Store, Forget, and Check: Using Algebraic Signatures to Check Remotely Administered Storage

Thomas Schwarz, S.J.
Department of Computer Engineering
Santa Clara University
tjschwarz@scu.edu

Ethan L. Miller
Storage Systems Research Center
University of California at Santa Cruz
elm@cs.ucsc.edu

Abstract

The emerging use of the Internet for remote storage and backup has led to the problem of verifying that storage sites in a distributed system indeed store the data; this must often be done in the absence of knowledge of what the data should be. We use m/n erasure-correcting coding to safeguard the stored data and use algebraic signatures—hash functions with algebraic properties—for verification. Our scheme primarily utilizes one such algebraic property: taking a signature of parity gives the same result as taking the parity of the signatures. To make our scheme collusion-resistant, we blind data and parity by XORing them with a pseudo-random stream. Our scheme has three advantages over existing techniques. First, it uses only small messages for verification, an attractive property in a P2P setting where the storing peers often only have a small upstream pipe. Second, it allows verification of challenges across random data without the need for the challenger to compare against the original data. Third, it is highly resistant to coordinated attempts to undetectably modify data. These signature techniques are very fast, running at tens to hundreds of megabytes per second. Because of these properties, the use of algebraic signatures will permit the construction of large-scale distributed storage systems in which large amounts of storage can be verified with minimal network bandwidth.

1. Introduction

As the Internet has increased in speed and bandwidth, remote storage of data over the network has become feasible. Peer-to-peer (P2P) storage systems, especially those based on the so-called Distributed Object Location and Retrieval (DOLR) systems [11] such as Oceanstore [18] are an important class of such systems. Systems like these face a number of challenges such as data privacy, protection of the data against alteration, data loss due to node unavailability and the *free rider* problem. In this paper, we introduce new techniques based on algebraic signatures that allow a "data origination site" to verify that a a remote site is storing data correctly, or whether a number of sites that collectively store a collection of objects is doing so correctly. Our scheme does not need the original data for its check, and only two small messages need be exchanged for each check. Both of these properties should be attractive to designers of remote storage schemes.

As peer-to-peer technology has matured, a number of systems such as Oceanstore [18], Intermemory [13], Ivy [27], PAST [32], Starfish [12], FarSite [1] have been built to utilize remote data storage. To protect against failure, this data is stored redundantly using either pure replication or m/n erasure coding. Similarly, Lillibridge, *et al.* [19] propose a scheme where participants mutually store each other's backup data. All these schemes store data on sites that cannot be trusted. In addition to peer unavailability, they must face the problem of *free riders*. Free riders only pretend to store others' data and thus enjoy the benefits of remote storage of their data without incurring any costs of their own. Our approach can be used to address the free rider problem as well as the more general problem of involuntary data loss or generic data corruption by using a system of challenges and responses. The naïve algorithm requests random blocks of data from the storage site, verifying them against the locally-stored data. This is particularly easy in a remote back-up scheme, since the original of the data is still available, but becomes quite difficult in remote storage systems where the original is not retained. DOLR and other P2P storage systems that use redundancy in storage face an additional problem of assuring that all data reflects the same state. While stale data in a replicated system might still be useful, stale parity data in a scheme based on erasure coding usually prevents the reconstruction of the application data in case of need.

In a peer-to-peer system, nodes might not have a high bandwidth connection; furthermore, many such nodes, including those in homes, have an asymmetric connection that limits uploads but has ample download capacity. A peer with such a connection can still store archival data that has little likelihood to be read, albeit storing such data slowly. A naïve challenge scheme, however, will regularly use the narrow upstream path and thus lead to noticeable performance losses.

An "ideal" versatile challenge-response scheme should use only small challenges and responses. It should allow the challenger to test unpredictably so that the responder cannot just simply precompute and store all challenges instead of the data. It should be able to detect changes in the stored data and in particular should discover the most frequent data corruptions, such as very minute changes or permutation of blocks. Finally, it should be able to check whether parity in a storage scheme using erasure correcting codes is coherent with the data without regenerating the parity data from the application data.

Our scheme fulfills all these criteria as long as the erasure correcting code is calculated using only XOR operations; such codes include X-codes [35], EvenOdd [2, 3], row-diagonal parity [9], and linear codes over a Galois field such as the popular Reed-Solomon codes [24]. Our signatures are *algebraic signatures* [22, 33]—hashes or checksums with algebraic properties. These are sometimes called "Rabinesque," after Michael Rabin who used a similar formula in the Karp-Rabin pattern matching algorithm and another similar formula to identify similarities between documents.

Our techniques work for systems that store data redundantly at a remote site by adding some parity data, generated by a linear erasure-correcting m/n code. For added security, the data or parity can be "blinded" (encrypted) by XORing it with with a pseudo-random stream. Both the erasure-correcting code and the pseudo-random stream can be easily regenerated by anyone knowing a secret. When a system wants to verify the data, it asks the storing sites to each return a signature calculated over a specified part of the data. The requesting system can then determine, solely based on the secret and the returned signatures, whether the returned values can reflect valid data. A random chosen answer has a probability of 2^{-l} of being correct, where l is the length of the answer in bits. Thus, with just a few random queries, our techniques can verify that a remote site or set of sites is indeed likely to be storing data correctly.

In this paper, we first discuss previous work in DOLR systems, focusing on techniques used to verify that remote data is correctly stored. We then describe the algebraic signatures that our approach uses. Next, we describe the protocols that we use to request and verify signatures, followed by implementation and performance results. We conclude with future directions for our work.

2. Related Work

Using hashes or signatures to condense the contents of stored data blocks into a few bytes for comparison purposes is a generic technique that has been used for a long time. The particular formula for the single symbol signature is similar to the one that was used by Harrison [15] for pattern searches. Rabin used the same type of formula for fingerprinting [28] and later analyzed it with Karp for remote pattern matching [17]. In another context, the calculation of a supersignature from page signatures used for file comparison was used by Schwarz, Bowdidge, and Burkhard [34] and further expended and analyzed by Litwin and Schwarz [22] for use in Scalable Distributed Data Structures. Broder, *et al.* use fingerprinting and a technique called shingling to measure the difference of documents [5, 6].

There are many schemes that make extensive use of the Internet for wide-area storage. For example, Lillibridge, *et al.* propose a scheme to back up user data over the Internet [19]. They verify the honesty of the storage providers by a simple challenge-response scheme, where the data owner asks the storer to retrieve a randomly chosen block to the owner. Other peer-to-peer storage systems, especially those based on Distributed Object Location and Retrieval (DOLR) [11] can similarly benefit from the algorithms we provide to check remotely stored data using techniques such as those proposed by Caronni and Waldvogel [7].

OceanStore [18] uses anonymity of the data owner to protect the data as well as periodic checks by Oceanstore itself that all data is still available. It uses erasure correcting codes (m/n codes with relatively high values for m, n, and $r = n/m$) to protect the data against site unavailability and failure, and performs checks to ensure that all data is accessible. It protects against malicious deletion by using *global sweeps* in which the data owners sweep through data under their control. The correctness of the data is verified by checking every byte of the data; this requires that OceanStore either trust the remote nodes to correctly verify their information or that the remote nodes return the data and checksums to a trusted node for verification. Intermemory [13] is another large-scale, distributed, fault-tolerant archival system that encrypts and erasure-encodes data and stores them on untrusted sites. PAST [32] and CFS [10] are also global-scale storage systems, but of read-only data protected by replication. Ivy [27] is a read-write P2P file system in which malicious failures are discovered "after the fact;" the use of our techniques might enable such a system to proactively discover failures. Starfish [12] replicates data items three-fold and uses a write quorum of two of the three sites to update them, though the parameters can of course be generalized. Like OceanStore, PASIS [14] uses m/n codes to store data, but it also uses m/n codes to provide some level of security. Unlike more recent systems, first generation file sharing systems such as Napster, Gnutella [31], and Freenet [8] also store data remotely, but tend to not pay attention to the validation and availability of the data. LOCKSS [25] replicates data among multiple library sites and uses voting techniques based on the exchange of cryptographically strong signatures of each library's copy of a particular file. This technique works in LOCKSS because each library must keep a copy of the file, allowing it to verify the cryptographic hash against its own copy. Because these P2P systems store data on remote untrusted nodes, they could all benefit from the use of algebraic signatures for remote storage verification.

3. Random Parity and Signatures

We propose to use a conjunction of algebraic signatures—small strings calculated from substrings of stored data in a way that has exploitable algebraic properties—and redundant storage generated with the help of linear, maximum distance separable error control codes. We use the same mathematical structure, a Galois field $\mathscr{GF}(2^f)$ to define both.

In order to operate on the data, we treat it as a stream of symbols that are bit strings of length f, with typical values $f = 8$ or $f = 16$. The 2^f different symbols form the elements of a Galois field $\mathscr{GF}(2^f)$ in which we can add, multiply, divide by, and subtract with the same rules as are valid for these operations in the better known field of real numbers. In $\mathscr{GF}(2^f)$, addition is the same as subtraction and both are the same as the XOR operation. The zero in this field is the string with only zeroes (*e. g.*, 0000 0000 for $f = 8$). The definition and implementation of multiplication is somewhat more involved, but there are standard implementation techniques [24] for it.

The most important property of algebraic signatures for our purposes is that calculating parity and taking a signature commute. In other words, the algebraic signature of a parity container can be calculated solely using the signatures of the data containers. This is true as long as we use the same field in calculating the signature and the parity, and the erasure correcting code is a linear code. Such codes include the simple parity code that calculates parity as the XOR of data. The remainder of this section describes algebraic signatures in detail and how the parity codes are generated.

3.1. Algebraic Signature Definition

An algebraic signature of a string (of symbols) $x_0, x_1, \ldots x_{N-1}$ is simply defined by

$$\mathrm{sig}_\alpha(x_0, x_1, \ldots, x_{N-1}) = \sum_{\nu=0}^{N-1} x_\nu \cdot \alpha^\nu$$

, and is itself a single symbol. Sometimes, it is useful to have slightly larger signatures, as can be obtained by concatenating several signatures:

$$\mathrm{sig}_{(n,\alpha)} = (\mathrm{sig}_{\alpha^0}, \mathrm{sig}_{\alpha^1}, \mathrm{sig}_{\alpha^2}, \ldots \mathrm{sig}_{\alpha^{n-1}})$$

$\mathrm{sig}_{n,\alpha}$ for certain α can detect any changes of up to n symbols [22]. The simple as well as the concatenated signature are linear in the string over which they are calculated. An interesting consequence of linearity is that we can combine signature calculation with blinding the data by XORing the data with a pseudo-random string. Basically, if X is the plaintext and Y the pseudo-random string, then $\mathrm{sig}_\alpha(X \oplus Y) = \mathrm{sig}_\alpha(X) \oplus \mathrm{sig}_\alpha(Y)$. Signatures interact in a similar way with various erasure and error correcting codes that use only the XOR operation, as do Hellerstein's proposal [16], EvenOdd[2], row-diagonal parity [9], and convolutional array codes [3]. More importantly, linear m out of n codes also have this property. More precisely, assume that we have an erasure correcting code that calculates k parity containers $\mathbf{P}_1, \ldots \mathbf{P}_k$ from the m data buckets $\mathbf{D}_1, \mathbf{D}_2, \ldots \mathbf{D}_m$ as $\mathbf{P}_i = \wp_i(\mathbf{D}_1, \mathbf{D}_2 \ldots \mathbf{D}_m)$. Then $\mathrm{sig}_\alpha(\wp_i(\mathbf{D}_1, \ldots, \mathbf{D}_m)) = \wp_i(\mathrm{sig}_\alpha(\mathbf{D}_1), \ldots, \mathrm{sig}_\alpha(\mathbf{D}_m))$. Another way of putting this property is that $(\mathrm{sig}_\alpha(\mathbf{D}_1), \ldots \mathrm{sig}_\alpha(\mathbf{D}_m), \mathrm{sig}_\alpha(\mathbf{P}_1) \ldots \mathrm{sig}_\alpha(\mathbf{P}_k))$ is a code word in the code. The proof of this and other properties can be found elsewhere [22, 33].

These algebraic signatures are called often Rabinesque after Michael Rabin, who used a formula of similar type to define fingerprints [17, 28] in a similar setting and make use of their algebraic properties for what is now known as the Karp-Rabin pattern matching algorithm and for fingerprinting. The same formula was also used for fast file comparison [34]. Fingerprinting and a technique called shingling can also be used to measure the difference of documents [5, 6]. Our use of algebraic signatures essentially compresses the contents of a large portion of data into a very small entity that changes if the data is changed a little bit. In this way, it is similar to "cryptographically secure" hash functions such as MD5, SHA-1, and SHA-256, though algebraic signatures are not cryptographically secure because it is easy to deliberately construct two strings that have the same algebraic signature. Using a SHA-type hash function allows for the comparison of files with very small messages, but only if the local system maintains a copy of the objects whose remote storage is to be verified.

3.2. Generating Random Linear Codes

Our proposal includes security in its more ambitious variant, described in Sections 4.2 and 4.3, from custom-tailoring the parity generating code to the stored object. We use a systematic, linear erasure correcting code [24] over a

Galois field such as $\mathscr{GF}(2^8)$ or $\mathscr{GF}(2^{16})$ in a fairly standard way. We create m data containers of approximately equal size; the m containers can be filled with parts of the same object or unrelated objects. To this, we add k parity containers of the same size as the data containers using the algorithm described in this section. We refer to the collection of $n = m + k$ data and parity containers as a *reliability group*. If a single object is distributed among the containers, this is similar to a variant of Rabin's Information Dispersal Algorithm [29].

Our codes are defined by a $m \times n$ *generator matrix* $\mathbf{G}$ that has the following two properties: (1) Every $m \times m$ submatrix of $\mathbf{G}$ (formed by selecting m columns) of $\mathbf{G}$ is invertible. (2) The first m columns of $\mathbf{G}$ is the identity matrix. Let $\mathbf{D}_1$, $\mathbf{D}_2, \ldots \mathbf{D}_m$ be the contents of the data containers written as a column vector with symbols as the coefficients. Then

$$(\mathbf{D}_1, \mathbf{D}_2, \ldots \mathbf{D}_m) \cdot \mathbf{G} = (\mathbf{D}_1, \mathbf{D}_2, \ldots \mathbf{D}_m, \mathbf{P}_1, \mathbf{P}_2 \ldots \mathbf{P}_k)$$

defines the contents $\mathbf{P_1}, \mathbf{P_2}, \ldots \mathbf{P}_k$ of the k parity containers, again as a column vector of symbols. Property 1 insures that the multiplication reproduces the data container columns. Property 2 means that given any m of the n containers in the reliability group, we can solve a linear equation to recalculate the remaining k containers, *i. e.*, our code is an m/n erasure correcting code. As such, it also has error correcting capabilities: given all n containers, we can identify one or a few altered containers and repair them.

As mentioned earlier, we must be able to generate different codes. To do this, we first define a family of $m \times n$ matrices that has Property 1. Next, we observe that elementary row transformations (adding a multiple of one row to another row, exchanging two rows, and multiplying a row with a non-zero Galois field element) leave Property 1 intact. Using these transformations, we can use the Gaussian elimination algorithm (for linear equations) in order to transform the matrix into a generator matrix in the right form

$$\mathbf{G}_{a_1,\ldots a_n} = \begin{pmatrix} 1 & 0 & \cdots & 0 & \hat{P}_{1,1} & \cdots & \hat{P}_{1,k} \\ 0 & 1 & \cdots & 0 & \hat{P}_{2,1} & \cdots & \hat{P}_{2,k} \\ \vdots & \vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & 1 & \hat{P}_{m,1} & \cdots & \hat{P}_{m,k} \end{pmatrix}$$

In addition to elementary row transformations, multiplying any column of a matrix with a non-zero Galois field element also retains Property 1. Thus, if required, we can additionally multiply the last k columns by a non-zero Galois field element in order to create even more generator matrices.

For our starting point, we use Vandermonde matrices

$$\mathbf{V}_{a_1,\ldots,a_n} = \begin{pmatrix} 1 & 1 & \cdots & 1 \\ a_1 & a_2 & \cdots & a_n \\ {a_1}^2 & {a_2}^2 & \cdots & {a_n}^2 \\ \vdots & \vdots & \ddots & \vdots \\ {a_1}^{m-1} & {a_2}^{m-1} & \cdots & {a_n}^{m-1} \end{pmatrix}$$

These have Property 1 if the a_1, a_2, $\ldots a_n$ are all different. Thus, there are $(2^f)(2^f - 1)\ldots(2^f - n + 1)$ ways to generate such a Vandermonde matrix over $\mathscr{GF}(2^f)$. In lieu of a Vandermonde matrix, we can use an $m \times n$ Cauchy matrix $\mathbf{C} = (c_{i,j})$ in which $c_{i,j} = \frac{1}{a_i + b_j}$. This matrix has Property 1 if the $m + n$ parameters a_1, a_2, $\ldots, a_m$, b_1, b_2, $\ldots$, b_m are all pairwise different. This condition imposes a limit on the number of parameters of the m/n code, namely $m + n \leq 2^f$ for Cauchy matrices with coefficients in $\mathscr{GF}(2^f)$ and $n \leq 2^f$ for Vandermonde matrices. In both cases, the contents of the *parity matrix* $(\hat{P}_{i,j})$—the right half of $\mathbf{G}_{a_1,a_2,\ldots a_n}$—can be given as a complex function of the parameters, but in practice, calculating the parity matrix with the Gaussian elimination algorithm is simpler and faster. The algorithm starts by selecting the coefficient in the first row and column as a pivot. It multiplies the first row in order to turn the pivot into 1. It then adds a multiple of the first row to generate zeroes elsewhere in the first column. It then proceeds row by row, selecting the coefficients in the main diagonal as pivots. It turns out that because we start with a matrix with Property 1, none of the pivots will ever be zero. This leads to a slightly streamlined generator matrix generating algorithm. In consequence, the total number of arithmetic operations is always less than or equal to $2m^2n$.

The generation of parity data themselves is also reasonably efficient. As with all linear codes over $\mathscr{GF}(2^f)$, parity calculation involves multiplying data symbols by the matrix coefficients p_ij and XOR operations. There are erasure correcting codes that only use XOR operations [23], but as it turns out, extensive use of look-up tables for matrix multiplication gives competitive parity generation performance. This type of code has been used for LH*RS [20, 21] and gave reasonable performance times.

The number of possible parity matrices is quite large. There are $2^f!/(2^f - n)!$ possible ways to generate a sequence of n different values among the 2^f symbols in the Galois field. Even if we use the tiny $\mathscr{GF}(2^8)$ and generate only one parity chunk, then there are $2^8 - 1$ possible parity encodings. If we only generate one parity chunk, then we can generate the parity matrix *ad hoc* as any single column matrix with non-zero coefficients. This gives us $2^f - 1$ possibilities for each parity matrix coefficient. In addition, by multiplying the columns, we can increase the possibilities further. Again, for security considerations, one has to keep in mind that any false guess by an attacker leads to a false answer to a challenge, and that, in the secure versions of our protocol, a malicious storage site cannot easily check the correctness of its made-up answers.

4. Signature-Based Challenges

Data origination sites typically want to check whether remote sites actually store the data entrusted to them. Often, origination sites maintain their own copy of the data. For such situations, Lillibridge, *et al.* [19] propose to ask the storage site to send a small, randomly-selected chunk of stored data to verify that the remote site is storing all of the data that has been sent to it. By selecting the start and end locations randomly, the test ensures that the storage site did not simply precompute and store signatures and discard the actual data. This approach has three drawbacks, however. First, the storage site must return the entire range of data, requiring a lot of upstream bandwidth. Second, this approach reveals the data; Lillibridge, *et al.* note that a malicious user could retrieve its data in this way. Third, the originating site must retain the original data for comparison, though this is not an issue if the originating site can ask *all* of the storage sites for the same range of their storage.

4.1. Basic Algorithm

We improve on existing approaches that require all of the data to be returned by asking the storing site to simply calculate and return an algebraic signature for a range chosen by the requester rather than the data that it covers. The originating site can verify that data has been stored correctly by combining this signature with signatures from other storing sites. Since the parity of the signatures is the same as the signature of the parity, the originating site can check the signatures and, if there is sufficient redundancy in the error-correcting code, even identify the site that presented an incorrect signature. Unlike previous approaches, these signature calculations do not require that the originating site retain the data for comparison; the signatures alone suffice for verification. Additionally, this approach reduces the amount of message data that must be sent and reveals little information about the original data, allowing a third party to conduct the verification without fear of compromising data. This approach is shown in Figure 1.

This simple procedure withstands two basic attacks. First, each response reveals some data and an attacker reading network traffic might be able to reconstruct the data, assuming that neither neither data nor messages are encrypted. As we will see, even if we do not use encryption, the information leakage is rather small because signatures are small and stored objects large. Second, and more importantly, a storing site could precompute answers to all possible challenges and store them instead of the

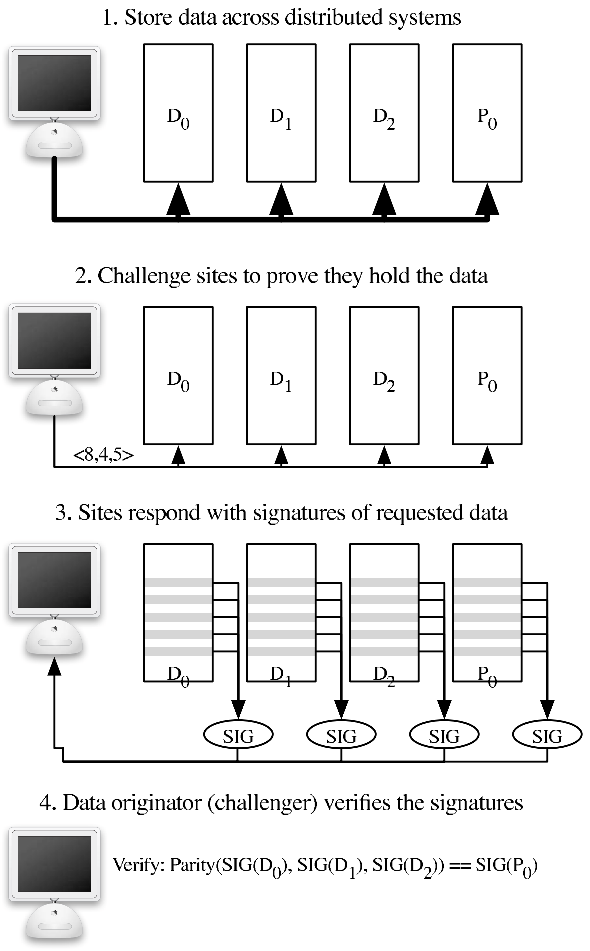
1. Store data across distributed systems
D0
D1
D2
P0
2. Challenge sites to prove they hold the data
<8,4,5>
3. Sites respond with signatures of requested data
SIG
SIG
SIG
SIG
4. Data originator (challenger) verifies the signatures
Verify: Parity(SIG(D0), SIG(D1), SIG(D2)) == SIG(P0)

Figure 1. Basic storage, signature request and verification protocol.

data. This only benefits the cheating storage site if the answers take less place than the actual data itself. This can be overcome by using a simple challenge request that supplies $\langle x,n,s\rangle$ for a given block, or set of blocks, where x is the starting offset, n is the number of samples, and s is the stride (all values in words). The storage site then calculates $sig[\alpha](d_x, d_{x+s}, d_{x+2s}, \ldots, d_{x+(n-1)s})$, where d_k is the k^{th} word of the block or set of blocks. Since the number of possible signatures is much higher than the actual size of the block, the storage site saves space by doing the right thing and simply storing the data rather than potential signatures. Note that, if needed, the number of signatures can be further increased by also specifying the α-parameter in the signature challenge.

Using the above scheme, the data originator can easily check that all of the remote data covered by the signatures has been stored correctly. The data originator, or anyone else who gets the signatures, can compute $parity[sig(d_0),\ldots,sig(d_{k-1})]$ and compare the result to $sig(parity)$, which was received from the system storing the parity for $d_0,\ldots,d_{k-1}$. Since parity calculations and algebraic signature operations commute, as discussed in Section 3, the two results should be equal. If they are not equal, and there is sufficient error correcting information, it is even possible to identify the signature that is incorrect. Note that anyone can perform this check, not just the data originator, since the verification does not require original data. In addition, the verification does not release much information about the original data, since only a few bytes of signature information are returned from each storage site. This prevents a malicious system from using this technique to retrieve data surreptitiously—it would take thousands of randomly-selected queries to retrieve sufficient signatures to be able to solve for the original data. If necessary, storage sites can refuse to answer "clustered" signature queries that might be used to derive the underlying data. Alternatively, storage sites may refuse to provide signatures for fewer than n of data words; recovery of the underlying data will require $O(n)$ signatures.

The technique we describe—calculating signatures on a portion of the data and returning just the signatures—can be done using more conventional hash algorithms such as MD5 and SHA-1. However, this approach suffers from a significant problem: the system requesting the signature must either keep the original data or precompute and store the signature values for any queries it plans to make. For example, a system might decide to precompute the hash values for 1,000 queries to each of twenty systems storing either data or parity. This approach could be done without algebraic signatures. However, if the 1,000 queries are exhausted, perhaps after a few months of use, the storage sites could simply remember the results of the previous 1,000 queries and discard the actual data, knowing that they would not receive any "new" queries for which they would have to retrieve data. Our approach has no such limitation because the results of any signature query can be verified mathematically, eliminating the need to precompute queries and results.

4.2. Collusion Resistance

The technique in Section 4.1, while it avoids the problems inherent in existing approaches, still suffers from a drawback: sites that collude can modify data or even make up signatures as long as they are internally consistent. In other words, a site receiving signatures can verify that the parity matches the data, but it cannot tell whether all of the sites colluded to provide fake signatures and then generated parity for them. To address this shortcoming, we developed a modification of the original protocol that allows a data originator to safeguard the signature verification process with a minimum of storage overhead.

In our storage scheme, we store data in containers of equal size. We form a reliability group out of m of these containers to which we add an additional k parity chunks of the same size, so that the group contains $n = m + k$ chunks. This configuration can tolerate k unavailable sites by recalculating any missing data chunks from a total of m data or parity chunk. It can also find up to $\lfloor k/2 \rfloor$ chunks in error and correct them. Even if we are not interested in erasure protection, we still generate some parity chunks to enable remote storage checking. The code used for the generation of the parity chunks uses the same Galois field as our signatures, but is different for each reliability group. It is always a linear, m/n code, and is described in detail in Section 3.2. We envision reasonably large chunks, so that the overhead of generating the codes is small. For example, we can derive the code from a secure hash function $h(object\_id, server\_secret, use\_parameter)$, where *object_id* identifies the reliability group, the server secret is used to prevent remote sites from deriving the same function, and *use_parameter* varies for each use of the hash function, as described below.

If enough of the storage sites collaborate, they can foil our scheme by first calculating the parity code, which amounts to solving a system of linear equations. To do so, they use about $m \cdot k$ symbols from the data and parity chunks to solve for the $m \cdot k$ coefficients of the generator matrix, as described in Section 3.2. An initial, and rather difficult challenge for the attackers would be to identify the data and parity chunks that make up a reliability group. A "security by obscurity" mindset would consider the difficulties in finding related storage blocks sufficient protection, but we can do better with very little overhead.

To prevent multiple sites from collaborating to fabricate consistent data and the signatures that go with them, we use two techniques: randomly-generated transformation matrices for parity, and "blinding" the parity values using a pseudo-random stream of data. The latter transformation is accomplished by using a stream-based encryption algorithm such as RC4 with a seed derived from the hash function described above. By blinding the parity information using a pseudo-random stream, multiple sites cannot uncover the parity code because they no longer have all of the values needed for the set of linear equations. The transformation functions must still be kept secret, however, since malicious sites that knew the transformation function but not the encryption stream could calculate the expected parity and thus recover the stream values, allowing them to substitute new values. It is only the combination of encrypted parity and secret parity matrices that prevents collusion.

The first step in generating parity under this scheme is to generate random parity matrices, as described in Section 3.2. These matrices are generated using the output of the hash function described above; each data chunk in each "row" has its own parity matrix. The specific matrix used for each chunk is thus a secret that the data originator knows, but that the storage sites do not know and have no way of solving for. Using these parity matrices, the originating site calculates the k^{th} parity word in the j^{th} parity chunk:

$$p_{j,k} = \hat{P}_{0,j} d_{0,k} \oplus \hat{P}_{1,j} d_{1,k} \oplus \cdots \oplus \hat{P}_{m-1,j} d_{m-1,k}$$

The parity chunk is then XORed with a pseudo-random stream generated by RC4 or a similar algorithm, seeded with a value derived from the above hash function; thus, the value stored for the parity word above would be $p_{j,k} \oplus r_{j,k}$, where $r_{j,k}$ is the k^{th} value of the pseudo-random stream for parity chunk j. If the chunks are large, it may be helpful to reseed RC4 every 4 KB or 8 KB to make verification faster.

Verification of the parity in this approach is similar to verification in the basic, insecure approach. As before, the data origination site sends a challenge of the form $\langle x, n, s \rangle$ to each of the storage sites and has them return the algebraic signature as before. However, the originating site must now remove the blinding factor from the parity signature. This is done by computing the signature of the values in the pseudo-random stream selected by $\langle x, n, s \rangle$ and XORing it with the parity signature. The result should be the parity of the data signatures. Since it is difficult to generate only the desired values from the random stream without generating the other values, this approach typically requires that the verifying system produce the entire pseudo-random stream from the start. By allowing the stream to restart every 4 KB or 8 KB, this time can be reduced by allowing the stream to be generated from the nearest block boundary.

Note that this technique blinds the parity, but stores the data unencrypted. It is also possible to store the data encrypted but leave the parity unencrypted, or to leave both the data and parity encrypted. If the data is to be stored encrypted, a data word stored as $d_{j,k} \oplus r_{j,k}$, where $r_{j,k}$ is the k^{th} word of the pseudo-random data stream for the j^{th} data chunk. Parity is still calculated on the raw, unencrypted data. This approach has nearly the same security as encrypting the parity; collaborating storage servers only have the encrypted data, but they need the unencrypted values as well as the $\hat{P}$ matrices to construct valid parity. However, when there are more parity chunks than data chunks. it would be possible to solve for the unencrypted data values; essentially, this would involve using $k > m$ cleartext parity chunks to reconstruct m cleartext data chunks. In such a situation, the parity chunks must be encrypted regardless of whether the data chunks are encrypted.

4.3. Verifying Storage on a Single Server

Because the collusion-resistant technique described in Section 4.2 cannot be broken by multiple entities, each of which holds part of the data, it is also resistant to subversion attempts from a single storage system. This property makes it attractive for storage verification for remote storage systems. Traditionally, remote storage on a single device is verified in one of two ways. First, the remote storage system might return an entire chunk of data. If this data is stored with appropriate protections [26, 30], the originating system can verify that the data was indeed stored and retrieved correctly and not corrupted. However, this approach suffers from the need to actually return the entire data chunk, and is not appropriate for verifying large amounts of storage because of the network bandwidth required. Typically, the entire chunk has to be retrieved because there is no way to verify less than a full chunk. A second approach would be to simply return the signature of a chunk rather than the data itself. As described earlier, however, this approach only requires that the storage server keep the signature, and provides no guarantee that the underlying chunk is actually stored.

Algebraic signatures provide a third alternative that has the advantages described earlier: verification of a relatively large data chunk by randomly sampling the data that make it up and returning the signature. Since the data is all stored on the same device, there is little benefit to having multiple parity chunks; a single parity chunk is sufficient to allow verification. A system that wants to store m data chunks computes parity across them as described in Section 4.2, and then stores the m data chunks and single parity chunk on the one server. The overhead is thus $1/m$, and can be very low if m is relatively large. Since all of the chunks are stored on the same server, m could be kept large by breaking up any chunk into m equal size pieces and computing parity across them.

To verify that the storage server is maintaining the data chunks, the data originator can send an $\langle x, n, s \rangle$ request to the storage server, which then computes and returns the signatures of the data chunks and parity chunk. These values are returned to the data originator, which can verify the integrity of the randomly-selected sequence of data as described in Section 4.2. As an added benefit, the data originator can correct errors in a single chunk if there are media errors on the storage system's disk.

5. Implementation and Performance Issues

We implemented a large number of different signature calculation algorithms to calculate compound signatures of size 4 B and of 8 B using underlying fields of $\mathscr{GF}(2^8)$ and $\mathscr{GF}(2^{16})$. We then tested them on two systems running

```
void MultiplyByAlpha(GFElement x)
{
  if(x != 0) {
     x = antilog[log[x]]+1];
  }
  return x;
}
```

Figure 2. Explicit multiplication by α method using logarithms and antilogarithms

Windows XP SP2: a desktop, with a 3 GHz Pentium 4 dual processor with 512 MB memory, and a laptop with a 2 GHz Pentium 4 Centrino processor and 1 GB of memory.

For completeness sake, we describe in broad strokes the best algorithms to calculate signatures. Recall that our signature consists of different component signatures. For example, if we calculate a 8 B signature using $\mathcal{GF}(2^8)$ over a (non-contiguous) substring B of a container, we actually calculate in parallel

$$(\text{sig}_1(B), \text{sig}_\alpha(B), \text{sig}_{\alpha^2}(B), \ldots, \text{sig}_{\alpha^7}(B)).$$

Here, the first signature, $\text{sig}_1(B)$, is the XOR of all the symbols in B.

We can of course calculate a component signature $\text{sig}_{\alpha^i}(B)$ according to its definition. It is slightly faster to calculate signatures from the back, i.e. to calculate $\text{sig}^{\text{op}}_{\alpha^i}(B) = \sum_{\nu=0}^{N-1} b_i \cdot \alpha^{N-1-\nu}$. We can now use a Horner scheme to calculate

$$\text{sig}_{\alpha^i}(B^{\text{op}}) = \left(\left((b_1\alpha^i + b_2)\alpha^i + b_3 \ldots\right)\alpha^i + b_{N-1}\right)\alpha^i + b_N$$

Thus, we build the signature processing a symbol at a time. At each step, we multiply the current result with α^i and then add the next symbol to it. Addition is done with XOR and hence fast; thus, the problem is fast multiplication by α^i.

One simple possibility for implementing multiplication by α is to use a table. This turns out to be efficient if the tables fit into the L1 cache, *i. e.*, for $\mathcal{GF}(2^8)$. For multiplication with α in $\mathcal{GF}(2^{16})$, we break a 16 bit symbol s into the left and the right part of 8 bits each. Using shift operations in C, we calculate them as $\mathsf{left}(s) = s \gg 8$ and $\mathsf{right}(s) = s\&0\text{xff}$, respectively. We generate two tables with entries between 0 and 0xff, tleft and tright, defined by $\mathsf{tleft}[i] = (i \ll 8) \cdot \alpha$ and $\mathsf{tright}[i] = i \cdot \alpha$. Therefore, $\alpha \cdot s = \mathsf{tleft}[\mathsf{left}(s)] + \mathsf{tright}[\mathsf{right}(s)]$. For example, if $s =$ 0xabcd, then $\mathsf{left}(s) =$ 0xab, $\mathsf{right}(s) =$ 0xcd, $\mathsf{tleft}[0\text{xab}] =$ 0xab00 $\cdot\, \alpha$, and $\mathsf{tright}[0\text{xcd}] =$ 0x00cd $\cdot\, \alpha$. It turns out that this "double table" method is somewhat faster than a single multiplication table because both tables can now reside in the L1 cache. However, we also need to optimize multiplication by powers of α. For example, if we calculate an 8 B signature over $\mathcal{GF}(2^8)$ we multiply with $\alpha, \alpha^2, \ldots \alpha^6$; the first component signature only uses XORing. We can save space and implement all multiplications by a power of α by successive look-ups to a single multiplication-by-α table. This strategy works reasonably well in the case of $\mathcal{GF}(2^{16})$, but results differ based on the cache sizes. For the smaller field $\mathcal{GF}(2^8)$, individual tables seem to be almost always better.

An alternative is the use of logarithms and antilogarithms. We pick an α that is a *primitive* element. Accordingly, each Galois field element β can be written as $\beta = \alpha^i$ and the power i—uniquely determined between 0 and $2^f - 1$—is the logarithm of β. Conversely, β is the antilogarithm of i. We can now exploit the properties of logarithms to calculate generic products with a logarithm and antilogarithm table. Figure 5 gives the idea in pseudocode applied to multiplication with α, which also extends to other powers of α. The test for zero can be avoided by defining $\log(0)$ to be a small, negative number and extending the antilogarithm table accordingly. Often, the resulting variant is slightly faster.

Another technique for signature calculations uses a scheme invented by Broder [4], and is based on the most basic way of defining multiplication in $\mathcal{GF}(2^f)$ as polynomial multiplication modulo a generator polynomial. The multiplication by the unknown X can then be performed by a left shift and XORing with an entity corresponding to the generator polynomial. We can identify α with the unknown so that multiplication by α consists of a left shift operation followed by an conditional XOR. It turns out that evaluating the condition is quite expensive and that the resulting implementation can have terrible runtimes. Broder recognized that one can perform several shift operations at once, consult a table that incorporates a number of decisions and use the table contents as the XOR-operand. The resulting implementations in general have excellent run-times, even though, in contrast to the table method, we must also multiply with X^2, and sometimes with X^3 and X^4 as well, since we use compound signatures.

In our experiments, particular implementations of Broder's method were always the best, but were sometimes equalled by table-based methods. Their speed came close to just calculating the 4 B XOR checksum of the data. Calculating a signature over 32 B in every 512 B block reached a throughput of over 900 MB/sec on the 2 GHz laptop and over 700 MB/sec on the desktop machine, falling to about 40 MB/sec on both once data had to be accessed from the disk. Based on our experiments, the performance bottleneck for algebraic signatures is clearly the data transfer rate between disk and memory.

6. Conclusions and Future Research

In this paper, we introduced the use of algebraic signatures to check on distributed data stored outside of the owner's control. Algebraic signatures offer two primary advantages. First, algebraic signatures compress a large block in a short byte string. The byte string can be made large enough to make an accidental fit extremely unlikely. For example, a 32 bit signature will suffer a collision with probability 2^{-32} and a 64 bit signature with probability 2^{-64}. This compression saves network bandwidth. Second, algebraic signatures interact well with linear error and erasure correcting codes. Thus, algebraic signatures may be used to verify that the parity and the client data are coherent without the need to obtain the entire data and parity. Furthermore, if the underlying code has error correcting capabilities, then the signatures alone can diagnose which data is incorrect, perhaps from an incorrectly performed update or a malicious storage site.

We use these properties to verify remote storage in two different ways. If data is stored across a number of sites using an m/n erasure correcting code, the use of a linear code to generate the $k = n - m$ parity chunks permits challenges based on algebraic signatures to check whether the storage sites together store the same data. Using the error correcting capacity of such a code, we can also determine that one or a few sites which are in error. Second, and perhaps more intriguing for a system such as OceanStore, we can check whether a set of colluding sites or even a single site accurately stores the data entrusted to it. We do so by breaking the object stored there into equal size data chunks, adding a few parity chunks to it, and blinding all the data stored there by XORing with a pseudo-random stream. Our scheme then allows issuing challenges that only a site that faithfully stores its data can answer correctly; it is probabilistically impossible for a site that does not know the secrets to generate a coherent set of signatures. This allows inexpensive verification of large amounts of data with relatively low network communication costs.

Our signature scheme is flexible enough to easily generate such a diversity of challenges that storing precomputed data instead of the data makes no sense to a cheating storage site. At the same time, each signature is small and thus reveals little information about the stored data, even if signatures are sent in cleartext. Furthermore, the signature generation process is fast, running at tens to hundreds of megabytes per second. Thus, this approach can permit the verification of very large quantities of remote storage with minimal network bandwidth and acceptable amounts of computation at the remote site.

We are currently implementing a simple peer-to-peer storage system that uses algebraic signatures for verification, and expect to measure the reduction in bandwidth as well as the increased ability to do verification in such a system. We are also developing a scheme that uses algebraic signatures to check on the consistency of replicated databases. It might even be possible to consider the use of signatures as a means of concurrency control, since they can detect inconsistencies between data and parity information.

While algebraic signatures are unsuitable as cryptographically secure hash functions such as MD5 or the SHA family of secure checksums, they are ideally suited for use in verifying remotely stored data in distributed systems. The combination of low network bandwidth, reasonable computation load, and resistance to malicious modification make algebraic signatures ideal for verifying that data entrusted to remote storage systems is actually being maintained. By allowing the verification of large amounts of stored data with minimal network impact, algebraic signatures have the potential to enable very large-scale verifiable distributed storage systems.

Acknowledgments

We would like to thank the faculty and students of the Storage Systems Research Center for their help and guidance. Support for this research was provided by SSRC industrial partners, including Engenio, Hewlett Packard, IBM, Intel, Microsoft, Network Appliance, Rocksoft, Veritas, and Yahoo.

References

[1] A. Adya, W. J. Bolosky, M. Castro, R. Chaiken, G. Cermak, J. R. Douceur, J. Howell, J. R. Lorch, M. Theimer, and R. Wattenhofer. FARSITE: Federated, available, and reliable storage for an incompletely trusted environment. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation (OSDI)*, Boston, MA, Dec. 2002. USENIX.

[2] M. Blaum, J. Brady, J. Bruck, and J. Menon. EVEN-ODD: An efficient scheme for tolerating double disk failures in RAID architectures. *IEEE Transactions on Computers*, 44(2):192–202, 1995.

[3] M. Blaum, P. G. Farrell, and H. C. A. van Tilborg. Array codes. In V. S. Pless and W. C. Huffman, editors, *Handbook of Coding Theory*, volume 2. North-Holland, Elsevier Science, 1998.

[4] A. Z. Broder. Some applications of Rabin's fingerprinting method. In R. Capocelli, A. D. Santis, and U. Vaccaro, editors, *Sequences II: Methods in Communications, Security, and Computer Science*, pages 143–152. Springer-Verlag, 1993.

[5] A. Z. Broder. On the resemblance and containment of documents. In *Proceedings of Compression and Complexity of Sequences (SEQUENCES '97)*, pages 21–29. IEEE Computer Society, 1998.

[6] A. Z. Broder, S. C. Glassman, M. S. Manasse, and G. Zweig. Syntactic clustering of the web. In *Proceedings of the 6th International World Wide Web Conference*, pages 391–404, Santa Clara, California, United States, Apr. 1997.

[7] G. Caronni and M. Waldvogel. Establishing trust in distributed storage providers. In *Proceedings of the Third International Conferences on Peer-to-Peer Computing*, pages 128–133. IEEE, Sept. 2003.

[8] I. Clarke, O. Sandberg, B. Wiley, and T. W. Hong. Freenet: A distributed anonymous information storage and retrieval system. *Lecture Notes in Computer Science*, 2009:46+, 2001.

[9] P. Corbett, B. English, A. Goel, T. Grcanac, S. Kleiman, J. Leong, and S. Sankar. Row-diagonal parity for double disk failure correction. In *Proceedings of the Third USENIX Conference on File and Storage Technologies (FAST)*, pages 1–14, 2004.

[10] F. Dabek, M. F. Kaashoek, D. Karger, R. Morris, and I. Stoica. Wide-area cooperative storage with CFS. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP '01)*, pages 202–215, Banff, Canada, Oct. 2001. ACM.

[11] F. Dabek, B. Zhao, P. Druschel, J. Kubiatowicz, and I. Stoica. Towards a common API for structured peer-to-peer overlays. In *Peer-to-Peer Systems II: Second International Workshop, IPTPS 2003, Springer Lecture Notes in Computer Science 2753*, pages 33–44, Berkeley, CA, USA, Feb. 2003.

[12] E. Gabber, J. Fellin, M. Flaster, F. Gu, B. Hillyer, W. T. Ng, B. Özden, and E. Shriver. StarFish: Highly-available block storage. In *Proceedings of the Freenix Track: 2003 USENIX Annual Technical Conference*, pages 151–163, San Antonio, TX, June 2003.

[13] A. V. Goldberg and P. N. Yianilos. Towards and archival intermemory. In *Advances in Digital Libraries ADL'98*, pages 1–9, April 1998.

[14] G. R. Goodson, J. J. Wylie, G. R. Ganger, and M. K. Reiter. Efficient Byzantine-tolerant erasure-coded storage. In *Proceedings of the 2004 International Conference on Dependable Systems and Networking (DSN 2004)*, June 2004.

[15] M. C. Harrison. Implementation of the substring test by hashing. *Communications of the ACM*, 14(12):777 – 779, December 1971.

[16] L. Hellerstein, G. A. Gibson, R. M. Karp, R. H. Katz, and D. A. Patterson. Coding techniques for handling failures in large disk arrays. *Algorithmica*, 12:182–208, 1994.

[17] R. M. Karp and M. O. Rabin. Efficient randomized pattern-matching algorithms. *IBM Journal of Research and Development*, 31(2):249–260, Mar. 1987.

[18] J. Kubiatowicz, D. Bindel, Y. Chen, P. Eaton, D. Geels, R. Gummadi, S. Rhea, H. Weatherspoon, W. Weimer, C. Wells, and B. Zhao. OceanStore: An architecture for global-scale persistent storage. In *Proceedings of the 9th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Cambridge, MA, Nov. 2000. ACM.

[19] M. Lillibridge, S. Elnikety, A. Birrell, M. Burrows, and M. Isard. A cooperative Internet backup scheme. In *Proceedings of the 2003 USENIX Annual Technical Conference*, pages 29–42, San Antonio, TX, 2003.

[20] W. Litwin, R. Moussa, and T. Schwarz. LH*RS – a highly-available scalable distributed data structure. *ACM Transactions on Database Systems*, 30(3):769–811, 2005.

[21] W. Litwin and T. Schwarz. LH*$_{RS}$: A high-availability scalable distributed data structure using Reed Solomon codes. In *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data*, pages 237–248, Dallas, TX, May 2000. ACM.

[22] W. Litwin and T. Schwarz. Algebraic signatures for scalable, distributed data structures. In *Proceedings of the 20th International Conference on Data Engineering (ICDE '04)*, pages 412–423, Boston, MA, 2004.

[23] M. G. Luby, M. Mitzenmacher, M. A. Shokrollahi, and D. A. Spielman. Efficient erasure correcting codes. *IEEE Transactions on Information Theory*, 47(2):569–584, February 2001.

[24] F. J. MacWilliams and N. J. Sloane. *The Theory of Error Correcting Codes*. Elsevier Science B.V., 1983.

[25] P. Maniatis, M. Roussopoulos, T. J. Giuli, D. S. H. Rosenthal, and M. Baker. The LOCKSS peer-to-peer digital preservation system. *ACM Transactions on Computer Systems*, 23(1):2–50, 2005.

[26] E. L. Miller, D. D. E. Long, W. E. Freeman, and B. C. Reed. Strong security for network-attached storage. In *Proceedings of the 2002 Conference on File and Storage Technologies (FAST)*, pages 1–13, Monterey, CA, Jan. 2002.

[27] A. Muthitacharoen, R. Morris, T. M. Gil, and B. Chen. Ivy: A read/write peer-to-peer file system. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation (OSDI)*, Boston, MA, Dec. 2002.

[28] M. O. Rabin. Fingerprinting by random polynomials. Technical Report TR-15-81, Center for Research in Computing Technology, Harvard University, 1981.

[29] M. O. Rabin. Efficient dispersal of information for security, load balancing, and fault tolerance. *Journal of the ACM*, 36:335–348, 1989.

[30] E. Riedel, M. Kallahalla, and R. Swaminathan. A framework for evaluating storage system security. In *Proceedings of the 2002 Conference on File and Storage Technologies (FAST)*, Monterey, CA, Jan. 2002.

[31] M. Ripeanu, A. Iamnitchi, and I. Foster. Mapping the Gnutella network. *IEEE Internet Computing*, 6(1):50–57, Aug. 2002.

[32] A. Rowstron and P. Druschel. Storage management and caching in PAST, a large-scale, persistent peer-to-peer storage utility. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP '01)*, pages 188–201, Banff, Canada, Oct. 2001. ACM.

[33] T. Schwarz. Verification of parity data in large scale storage systems. In *Proceedings of the 2004 International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA '00)*, Las Vegas, NV, 2004.

[34] T. J. Schwarz, R. W. Bowdidge, and W. A. Burkhard. Low cost comparison of files. In *Proceedings of the 10th International Conference on Distributed Computing Systems (ICDCS '90)*, pages 196–201, 1990.

[35] L. Xu and J. Bruck. X-code: MDS array codes with optimal encoding. *IEEE Transactions on Information Theory*, 45(1):272–276, 1999.