as AES [6]. A final codeword, c_k, is calculated to be a function of K and a hash of the other codewords. The AONT has *computational security,* which means that unless an attacker possesses all $s+1$ codewords or can guess K, the attacker cannot get information about *any* word or data. We will discuss this further in section 5 below.

We modify this scheme slightly. We add an extra word of data d_s, called a *canary* [2]. This word has a known, fixed value, which allows us to check the integrity of the data when it is decoded.

We generate $c_0, \ldots, c_s$ as described above and then calculate a hash h of the $s+1$ codewords using a standard hash algorithm such as SHA-256 [15] having an output at least as long as K. We then calculate a final block c_{s+1} as:

$$c_{s+1} = K \oplus h.$$

Our second modification of Rabin's IDA is to employ a *systematic* erasure code instead of a *non-systematic* one. A systematic code is defined to be one where the codeword contains the original elements of D. Without loss of generality, the first k elements of C are equal to the elements of D: $c_i = d_i$ for $0 \leq i < k$. This means that the first k rows of G compose a $k \times k$ identity matrix as pictured in Figure 2.

Employing a systematic erasure code instead of a non-systematic one (as in both the Shamir and Rabin algorithms) improves performance because it eliminates the need to encode the first k codewords. Since many systems use values of k that are large relative to n (e.g. POTSHARDS' evaluation uses a (3,5) Shamir scheme [30]) the savings during encoding with a systematic erasure code are substantial. Moreover, when decoding, codeword elements that are equal to data elements do not have to be decoded, which improves performance further.

[1]Since AONT-RS mixes AONT with dispersal, we differentiate its word size from the dispersal's word size using w_A instead of w.

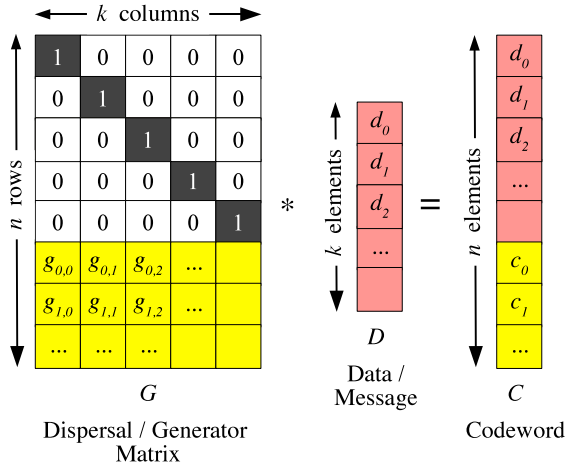


Figure 2: A systematic erasure code.

We call our dispersal technique *AONT-RS*, as it is a combination of the All-Or-Nothing Transform and Reed-Solomon coding. The intuition is that we use the AONT for security and the dispersal for availability, proximity and fault-tolerance. This is unlike Shamir, Rabin and SSMS which use dispersal to achieve both functions.

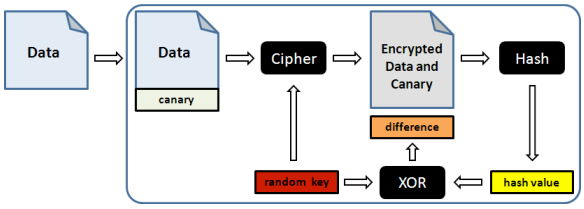


Figure 3: Encoding operation of AONT.

Several diagrams depict the operation of AONT-RS and interaction between AONT and Reed-Solomon coding. In Figure 3, data is processed by AONT. A canary is appended to the data, and the data and canary are encrypted with a random key. A hash value of the encrypted data is computed. The hash value and random key are then combined via bitwise exclusive-or to form a difference, which is appended to the encrypted data to form the AONT package.

Once processed by AONT, the result is treated as normal input to a systematic IDA, as depicted in Figure 4.

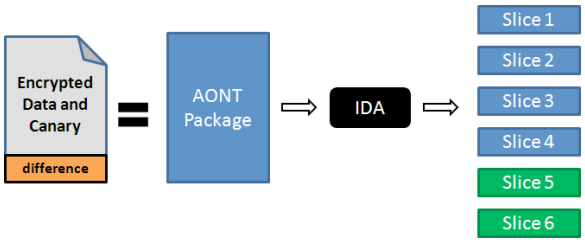


Figure 4: Dispersal of AONT package using a systematic IDA such as Reed-Solomon coding.

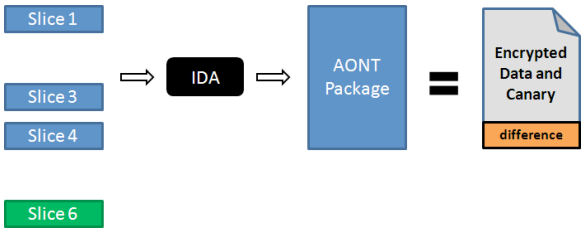


Figure 5: Recovering the AONT package from a threshold number of slices.

The IDA splits the input into k slices formed directly from the input and computes $n-k$ coding slices. Slices are then stored to separate locations.

At a future time, slices may be retrieved and used to recover the data. The first step in this process requires obtaining a threshold number of slices, as in Figure 5. Short of a threshold number of slices the entire AONT package cannot be recovered; there is not enough information contained in $m < k$ slices to yield the original input, whose length is k times the slice length. However, if one possesses any k of the slices, they may compute the original input to the IDA which in this case is the AONT package.

As shown in Figure 6, Reversing the AONT operation is trivial when one possesses the entire package. The first step is to compute the hash, h, of the encrypted data. Since the last block contains $K \oplus h$ and we know the hash value h, we may exclusive-or the last block with the hash to find $(K \oplus h \oplus h)$. Since $h \oplus h$ equals zero, the result is the random key K. The random key is then used to decrypt the encrypted data, and the canary is checked to detect corruption.

4 A Concrete Example

To help illustrate, we present a concrete example. Suppose we have a 4KB block of data, D that we wish to massage into 16 *slices* on 16 storage nodes so that we may reconstruct and verify the data so long as we possess any 10 slices.

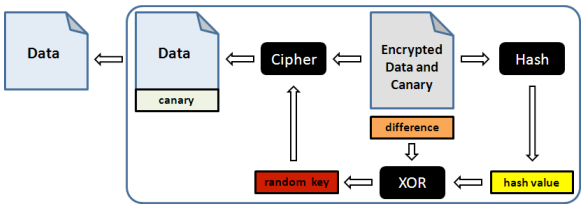


Figure 6: Restoring data from an AONT package.

Shamir: To apply Shamir's algorithm, we view the data as 4096 individual bytes, $d_0, \ldots, d_{4095}$. Each of the 16 slices $S_0, \ldots S_{15}$ will also be composed of 4096 individual bytes $s_{i,0}, \ldots, s_{i,4095}$ such that $s_{i,j}$ is a function of d_j and nine random bytes. Specifically,

$$s_{i,j} = d_j \oplus \sum_{x=1}^{9} (i+1)^x r_{j,x},$$

where $r_{j,x}$ is a random byte and arithmetic is over $GF(2^8)$. The total storage requirement is 64 KB.

Rabin: To apply Rabin, we pad D to be 4100 bytes and then partition it into ten data slices $DS_0, \ldots, DS_9$ of 410 bytes each. As with Shamir, we view each data slice DS_i to be composed of 410 individual bytes $DS_{i,0}, \ldots Ds_{i,409}$. We then calculate each of the 16 slices using Reed-Solomon coding on the individual bytes: [2]

$$s_{i,j} = \sum_{x=0}^{9} (i+1)^x d_{x,j}.$$

Again, arithmetic is over $GF(2^8)$. The total storage requirement is 16*410 = 6.41 KB.

SSMS: With SSMS, we select a random 16-byte encryption key and encrypt the data with an encryption algorithm such as AES. We then disperse it using Rabin and disperse the key using Shamir. The total storage requirement is 16*(410+16) = 6.65 KB.

AONT-RS: We will be adding 34 additional bytes to the data, and we will first view it as being composed of 257 16-byte words, $d_0, \ldots, d_{256}$, where the first 256 words are the original data. We set d_{256} to be a 16-byte canary value. We choose K to be sixteen random bytes and set each c_i to equal $d_i \oplus E(K, i+1)$ where E is a standard encryption algorithm. Next we calculate h to be a 16-byte hash of $c_0, \ldots, c_{256}$. Finally, we set c_{257} to equal $h \oplus K$. The last 2 bytes are immaterial – they are simply padding so that the data may be partitioned into ten equal slices. They could be used as additional canaries if desired.

[2] While Rabin does not use a Vandermonde matrix in [21], the matrix he employs has the same properties.

As with Rabin, we partition the 4130 bytes into ten data slices $DS_0, \ldots, DS_9$ of 413 bytes each. These will be stored on the first ten storage nodes. Six additional coding slices $CS_0, \ldots, CS_5$ will be calculated using a different dispersal matrix, such as the one depicted in Figure 7, which is derived from the Vandermonde matrix for systematic coding (see [18] for an explanation of why a Vandermonde matrix is inadequate for this purpose). The total storage requirement is 16*413 = 6.45 KB.

$$\begin{pmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 147 & 138 & 73 & 93 & 161 & 103 & 58 & 99 & 178 \\ 1 & 103 & 156 & 151 & 123 & 187 & 166 & 175 & 244 & 83 \\ 1 & 58 & 203 & 60 & 48 & 51 & 175 & 52 & 16 & 30 \\ 1 & 93 & 151 & 205 & 212 & 44 & 123 & 48 & 197 & 244 \\ 1 & 220 & 166 & 123 & 82 & 143 & 245 & 40 & 167 & 122 \end{pmatrix}$$

Figure 7: Dispersal matrix for the systematic $(10,16)$ Reed-Solomon code over $GF(2^8)$.

In each of the four methods, a client or attacker needs to acquire 10 of the 16 slices to read the data. Each method has different security and performance characteristics, which are included in the sections of Security and Performance below.

5 Security Evaluation

The threat model that we use is one where individual storage servers belong to different domains, both administrative and physical. Servers may be lost due to non-security-related events like power failure or water damage, or their security may be compromised; for example a rogue system administrator or outside attacker can steal data. Moreover, servers may become corrupted either maliciously or due to the natural process of time. We assume that the physical dispersal of storage servers is limiting on an attacker, and that the difficulty of breaching servers in multiple domains, along with a judicious choice of k and n, is sufficient to make the system secure.

All of these schemes provide a good level of security – if one cannot truly decode the data without acquiring all k slices, then an attacker without some *a priori* information about the data will not be able to glean anything from fewer than k slices. In the words of Rabin, "We do not see a way of fully reconstructing even small portions of D from $k-1$ pieces" [21]. [3]

However, if an attacker has some notion of what data he or she is seeking but possesses fewer than $k-1$ slices, then the schemes differ greatly. We will consider the most pathological example: An attacker possesses $m < k$ slices of the codeword C and wants to verify whether the data that it encodes matches some predetermined value. Further, if the attacker can verify that one slice of D matches, then the attacker can be assured that the rest matches. While this seems rather generous to the attacker, there are many realistic attacking scenarios that can be reduced to this one [7]. For each algorithm, we assume that the attacker knows how the slices were generated, except for the random numbers.

Shamir: Shamir's security is guaranteed. Attackers cannot get any information from fewer than k slices, regardless of their computing power. For example, with $k-1$ slices each of size w, there are 2^w potential values of d_0 that can generate those slices. Thus, every possible value of d_0 is equally likely. One needs the k-th slice to determine the actual value of d_0. This is *information theoretic* security.

Rabin: Since Rabin's IDA has no randomness, it has no security, even if the attacker owns just one slice. Since the attacker knows how the slices are generated, compromise consists solely of verifying that a slice has a predetermined value. Further, if the generator matrix is known and the data has recognizable patterns (i.e. it is not random looking) then it is possible to guess the content of missing slices. If one has $k-1$ slices, trying each of the 2^w possibilities for words of a missing slice will yield k recognizable words when the correct value is attempted.

SSMS: SSMS has *computational security* [12]. Without the key, one has to break the encryption, which can be made computationally intractible with a large enough key. Moreover, since Shamir protects the key with information theoretic security, there is no way get the key with fewer than k slices.

AONT-RS: AONT has the property that unless one has all of the encrypted data, one cannot decode any of it. This is because one needs all of the data to discover K, and one cannot decode any of the data without K. However, if an attacker owns K and one slice, then the attacker can easily verify that D has a predetermined value, just as in Rabin. Thus, we analyze the difficulty in having the attacker figure out K's value. Suppose the attacker owns the first slice, which contains the first encoded word of D, which is equal to $d_0 \oplus E(K,1)$. The encoding function guarantees that enumeration is the only way to discover K's value, which means that an attacker must test up to 2^{w_A} potential values of K to discover its real value. Like SSMS, this is *computational security*.

Thus, both AONT-RS and SSMS have computational security. If an attacker owns any data slice, then compromise can only occur by discovering K as above. If an attacker owns a coding slice, then the attacker must again enumerate potential values of K, calculate potential values of the slice and verify them. Owning $k-1$

[3] We have changed the variables in the quote to match our paper.

Algorithm	Running Time	Storage
Shamir	$\mathbf{Perf}(n,k,kb)$	nb
Rabin	$\mathbf{Perf}(n,k,b)$	$\frac{nb}{k}$
AONT-RS	$\mathbf{AONT}(b)+\mathbf{Perf}(n-k,k,b)$	$\frac{n(b+w_A)}{k}$

Table 1: Running time and storage requirements of the three dispersal algorithms.

slices adds no information – the act of verification still boils down to enumerating all potential values of K. The encryption and therefore missing words in other slices cannot be guessed in the same way they can under Rabin.

Special mention must be made of storing $K \oplus h$ as the last element of the codeword. Cryptographic hash functions are designed to have an unpredictable and uniformly distributed output. Further, they are designed to follow the strict avalanche criterion [35], meaning h is dependent on every bit of input. Therefore unless an attacker knows all code words $c_0,\ldots,c_s$, h cannot be predicted. Modeling the hash function as a random oracle, h encrypts K in the same manner as a One-Time-Pad [34] and provides information theoretic security since h is the same length as K. Therefore $K \oplus h$ yields no information about K when h is unknown.

Moreover, the avalance criterion allows the canary to be sufficient to check integrity. If any bit of the stored slices is modified, then with sufficient probability, the calculated hash h' will be different from the one used to calculate the difference. Since h' differs from h, the calculated encryption key K will be incorrect, and as a result, the value in the calculated canary will differ from its known value.

While computational security is not as strong as information theoretic security, in our view it is functionally equivalent. As long as w_A is sufficiently large, it is computationally infeasible for an attacker to even verify that slices hold given data. For example, when $w = 256$ as in Section 4, compromise requires the enumeration of 2^{256} keys. To put this in perspective, if each person on earth had access to a trillion computers that can test a trillion keys per second, it would take over 10^{35} years on average to correctly guess the key. According to some estimates of proton half-life, most matter in the universe will have decayed before the key would be found [1].

6 Theoretical Performance

Let $\mathbf{Perf}(R,C,S)$ be the CPU time that it takes to encode D, composed of S total bytes, with a $R \times C$ dispersal matrix. In terms of big-O notation, $\mathbf{Perf}(R,C,S) = O(RCS)$. A more precise evaluation of $\mathbf{Perf}(R,C,S)$ is difficult, because of the variety of ways that the encoding may be implemented. If one implements the encoding with standard finite field arithmetic, then:

$$\mathbf{Perf}(R,C,S) = \frac{S}{C}\left(\frac{(R-1)(C-1)}{\mathbf{Mult}} + \frac{R(C-1)}{\mathbf{XOR}}\right),$$

where **Mult** is the bandwidth of performing Galois Field multiplication and **XOR** is the bandwidth of performing XOR operations. This is because encoding becomes a series of dot products to create R coding slices each of whose size is $\frac{S}{C}$ bytes. The difference in the number of multiplications vs. XORs arises becuase nearly all dispersal matrices are like Figure 7 and have ones in their top rows and leftmost columns. Implementations of Reed-Solomon coding do, however, differ in their performance characteristics. Using Cauchy Reed-Solomon coding [3], for example, substitutes additional XOR operations for the multiplication and can improve performance significantly [19].

Additionally, let $\mathbf{AONT}(S)$ be the time that it takes to perform the AONT on S bytes of data. The choice of w_A, encryption and hashing technique will all affect $\mathbf{AONT}(S)$. In general, though, it is $O(S)$ and is also easy to parallelize [24].

Given the parameters k, n, b, $\mathbf{Perf}(R,C,S)$, and $\mathbf{AONT}(S)$ the performance of the three main dispersal algorithms and their storage requirements are given in Table 1. Since SSMS doesn't specify a recommended dispersal or encryption algorithm, we omit it from the remaining analyses. Roughly, its performance will be close to AONT-RS.

7 Microbenchmark Performance

To assess actual performance, we used open-source C libraries to perform the various functionalities. All tests were performed on a 4-core Intel Xeon W3530 at 2.80 GHz with 6 GB of memory at 1066 MHz running Linux kernel 2.6.32. Despite having multiple cores, all benchmarks were performed using a single thread. For Reed-Solomon coding, we used Luigi Rizzo's open source library over $GF(2^8)$ [25]. We tested a variety of k-of-n configurations, ranging from 3-of-6 to 32-of-64, measuring c_e, defined as the bandwidth of creating each coding slice, times k. For a given machine, c_e should be relatively constant, since the time to create each coding slice

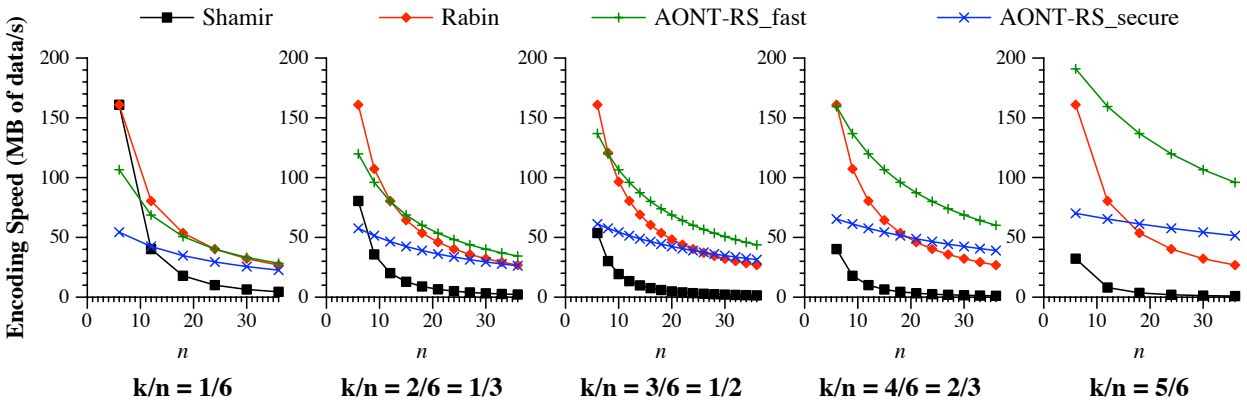


Figure 8: Performance comparison of the dispersal algorithms. Each graph affixes the k-to-n rate and plots speed of encoding with each dispersal algorithm.

should be linear in k. Despite the wide disparity in configurations, we observe that c_e is fairly consistent, with a minimum of 921.60 MB/s in the 3-of-6 configuration, to a maximum of 994.00 MB/s in 27-of-54. The average performance for the 30 configurations tested is 965.61 MB/s with a standard deviation of 11.42 MB/s. Thus, we can use c_e to approximate **Perf** as:

$$\textbf{Perf}(R,C,S) = \frac{RCS}{965.61MB/s}.$$

The encoding time for AONT is dependent on the choice of cipher and hash function. To encode S bytes using AONT, both the cipher and hash function must process S bytes. Therefore the time equals the sum of the time to encrypt S bytes plus the time to hash S bytes. We tested the performance of two pairs of cipher/hash algorithms, one tailored for high security (AES-256 and SHA-256) and the other tailored for performance (RC4-128 and MD5). For this test, we used OpenSSL 0.9.8k with a block size of 8 KB. The results are in Table 2.

	Encoding Rate (MB/s)
AES-256	143.30
RC4-128	414.17
SHA-256	160.03
MD5	559.47

Table 2: Performance of two encryption algorithms (AES-256 and RC4-128) and two hash algorithms (SHA-256 and MD5).

Thus, we come up with two functions for $\textbf{AONT}(S)$, one which we call **secure** (AES-256 and SHA-256), and one which we call **fast** (RC4-128 and MD5):

$$\textbf{AONT}_{\textbf{secure}}(S) = \frac{S}{75.60MB/s}$$

$$\textbf{AONT}_{\textbf{fast}}(S) = \frac{S}{237.99MB/s}$$

We now have the necessary information to use Table 1 to evaluate the performance of the three dispersal algorithms for any k-of-n configuration. We do so in Figure 8. Each graph affixes a k-of-n ratio called a *rate* and then plots the speed of encoding in MB of data per second. The rates increase by $\frac{1}{6}$ for each successive graph, starting with a very low rate of $\frac{1}{6}$ and proceeding to a very high rate of $\frac{5}{6}$.

The trade-offs of the various formulas are apparent from the graph. There is a dispersal cost for all three algorithms and an AONT cost for the AONT-RS algorithms. The AONT cost is constant, since it depends solely on the size of the data. Thus, when dispersal is very fast, as in the 1-of-6 and 2-of-6 cases, Rabin outperforms AONT-RS$_{\textbf{fast}}$ and Shamir outperforms AONT-RS$_{\textbf{secure}}$. As k and n grow, however, the dispersal costs increase. This increase is most pronounced with Shamir, then with Rabin and finally with AONT-RS. For each rate except the very low $\frac{1}{6}$, there is a point where the performance of AONT-RS$_{\textbf{fast}}$ becomes the best, and a point where AONT-RS$_{\textbf{secure}}$'s performance surpasses both Shamir and Rabin. These points come at lower values of n for higher k-of-n rates.

A schematic of Cleversafe's storage architecture is depicted in Figure 9. Although not plotted above, of special interest is the 3-of-5 data point, since this is the k-of-n configuration measured by POTSHARDS [30], an archival storage system that uses Shamir for both fault-tolerance and security. For this configuration, the perfor-

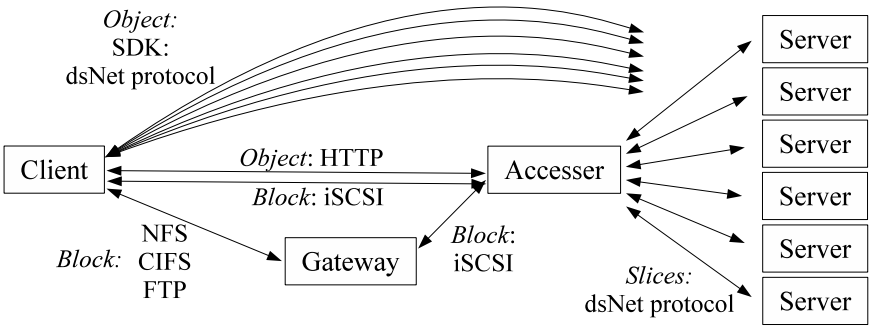


Figure 9: A high-level picture of Cleversafe's storage architecture.

mance of AONT-RS$_{\textbf{secure}}$ (65.4 MB/s) is nearly identical to Shamir (64.4 MB/s), which means that a system like POTSHARDS can achieve computational security rather than information theoretic security for the same performance, but with a factor of three less storage.

8 Commercial Dispersed Storage System

AONT-RS is a feature in the storage software and appliances sold by Cleversafe, which developed the technique to address the threat model of compromise, theft or loss of disks and devices. By appropriately tuning the dispersal configuration, all disks or devices at an entire site can be stolen and the data will remain confidential. Similarly, as long as a minimum threshold of servers are available, subsets of servers may be brought offline temporarily for maintenance, or permanently for replacement. Since the servers are protected by AONT-RS, storage owners may dispose of servers without having to "wipe" the drives clean, since the information on the servers is impossible to obtain without gaining access to some subset of the remaining servers.

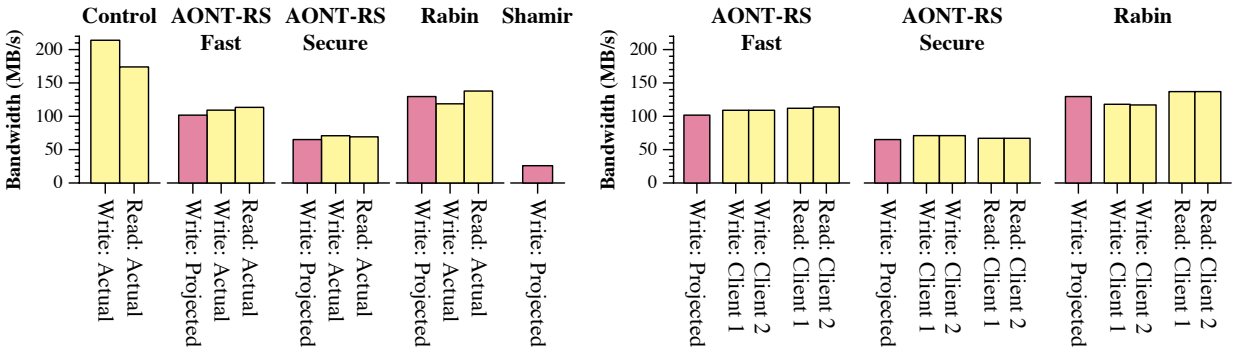
Two paradigms are exposed to clients — a block paradigm that supports standard protocols like NFS, CIFS, FTP and iSCSI, and an object paradigm that supports larger storage units for better performance. An *Accesser* calculates mappings that associate blocks or objects to slices on dispersed storage servers (termed "Slicestores" in Cleversafe's product). A common configuration is to encode each block or object into 16 slices using a $(10,16)$-threshold AONT-RS scheme.

Block reads and writes that use iSCSI go through the Accesser. The Accesser performs the block-to-slice encoding and decoding, and also manages the traffic to and from the servers. The other protocols require a *Gateway*, typically co-located with the Accesser, that translates between the various file protocols and iSCSI. Since this path has two hops and interacts with the servers with small messages, the performance of the block protocols is limited by the networking hardware and not the AONT-RS protocol. Storage servers do support multiple Accessers, which relieves one bottleneck of the block-based system.

To achieve better performance, Cleversafe also exports a protocol for large objects. Objects are partitioned into Megabyte-sized chunks, which are then encoded into slices for dispersal. Clients may either read and write objects through the Accesser using HTTP, or they may use a SDK to perform their own AONT-RS encoding/decoding so that they may interact directly with the servers. In both cases, the client manages the context of the object name. A common software architecture is that clients use a database to maintain the the meaning and relationships of the content, and they store the object names in a column of the database.

Slice pointers are 48 bytes in length and are composed of three parts: *routing information* that enables slices to be routed to and from the correct servers, the *source name* which identifies the slice, and *vault information* which enables access control. The source name is opaque – its interpretation is dependent on the specific client and server. Vaults are logical containers of storage. Each vault has its own quotas, data coding parameters and access controls. Access controls are identity-based; each vault may have an arbitrary number of accounts granted read or write permissions to it.

Each slice is stored with metadata that identifies the slice's coding parameters and a version number. The version number is increased for each distinct write of the block or object, and concurrency control is maintained via the SDK with transactions and a three-phase commit. An additional parameter of each system is the *write threshold*, z, where $k \leq z \leq n$. This specifies how many slices must be written before a write can be committed. Setting z closer to k improves latency at the expense of reliability for a window of time. The remaining $(n-z)$ writes are processed in the background, which reduces this window of exposure.



(a) One client

(b) Two clients

Figure 10: Actual and projected performance of dispersed storage of 10 MB objects on a $(5,8)$ test configuration.

Authentication in the system is two-way: servers authenticate themselves to clients by means of a digital certificate, which identifies it within the dispersed storage system and allows TLS sessions to be created. The method of authentication of the client to the server is flexible — both password and certificate-based authentication are supported. Despite use of AONT-RS, secure network communication is still required for security since a threshold number of slices travel together over the 'last mile' of the client's connection.

All components are written in Java. Reed-Solomon erasure coding is performed using Java's FEC library [16], and encryption using SunJCE.

9 Measured Performance

To measure performance, we use a commercial configuration with one or two clients and eight servers. The client and Accesser machines each have a 4-core Intel(R) Xeon(R) X3430 processor running at 2.40 GHz with 8 MB cache and 16 GB of ECC RAM. Four GB of memory is allocated to the JVM when executing the software. We use the Java HotSpot(TM) 64-Bit Server VM (build 17.0-b16, mixed mode) running Java 1.6.0_21. The storage servers each have a 4-core Intel(R) Xeon(R) X3460 processor at 2.80 GHz with 8 MB cache and 16 GB of ECC RAM. For storage, each server has twelve 2 TB Seagate SATA drives. The networking between components consists of a 10 Gb Ethernet switch. To handle simultaneous connections to multiple servers, the Accessers have 10 Gb network interface cards. The servers' cards are 1 Gb.

Our main test has the client spend 10 minutes reading and writing 10 MB objects, held in main memory, to the eight-server storage network, using the SDK and object interface. The coding parameters are $k = 5$ and $n = 8$, and five threads are employed by the client to leverage all of its cores. As in section 7, we recorded microbenchmarks of the various components of dispersal:

$$\begin{aligned} \mathbf{AONT_{secure}}(S) &= \frac{S}{104.77MB/s} \\ \mathbf{AONT_{fast}}(S) &= \frac{S}{249.03MB/s} \\ c_e &= 2628MB/s \end{aligned}$$

The performance of a control and the dispersal algorithms is shown in Figure 10(a). The control has the client perform no encoding, but still sends 8 slices to the servers. While the Cleversafe implementation is flexible, allowing us to embed Rabin and both AONT-RS dispersal algorithms, we did not implement Shamir within the framework. This is because the blowup of storage requirements by a factor of five would be unreasonable.

We show the actual performance of writes and reads for the control, the two AONT-RS implementations and Rabin. We also include the projected write performance of the dispersal algorithms, including Shamir, using the performance equations from section 7, the microbenchmarks, plus the performance of the control as the actual dispersal bandwidth (214 MB/s).

For the three dispersal algorithms that we tested, the projected performance was within ten percent of the actual performance. We find this result compelling because the system on which the tests were performed was a production-level system, implementing the full functionality of Cleversafe's commercial storage system, including access control and metadata management.

In the tests with coding, the CPU utilization of the client is measured to be 90%. Since the closest I/O bottlenecks are the eight 1-Gbps links to the storage servers, it is clear that the limiting factor in these tests is the ability of the client computer to process data. To further affirm the client as bottleneck, we ran two clients

simultaneously and present their performance in Figure 10(b). The clients' performance is nearly identical to Figure 10(a).

It is worth noting that AONT-RS$_{\textbf{secure}}$ exhibits worse performance when reading than when writing; we expected that during reads, less CPU resources would be required, since some slices do not need to be processed by the IDA. The worse performance is due to the SunJCE implementation of AES, which is significantly slower when decrypting than when encrypting. In a stand-alone benchmark we observed 31.51 MB/s vs. 44.77 MB/s when encrypting.

10 Tales of Deployment

Today, there are over 20 Cleversafe dispersed storage installations in pilot and production around the world, with customers drawing from a diverse set of industries including financial, health care, entertainment, and defense. Several customers (who have asked to remain anonymous) have cited one important factor in their purchasing decision: that the contents of small sets of servers are meaningless in isolation. Thus, one can decommission disk drives or potentially even server sites without having to "wipe" them, which can be expensive [4]. Since nearly all U.S. states have "data breach laws," that require companies to proactively disclose the loss of storage that is not encrypted [33], using AONT-RS can save companies time, attorney fees and bad publicity that results from having to alert consumers to a data breach.

One of Cleversafe's deployments is for The Museum of Broadcast Communications that serves its video collections on the Internet. In particular, over 8,500 hours of historical audio and video content have been digitized and stored on tens of terabytes in one of Cleversafe's dispersed storage systems. Roughly 200,000 monthly visitors access the archives over the web.

The Museum deployment is composed of 16 storage servers, each having 4 TB of raw capacity and spread across 8 sites: Chicago, Dallas (two locations) Denver, New Jersey, San Francisco, Seattle and Tampa. The sites are situated across three power grids in the continental United States, and the data is dispersed in a 10-of-16 configuration. In this way, even if one entire power grid shuts down, enough servers will remain accessible to retrieve all the data. The Museum uses the object store interface inside its internal database, so that users employ the database to search a rich set of metadata about the movies, which can then be retrieved using the object handle.

[4]For example, see `http://www.east-tec.com/enterprise/disposesecureent/`.

Internally, Cleversafe maintains dispersed storage systems having over 1 PB of capacity. These are used internally for development, testing and storing production data. Employees have their own personal vaults with access to a 30 TB pool of dispersed storage, which is implemented over 8 geographically separated storage servers across the United States.

In one case, Cleversafe initially deployed a system across four sites, but at a later time decided that it should be migrated to 8 sites to provide better tolerance to site and power grid outages. To accomplish this without bringing the system down, machines were incrementally boxed up and shipped across the country, such that at all times a threshold number remained online. Therefore the system remained accessible for reads and writes throughout the process. The same essential technique is now used to apply software updates. Nodes are upgraded individually allowing the system to maintain availability throughout the upgrade process.

11 Conclusion

Dispersed storage systems enable availability, scalability, and performance based on physical proximity. They also enable security via (k,n) threshold schemes that require attackers to authenticate themselves to k of n storage nodes in order to read data. The threshold schemes provide this security without relying on the secure storage of encryption keys, which is a notoriously difficult problem.

We have described a new dispersal algorithm called AONT-RS, which combines the All-Or-Nothing Transform with systematic Reed-Solomon codes to achieve computational security. Compared to traditional approaches to dispersal, AONT-RS has a very attractive blend of properties. Its storage and computational footprint is much less than Shamir secret sharing. While Shamir achieves information theoretic security AONT-RS's security can be tuned so that compromise is computationally infeasible. Compared to Rabin's classic dispersal algorithm, AONT-RS achieves a far greater degree of security, and also better performance for larger installations. This is because AONT-RS is based on a systematic Reed-Solomon erasure code rather than the non-systematic code employed by Rabin. We have detailed the theoretical and applied performance of the dispersal algorithms, and described a commercial dispersed storage product that is based upon the dispersal algorithm.

AONT-RS is not specific to our dispersal solution. For example, the POTSHARDS archival storage system [30] could use AONT-RS to implement computational rather than information theoretic security and reduce their storage requirements by a factor of three. Other solutions such as Gridsharing [31] can improve their security by

employing AONT-RS rather than a standard systematic Reed-Solomon code.

In future work, we would like to collect data from our private and commercial deployments concerning failures, node availability, compromise and attack. Such data will enable us to make better policy decisions concerning configurations of dispersed storage. These decisions will allow us to tune the AONT and erasure code configuration used, and will also allow us to make the most efficient use of our storage.

12 Acknowledgements

This material is based upon work supported by the National Science Foundation under grants CNS-0615221 and CSR-1016636. The authors gratefully acknowledge Ilya Volvovski's contribution to the work, plus Steve Hand for shepherding the paper through the final reviewing process.

References

[1] AMSLER *et al*, C. Review of particle physics. *Physics Letters B 667*, 1 (2008).

[2] AYCOCK, J. *Computer Viruses and Malware (Advances in Information Security)*. Springer-Verlag, New York, 2006.

[3] BLOMER, J., KALFANE, M., KARPINSKI, M., KARP, R., LUBY, M., AND ZUCKERMAN, D. An XOR-based erasure-resilient coding scheme. Tech. Rep. TR-95-048, International Computer Science Institute, August 1995.

[4] CLEVERSAFE, INC. Cleversafe dispersed storage. Community portal:`www.cleversafe.org`, 2010.

[5] COHEN, B. Incentives build robustness in BitTorrent. In *Workshop on Economics of Peer-to-Peer Systems* (Berkely, CA, June 2003).

[6] DAEMEN, J., AND RIJMEN, V. *The Design of Rijndael, AES — The Advanced Encryption Standard*. Springer-Verlag, New York, 2002.

[7] FERGUSON, N., SCHNEIER, B., AND KOHNO, T. *Cryptography Engineering*. John Wiley & Sons Ltd, Chichester, 2010.

[8] GHEMAWAT, S., GOBIOFF, H., AND LEUNG, S. T. The Google file system. In *19th ACM Symposium on Operating Systems Principles (SOSP '03)* (2003).

[9] GOODSON, G. R., WYLIE, J. J., GANGER, G. R., AND REITER, M. K. Efficient byzantine-tolerant erasure-coded storage. In *DSN-04: International Conference on Dependable Systems and Networks* (Florence, Italy, 2004), IEEE.

[10] GREENAN, K., MILLER, E., AND SCHWARTZ, T. J. Optimizing Galois Field arithmetic for diverse processor architectures and applications. In *MASCOTS 2008: 16th IEEE Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems* (Baltimore, MD, September 2008).

[11] HAEBERLEN, A., MISLOVE, A., AND DRUSCHEL, P. Glacier: Highly durable decentralized storage despite massive corrolated failures. In *2nd Symposium on Networked Systems Design and Implementation (NSDI)* (2005).

[12] KRAWCZYK, H. Secret sharing made short. In *13th Annual International Conference on Advances in Cryptology* (1993).

[13] MACWILLIAMS, F. J., AND SLOANE, N. J. A. *The Theory of Error-Correcting Codes, Part I*. North-Holland Publishing Company, Amsterdam, New York, Oxford, 1977.

[14] MANIATIS, P., ROSENTHAL, D. S. H., ROUSSOPOULOS, M., AND BAKER, M. LOCKSS: A peer-to-peer digital preservation system. *ACM Transactions on Computer Systems 23* (2003).

[15] NATIONAL INSTITUTE OF STANDARDS AND TECHNOLOGY. Secure hash standard (shs). FIPS PUB 180-3, `http://csrc.nist.gov/publications/fips/fips180-3/fips180-3_final.pdf`, October 2008.

[16] ONION NETWORKS. Java FEC Library v1.0.3. Open source code distribution: `http://onionnetworks.com/fec/javadoc/`, 2001.

[17] PLANK, J. S. A tutorial on Reed-Solomon coding for fault-tolerance in RAID-like systems. *Software – Practice & Experience 27*, 9 (September 1997), 995–1012.

[18] PLANK, J. S., AND DING, Y. Note: Correction to the 1997 tutorial on Reed-Solomon coding. *Software – Practice & Experience 35*, 2 (February 2005), 189–194.

[19] PLANK, J. S., LUO, J., SCHUMAN, C. D., XU, L., AND WILCOX-O'HEARN, Z. A performance evaluation and examination of open-source erasure coding libraries for storage. In *FAST-2009: 7th Usenix Conference on File and Storage Technologies* (February 2009), pp. 253–265.

[20] PLANK, J. S., SIMMERMAN, S., AND SCHUMAN, C. D. Jerasure: A library in C/C++ facilitating erasure coding for storage applications - Version 1.2. Tech. Rep. CS-08-627, University of Tennessee, August 2008.

[21] RABIN, M. O. Efficient dispersal of information for security, load balancing, and fault tolerance. *Journal of the Association for Computing Machinery 36*, 2 (April 1989), 335–348.

[22] REED, I. S., AND SOLOMON, G. Polynomial codes over certain finite fields. *Journal of the Society for Industrial and Applied Mathematics 8* (1960), 300–304.

[23] RHEA, S., WELLS, C., EATON, P., GEELS, D., ZHAO, B., WEATHERSPOON, H., AND KUBIATOWICZ, J. Maintenance-free global data storage. *IEEE Internet Computing 5*, 5 (2001), 40–49.

[24] RIVEST, R. All-or-nothing encryption and the package transform. In *4th International Workshop on Fast Software Encryption* (1997), pp. 210–218.

[25] RIZZO, L. Erasure codes based on Vandermonde matrices. Gzipped **tar** file posted at `http://planete-bcast.inrialpes.fr/rubrique.php3?id_rubrique=10`, 1998.

[26] ROWSTRON, A., AND DRUSCHEL, P. Storage management and caching in PAST, a large-scale, persistent peer-to-peer storage utility. *ACM SIGOPS Operating Systems Review 35*, 5 (2001), 188–201.

[27] SANTRY, D. S., FEELEY, M. J., HUTCHINSON, N. C., VEITCH, A. C., CARTON, W., AND OFIR, J. The Google file system. In *17th ACM Symposium on Operating Systems Principles (SOSP '99)* (1999).

[28] SHAMIR, A. How to share a secret. *Communications of the ACM 22*, 11 (November 1979), 612–613.

[29] STORER, M. W., GREENAN, K. M., MILLER, E. L., AND VORUGANTI, K. Pergamum: Replacing tape with energy efficient, reliable, disk-based archival storage. In *FAST-2008: 6th Usenix Conference on File and Storage Technologies* (San Jose, February 2008), pp. 1–16.

[30] STORER, M. W., GREENAN, K. M., MILLER, E. L., AND VORUGANTI, K. POTSHARDS – a secure, long-term storage system. *ACM Transactions on Storage 5*, 2 (June 2009).

[31] SUBBIAH, A., AND BLOUGH, D. M. An approach for fault tolerant and secure data storage in collaborative work environments. In *ACM Workshop on Storage Security and Survivability* (2005).

[32] TAHO-LAFS. Tahoe least authority file system. Open source code distribution: `http://tahoe-lafs.org/trac/tahoe-lafs`, 2010.

[33] VANCE, K. Keeping pace with data encryption laws. `www.esecurityplanet.com/trends/article.php/3887111`, June 2010.

[34] VERNAM, G. S. Cipher printing telegraph systems for secret wire and radio telegraphic communications. *Journal of the IEEE 55* (1926), 109–115.

[35] WEBSTER, A. F., AND TAVARES, S. E. On the design of S-boxes. In *Advances in Cryptology - Crypto '85* (1985), Springer-Verlag, pp. 523–534.

[36] WILCOX-O'HEARN, Z. Zfec 1.4.0. Open source code distribution: `http://pypi.python.org/pypi/zfec`, 2008.