Efficient Verification for Provably Secure Storage and Secret Sharing in Systems Where Half the Servers Are Faulty

Rida A. Bazzi [*] and Goran Konjevod [**]

Computer Science and Engineering Department
Arizona State University
Tempe, AZ 85287-8809
{bazzi,goran}@asu.edu

Abstract. We present new decentralized storage systems that are resilient to arbitrary failures of up to a half of all servers and can tolerate a computationally unbounded adversary. These are the first such results with space requirements smaller than those of full replication without relying on cryptographic assumptions. We also significantly reduce share sizes for robust secret-sharing schemes with or without an honest dealer, again without cryptographic assumptions. A major ingredient in our systems is an information verification scheme that replaces hashing (for storage systems) or information checking protocols (for secret sharing). Together with a new way of organizing verification information, this allows us to use a simple majority algorithm to identify with high probability all servers whose information hasn't been corrupted.

Keywords: Secure storage, secret sharing, Byzantine failures

1 Introduction

In a distributed storage system subject to failures, redundancy needs to be introduced so that data can be correctly retrieved. A simple form of redundancy is full replication. For example, t crash failures can be tolerated by replicating the data on $t+1$ servers, while t Byzantine failures can be tolerated by replicating the data on $2t+1$ servers. To retrieve the data, a client reads from all the servers and accepts the value returned by a majority. One can think of replication as a form on *information verification:* the value stored at one server is used to verify the information stored at another. If one server is correct and the value it stores is identical to the value at another server, then the value at the other server is also correct. This is the basis for using majority to retrieve the correct value. In general, the verification information should make it impossible or hard (in

[*] Research supported in part by the National Science Foundation grants CCR-9972219 and CCR-9876952.

[**] Research supported in part by the National Science Foundation grant CCR-0209138.

an information-theoretic or a computational sense) for a server to provide corrupt information without being detected. This is the *no cheating* requirement for verification information.

In a secret sharing scheme [1], a dealer has a secret and distributes its *shares* to n participants in such a way that any $t+1$ shares can be used to recover the secret, but any t shares provide no information about it. A secret sharing scheme consists of two protocols executed by the dealer and the participants. In the *sharing* phase the dealer distributes the shares, and in the *reconstruction* phase the participants *open* their shares (make them available to each other) in order to reconstruct the secret. Robust secret sharing schemes [2] are those that can tolerate faulty participants. In these schemes, the dealer provides participants with verification information that allows detection of corrupt shares during the reconstruction. The definition of secret sharing imposes an additional requirement on the verification information held by any participant: it should leak no information about the *share* (data) that it verifies. This is the *no leakage* requirement for verification information.

The size of the verification information is an important performance measure. In a distributed storage system, the ratio of the total space used to store the data to the size of actual data is called the *space blow-up*. (For full replication, this ratio is equal to n.) By using error-correcting codes [3, 4], the blow-up can be reduced to $n/(n-2t)$, where $t < n/2$ is the number of faulty servers (it is clear that in general, a correct majority is needed to retrieve the stored data). In a system of n servers, if $2t$ is close to n, this factor becomes $\Omega(n)$ and in fact, Krawczyk [4] proves that $n/(n-2t)$ is the optimal space blowup unless the solution allows for a positive probability of error. He introduces a *distributed fingerprinting* scheme based on one-way hash functions to circumvent this lower bound. The space blow-up of his method is $n/(n-t)$ in the size of the data file, but it introduces an overhead of $nh/(n-2t)$ per server, where h is the size of a fingerprint. If $n = 2t+1$, the space blow-up is 2 and the fingerprinting overhead is nh per server. The additional overhead can be relatively large if the file is not large. Alon et al. [5] further reduce the fingerprinting overhead by using a verification structure where each server stores verification information for data at other servers. They succeed in achieving a blow-up of $2+\epsilon$ plus an overhead of $\Theta(\log nh)$ per server for $n \geq 2t+1$, with a hidden constant larger than 1500. Recently, the same authors of [5] developed a scheme that significantly reduces the hidden constant [6]. The schemes of Krawczyk and of Alon et al. use one-way hash functions, so they all rely on unproven assumptions. Hence, the probability of error introduced by their use of hash functions cannot be quantified. By reducing the blow-up factor, the schemes of both Krawczyk and Alon et al. lose an important property of schemes based on error correcting codes: they cannot tolerate computationally unbounded adversaries (for lack of space, we omit the proof of this fact).

The situation is similar for secret sharing. There is a very large body of literature [7], and the vast majority of schemes consider models where information about the secret is not leaked in a computational sense and where faulty participants have bounded computational power. If $n < 3t$, then there is no solution to the problem that does not have a probability of error. Rabin [2] presented a scheme that tolerates $t < (n-1)/2$ faulty participants, but assumes the existence of a broadcast channel to be used by the participants and the dealer. She gave solutions for the case of a correct dealer as well as for the case of a faulty dealer, allowing correct reconstruction of a secret with high probability, without any cryptographic assumptions and in the presence of a computationally unbounded adversary. In her scheme, each server stores n different pieces of verification information, one for each other server, with each piece as large as the share (data) being verified.

Existing work leaves open three important questions for systems in which $n \geq 2t + 1$:

1. Is there a distributed storage scheme that doesn't depend on unproven assumptions, cryptographic or other, and whose space requirements are smaller than those of error correcting codes?
2. Is there a distributed storage scheme that can tolerate a computationally unbounded adversary, and whose space requirements are smaller than those of error correcting codes?
3. Is there a secret sharing scheme that does not depend on unproven assumptions and provides information-theoretic secrecy with high probability, but whose space requirements are smaller than those of Rabin [2]?

We answer all questions in the affirmative by introducing two new techniques for information verification in the presence of a computationally unbounded adversary. The first technique, which we call *private hashing*, allows a server p to verify information of size s of another server q by using $O(hms^{1/m})$ private bits at p and h bits at q. If q changes its information, this will be detected with probability $1-(1-1/2^m)^h$ (for $m = 2$, the detection probability is $1-(3/4)^h$). Also, p learns nothing about q's information. Our numbers are to be compared to at least $2s$ bits of information at p and at least s bits of information at q that are needed by Rabin's scheme, with detection probability $1-(1/2)^s$.

Our second technique organizes the verification information in such a way that each server needs to verify only $O(\log n)$ others—in other schemes that tolerate computationally unbounded adversaries, each server verifies the information of all others. This technique relies on the fast majority vote algorithm of Boyer and Moore [8]. (We organize the verification information in such a way that a modified form of the Boyer-Moore algorithm can be used even though every server can only verify $O(\log n)$ others.)

These techniques allow us to achieve the following results.

For distributed storage, we present a scheme with *total* space usage $2S + O(nhm(\log n + k)S^{1/m})$, where S is the total data size, m is a constant and k and h are security parameters. The failure probability is no more than $(9/4)n(1-2^{-m})^h + (1/2)^k$. When we take $h \in \Theta(\log n)$ to reduce the error probability to an arbitrary constant $\epsilon < 1$, these numbers guarantee a blow-up less than 2.1 and space overhead in $O((\log n)^2)$, which is independent of the data size. In comparison to the previously proposed schemes, we reduce the storage requirements significantly without relying on unproven assumptions, and still tolerate computationally unbounded adversaries. Our scheme tolerates an adaptive adversary that can choose which servers to corrupt next based on publicly available information and the private information of already corrupted servers. Our scheme does not tolerate a fully adaptive adversary—the difference is that a fully adaptive adversary can look at a server's private data and then decide whether or not it wants to corrupt it. In practice, the adaptive adversary model is the most powerful model of interest because we can consider a server compromised as soon as any of its private data is compromised. As in all previously proposed schemes, we assume that the clients are correct, but unlike other schemes we also require that the readers not be subject to Byzantine failures (we discuss this requirement in Section 2).

For secret sharing, we present a scheme with share size $s + O(\log^2 nms^{1/m})$, for any constant m, where s is the secret size, for the case of a correct dealer. This scheme relies on private hashing and our modification of the Boyer-Moore algorithm. This provides a significant improvement over Rabin's share size of $(3n + 1)s$. Our secret sharing schemes are formulated in a model identical to that of Rabin, namely an adaptive computationally unbounded adversary and an arbitrarily small probability of error. A fully adaptive adversary does not make sense for secret sharing; if the adversary could access servers' private information, it could defeat any scheme by reconstructing the secret. Even in the case of a faulty dealer, we can use our hashing procedure for verification in place of Rabin's verification procedure and reduce the share size from $s + sf(n)$ to to $s + s^{1/m}f(n)\text{polylog}(n)$ for any constant m (here, f specifies the overhead associated with the secret-sharing scheme).

2 System Model

Storage System and Secret Sharing. The storage system consists of n servers $s_1, \ldots, s_n$ and is accessed by external clients. Each server may be *correct* or *faulty*, but the number of faulty servers is at most $(n-1)/2$. The correct servers work as intended throughout the period in which the system is accessed. The faulty servers may fail in any arbitrary way at any time. For example, they may stop responding to requests, modify the data stored on them, or collude with other faulty servers to modify the data and attempt to mislead the clients that access them. In secret sharing, the servers are called participants. The dealer can be thought of as a client and the participant themselves become clients when the secret is opened.

Communication and Synchrony. We assume reliable and private communication channels between the clients and the servers. This is the same model assumed in Rabin's paper [2] and is standard in unconditionally secure secret sharing schemes. Assuming private communication channels is not standard for the secure storage problem, where existing solutions assume authenticated channels. In practice, private channels can be implemented using encryption and authentication, but this is not the only way to implement private channels, so assuming private channels does not imply assuming encrypted channels. We assume a synchronous system, in other words, that it is possible to detect non-responding servers. Even though it has not been explicitly stated in previous work, this assumption is needed to tolerate $(n-1)/2$ arbitrary failures.

Clients. In this work, we assume clients that access the storage system are correct. A writer will correctly follow its protocol and a reader will not divulge private information that it collects during a read operation. The assumption of a correct reader is not significantly stronger than that of a correct writer. In practice, it is not hard to check that a reader will only send read requests and nothing else to servers. This can be enforced locally by restricting read requests to use a well defined interface or by requiring them to go through a trusted proxy. The correct reader assumption only affects the assumptions for the secure storage system, and, as mentioned in the introduction, is not an issue in the model for secret sharing.

Adversary. We consider an adaptive adversary that can choose which servers to corrupt next based on publicly available information and private information of already corrupted servers. The adversary has unbounded computation power. We do not assume a fully adaptive adversary that can decide what to do next based on private information of non-corrupted servers.

3 Efficient Information Checking

In information checking, there are four participants: the dealer, the recipient, the checker, and the verifier (In Rabin's scheme, the recipient is called intermediary, the checker is called the recipient that also functions as a verifier). The dealer sends verification information V to the checker. Also, the dealer sends data S and leakage-prevention information r to the recipient. At a later time, the recipient and the checker pass the information they received to a verifier. Information checking should satisfy the following two properties:

1. No cheating. If the dealer, checker, and verifier are correct, then the recipient cannot provide incorrect data to the verifier without being detected with very high probability.
2. No leakage. If the dealer is correct, then V leaks no information about S.

In our scheme, the verification information V consists of a *private hash* value H and a random selection pattern $Select$. The size of V is considerably smaller than the size of S. Hashing is done recursively by dividing S into pieces, then combining the hash values of the pieces to obtain the hash of S. The space and time complexity of the hashing function depends on a parameter m that determines the depth of the recursion, or number of levels of hashing. We first describe the 1-level base case, then we describe the m-level case.

Figure 1 shows the 1-level and m-level procedures used by the dealer to calculate a single bit of H; to calculate h bits, the same procedure is repeated h times. The h applications of the algorithm are independent and so are the random patterns generated for each of the h bits.

```
Hash($S$: bit string of size $s_m = k^m$, $Select$: matrix of $m \times k$ bits, $r$: bit)

    Hash$_1$($select, S, start, end$)
        Hash$_1 = 0$
        **for** $i = start$ **to** $end$ **do**
            **if** $select[i]$ **then**
                Hash$_1$ = Hash$_1 \oplus S[i]$

    Hash$_m$($S, start, end$)
    1:  **if** $m = 1$ **then**
    2:      Hash$_m$ = Hash$_1$($Select[1], S, start, end$)
    3:  **else**
    4:      Hash$_m = 0$
    5:      $s_{m-1} = (end - start + 1)/k$
    6:      **for** $i = 0$ **to** $k$ **do**
    7:        **if** $Select[m][i]$ **then**
    8:          Hash$_m$ = Hash$_m \oplus$ Hash$_{m-1}$($S, start + is_{m-1}, start + (i+1)s_{m-1} - 1$)

**begin**
    Hash = Hash$_m$($0, size - 1$) $\oplus r$
**end**
```

Fig. 1. Calculating 1-bit hash value

To produce a single hash bit in 1-level hashing, we calculate the XOR of a randomly selected subset of bits of S. In the function Hash$_1$, a contiguous set of bits of S starting at *start* and ending at *end* is hashed. To hash all of S, we use *start* $= 0$ and *end* $= \text{size}(S) - 1$ and these are the values used when Hash is called with the number of levels m equal to 1. The function Hash$_1$ has a parameter *Select*, which is a uniformly random string of 0's and 1's, and is used to determine which bits of S are XOR-ed together.

After the appropriate bits are XOR-ed together, a random bit r is XOR-ed with the result in order to prevent leakage; with this addition to the protocol, a checker that stores a hash bit for S cannot learn anything about S.

In m-level hashing, a string of size k^m is divided into k strings of size k^{m-1} each. Then $(m-1)$-level hashing is applied (recursively) to each of the k strings, and finally 1-level hashing (lines 6–8) to the resulting k bits. In the recursion, the selection patterns are not independent: the k hashings at level $(m-1)$ all use the same $(m-1) \times k$ sub-matrix of the $m \times k$ *Select* matrix. The 1-level hashing done at level m (lines 6–8) uses the remaining $1 \times k$ sub-matrix of the *Select* matrix.

To summarize: in order to hash an s-bit string into h bits, we use a pattern consisting of $mhs^{1/m}$ bits, and an additional h random bits. The hash can be calculated in no more than mhs steps. Our final algorithm (Section 4.2) requires $2(\log n + k)$ hashes to be stored by each server. The total overhead for calculating all of these is thus $O(hms^{1/m} n \log n)$, where m is an arbitrary constant and s the size of the data. If $h \in \Theta(\log n)$, this reduces to $O(ms^{1/m} n \log^2 n)$.

Lemma 1. *Let Select be an $(m \times k)$-bit matrix where each bit is generated independently uniformly at random. Let r be a bit and S a k^m-bit string, and $H = \text{Hash}(S, \mathit{Select}, r)$ the hash value of S. For k^m-bit string $S' \neq S$ and any bit r', the probability (over the random choice of Select) that*

$$\text{Hash}(S, \mathit{Select}, r) = \text{Hash}(S', \mathit{Select}, r') \tag{1}$$

is at most $p_m = 1 - 2^{-m}$.

Proof. Let $v = r' \oplus r$. Then by the definition of the function Hash, (1) holds if and only if $\text{Hash}_m(S, 0, k^m - 1) \oplus \text{Hash}_m(S', 0, k^m - 1) = v$.

Since the same pattern *Select* is used for S and S', $\text{Hash}_m(S, 0, k^m - 1) \oplus \text{Hash}_m(S', 0, k^m - 1) = \text{Hash}_m(S \oplus S', 0, k^m - 1)$.

To prove the statement of the lemma, we prove by induction on m that for any bit v, the probability that $\text{Hash}_m(S \oplus S', 0, k^m - 1) \neq v$ is at most $1 - 2^{-m}$.

For the base case ($m = 1$), let C_1 be the (non-empty) set of positions on which S and S' differ and let A_1 be the set of positions that are *selected* by Hash_1. Then $\text{Hash}_1(S \oplus S', 0, k^m - 1) = v$ if and only if the parity of $|A_1 \cap C_1|$ is the same as the parity of v. Since A_1 is random, it follows that $A_1 \cap C_1$ is a random subset of C_1. Thus the probability that $|A_1 \cap C_1|$ is even (or odd) is exactly 1/2. In other words, in the base case the probability that (1) holds is exactly 1/2.

For the induction step, consider the loop in lines 6–8. The function Hash_{m-1} is applied to k groups $S_1, \ldots, S_k$ of bits of S, resulting in k bits, some of which are then XOR-ed together. The choice of which among the k bits will be XOR-ed together is determined by the k-bit vector $\mathit{Select}[m]$.

Let $C_m = \{i \mid \text{Hash}_{m-1}(S_i) \neq \text{Hash}_{m-1}(S'_i)\}$.(We are abusing the notation slightly here by writing S_i as a parameter of Hash_{m-1} instead of specifying S_i as a subset of S using *start* and *end*.) Since $S \neq S'$, there is an i^* such that $S_{i^*} \neq S'_{i^*}$. The probability that C_m is nonempty is at least the probability that $i^* \in C_m$. By the induction hypothesis, $\text{Hash}_{m-1}(S_{i^*}) \neq \text{Hash}_{m-1}(S'_{i^*})$ with probability at most $1 - 2^{-(m-1)}$ (note that $\text{Hash}_{m-1}(S_{i^*}) \neq \text{Hash}_{m-1}(S'_{i^*})$ if an only if $\text{Hash}_m(S \oplus S', 0, k^m - 1) \neq 0$).

In fact, if C_m contains more than one element, this probability will be even smaller, but in any case it is at most $1 - 2^{-(m-1)}$. Call this probability p_{m-1}. Let $A_m = \{i \mid \mathit{Select}[m][i] = 1\}$, that is, the set of level $m-1$ hash bits that are selected by Hash_m to calculate the hash bit in line 7. Let $J_m = C_m \cap A_m$. Clearly, $\text{Hash}_m(S \oplus S', 0, k^m - 1) = \bigoplus_{i \in A_m} (\text{Hash}_{m-1}(S_i) \oplus \text{Hash}_{m-1}(S'_i)) = |J_m| \mod 2$.

The above expression is equal to v if and only if the parity of $|J_m|$ is. Since A_m is random, it follows that $J_m = A_m \cap C_m$ is a random subset of C_m. Thus the probability that the parity of $|J_m|$ is equal to v is exactly 1/2 if C_m is nonempty. If C_m is empty, then the $|J_m| = 0$. Thus for $v = 0$, the probability that $|J_m| \mod 2 = v$ is $(1/2) \cdot p_{m-1} + 1 \cdot (1 - p_{m-1})$. For $v = 1$, on the other hand, this probability is $(1/2) \cdot p_{m-1} + 0 \cdot (1 - p_{m-1})$. In both cases, $|J_m|$ is of equal parity as v with probability $p_m \leq p_{m-1}/2 + (1 - p_{m-1})$. This is maximized for $p_{m-1} = 1 - 2^{-(m-1)}$, which gives $p_m \leq 1 - 2^{-m}$ and proves the induction step.

Lemma 2. *(No cheating.) When using level-m private hashing, if the dealer, checker and verifier are correct, then the recipient cannot provide incorrect data to the verifier without being detected with probability* $1 - (1 - 2^{-m})^h$*, where* h *is the length of the verification information.*

Proof. Follows from the description of information checking and Lemma 1 by noting that the h hash bits are independent.

Lemma 3. *(No leakage.) In private hashing, assuming the dealer is correct, the verification information leaks no information about* S*.*

Proof. The matrix $Select$ is generated randomly, independently of S, and $H = \mathrm{Hash}(S, Select, r)$ is an XOR with the (uniform and unknown to the checker) random string r, and thus random and independent of S as well.

4 Storage and Recovery

Distributed Storage When storing the data, a server stores the data pieces computed using IDA [9] with blow-up factor equal to 2. The IDA scheme assumes that data cannot be tampered with, so we need to add extra verification information to detect pieces that have been tampered with and recover the original data. The verification information kept at each server verifies the data as well as the verification information kept at other servers. To retrieve the data, the reader collects all the information (data and verification information) from all the servers. Then, the reader executes an algorithm that enables it to identify (with high probability) a majority of servers whose information has not been tampered with. A basic operation is for the reader to check whether the verification information obtained from one server correctly verifies the information it's supposed to verify. If the verification information has not been tampered with, then with high probability the verified information is correct. The details of the particular algorithm and verification checks are given in subsequent sections.

Secret Sharing We only present the case in which the dealer is correct. In the sharing phase, the dealer computes the shares using [1] and sends them to the participants; In addition the dealer send the participants verification information for the shares of other participants. In the reconstruction phase, every participant sends all the information it has to all other participants. Then, each participant executes an algorithm that enables it to identify (with high probability) a majority of servers whose information has not been tampered with. This will enable all correct participants to reconstruct the secret.

We start by presenting a scheme in which every server verifies every other server. Then we present a scheme in which every server verifies only $2(\log n + k)$ other servers, where k is a security parameter. This will allow us to achieve the results listed in the introduction.

4.1 Full verification

In this section we describe a verification scheme in which each server contains information to verify each other server. In this scheme, the servers are arranged on a line from left to right.

```
0: for $i = 1$ to $n$ do
1:     $V[i, 0]$ = IDA data piece
2: for $i = 2$ to $n$ do
3:     for $j = i - 1$ downto $1$ do
4:         $V[i, j] = H(V[j], R[j])$
5:         $R[j, i]$ = the random bits computed in line 4 to prevent leakage
6: for $i = n - 1$ downto $2$ do
7:     for $j = i + 1$ to $n$ do
8:         $V[i, j] = H(V[j])$
```

Fig. 2. Full Verification

We can divide the verification information at a given server into two types. The *left-verification* information of a server p is the verification information that p keeps for servers to its left. The *right-verification* information of a server p is the verification information that p keeps for servers to its right. The left-verification information of a server p_r verifies *all* the left-verification information of a server p_l to its left. The right-verification information of a server p_l verifies *all* the verification information (both left- and right-) of a server p_r to its right. We say that the information on two servers is *consistent* if each of the two servers verifies the information of the other server. We will abuse notation and say that the *servers are consistent* and they are related by the *consistency* relation. The algorithm for calculating the verification information is shown in Figure 2. In the algorithm,$V[i, j]$ is i's verification information for j and $R[i, j]$ are the random bits kept at j to prevent leakage to i. In the algorithm, $V[j]$ refers to all of j's verification information at the point it is used in the computation. Similarly we define $R[j]$. In line 4, $V[j]$ is j's left verification information because at that point only j's left verification information has been computed. In line 8, $V[j]$ is j's total verification information.

Majority elements. A simple approach to recovery would have a reader check that the server's information is consistent with the information of a majority of servers. This will guarantee that all correct servers will pass the check and their information can be used in recovery and also that no information that has been tampered with will be used. Unfortunately, this simple approach will lead to a quadratic number of verifications at recovery time (for now, we will ignore the actual cost of checking whether two servers are consistent). Our goal is to reduce the number of consistency checks to $2n$. We will first find a server p_c whose information is guaranteed to be correct and then find all the correct servers among the subset of servers whose information is verified by p_c. To achieve a linear number of checks, we modify the well-known Boyer-Moore linear-time algorithm [8] for finding a majority element in an array. The linear-time majority element algorithm uses only equality tests. In our setting, we do not have an equality test, but we can check if the information on two servers mutually verifies each other. The consistency relation does not have the same transitivity properties as equality and so we modify the majority algorithm so that the transitivity properties of consistency are sufficient.

Transitivity of verification. Here we prove the necessary transitivity properties of the verification information.

Lemma 4 (Correctness). *Let p, q be two consistent servers such that p appears before q on the verification line and the information of p is correct. All the information at q is then correct with high probability.*

Proof. Since, p appears to the left of q, it follows that p verifies all the information of q. Since, p is correct, by Lemma 2, if q provides incorrect information it will be verified by p with probability at most $(1-2^{-m})^h$, where h is the size of the hash.

Lemma 5 (Right Transitivity). *Let $p_1, \ldots, p_u$ be a sequence of servers that appear in that order on the verification line and such that p_i and p_{i+1} are consistent, for all $1 \leq i \leq u-1$. If p_1 is correct, then all the information on servers p_u, is correct with probability $1-(u-1)(1-2^{-m})^h$.*

Proof. If all consecutive pairs of servers are consistent and p_u is incorrect, then one of the verifications along the line (say p_i by p_{i-1}) must have failed. For each i, by Lemma 4, the verification of i by $i-1$ fails with probability $(1-2^{-m})^h$. Thus the probability that there is a failure in at least one of the $u-1$ verifications is $(u-1)(1-2^{-m})^h$.

Lemma 6 (Left Transitivity). *Let $p_1, \ldots, p_u$ be a sequence of servers that appear in that order on the verification line and such that p_i and p_{i+1} are consistent, $1 \leq i \leq u-1$. If p_u is correct then all the left-verification information on servers p_1 is correct with probability $1-(u-1)(1-2^{-m})^h$.*

Proof. The proof is similar to the proof of Lemma 5 and is omitted.

In what follows, we assume that the security parameter h is chosen so that the probability of failure is small enough. We will later calculate a value of h that works.

Lemmas 5 and 6 imply that if p_j is correct for some $1 \leq j \leq u$, then all the data on servers $p_1, \ldots, p_u$ is correct with high probability.

The lemmas we have proved enable us to find a server whose data and verification information is correct using the algorithm shown in Figure 3.

The algorithm is almost identical to that of Boyer and Moore [8], but with one important difference. In Boyer and Moore's algorithm, there is no need for

```
1: $count := 0$
2: for $i = 1$ to $n$ do
3:    if $count = 0$ then
4:        $correct := i$
5:        $count := 1$
6:    else
7:        if $p_i$ and $p_{correct}$ are consistent then
8:            $correct := i$
9:            $count := count + 1$
10:       else
11:           $count := count - 1$
```

Fig. 3. Finding a correct server

```
0: for $i = 1$ to $n$ do
1:    $V[i, 0]$ = IDA data piece
2: for $i = 2$ to $n$ do
3:    for $j = i - 1$ downto $\max(1, i - \ell)$ do
4:        $V[i, j] = H(V[j], R[j])$
5:        $R[j, i]$ = the random bits computed in line 4
                  to prevent leakage
6: for $i = n - 1$ downto 2 do
7:    for $j = i + 1$ to $\min(i + \ell, n)$ do
8:        $V[i, j] = H(V[j])$
```

Fig. 4. Verification on an ℓ-Path

the assignment on line 8, whereas in our algorithm, the assignment is crucial to guarantee that the transitivity lemmas can be used. The following theorem (whose proof is omitted for lack of space) shows that the algorithm in Figure 3 will find a correct server (a server whose information has not been tampered with) if a majority of the servers is correct.

Theorem 1. *If a majority of the servers is correct, then with high probability the information of server $p_{correct}$ at the end of the main loop of algorithm of Figure 3 is not tampered with.*

The algorithm allows us to find a server $p_{rightmost}$ with correct data. To find all such servers (this includes all correct ones), we find all servers consistent with $p_{rightmost}$ (note that no server to the right of $p_{rightmost}$ is correct). Finding all correct servers in a set of n thus takes no more than $2n$ consistency checks, therefore the probability that the servers identified as correct really are so is at least $1 - 2n(1 - 2^{-m})^h$.

4.2 Efficient Verification

We present a scheme that relies on the Boyer-Moore majority algorithm, with each server verifying only a small subset of other servers, and achieves very good performance. The scheme guarantees with high probability that the set of correct servers and other servers whose data is not corrupted is identified. It requires $2(\log(n) + k)$ hash values of size h per server to provide a probability of failure at most $(9/4)n(1 - 2^{-m})^h + 2^{-k}$.

The verification graph. The verification graph we use is not complete. In the verification algorithm, the servers are arranged in a line $s_1, s_2, \ldots, s_n$ and s_i and s_j verify each other by if and only if $|i - j| \leq \ell$. The resulting graph is called the *ℓ-th power of a path* on n vertices and is denoted by P_n^ℓ. We calculate the verification information as shown in Figure 4, where ℓ is a parameter that determines how many servers each server verifies.

Since each server is verified by only $\ell \ll t$ other servers, an adversary could corrupt all of them and make the now isolated server useless even if it is correct. To prevent this, the writer hides the ordering of the servers in the line by giving each of them a random, unique and private ID. The IDs are used by the reader during reconstruction. However, they are not public information prior to the reconstruction phase and so are unknown to the adversary, who can only learn a server's ID after corrupting the server. Thus the adversary cannot choose a particular ID and then corrupt the server with that ID. So the corrupted server's IDs are selected independently at random. The algorithm must take into account the situations in which corrupted servers report incorrect IDs.

Since at most $t < n/2$ servers are ever corrupted, the probability that a given server is faulty is less than $1/2$, regardless of whether its neighbors in the line are corrupted or not. Thus, given $i \in \{1, 2, \ldots, n - \ell\}$, the probability that ℓ servers immediately following the i-th one are faulty is less than $2^{-\ell}$. If we choose $\ell = 2(\log n + k)$, then the probability that no server is followed by ℓ faulty servers is bounded above by $1/2^k$. It follows that with probability at least $1 - 1/2^k$, in P^ℓ, the set of all correct servers forms a connected component. In what follows we assume without referring to probability (except, of course, when computing the probability of success of the algorithm) that there is no contiguous subsequence of faulty servers of length more than ℓ.

Boyer-Moore Majority for Path Powers. The algorithm is based on the Boyer-Moore majority procedure, but some modifications are necessary to make it run with high probability. First, in Figure 5 we show the initialization.

Since the corrupted servers can advertise themselves under fake names, it may happen that several servers give the same value as their ID. Clearly, at most one of these is correct. During initialization, the servers are sorted into buckets, according to the IDs they report. Each bucket is maintained in the form of a linked list. The (arbitrary) order of servers in each bucket induces, together with the order on the buckets (according to the IDs reported by the servers), an order on the set of servers. It is this order that the algorithm uses to examine the servers one by one.

We assume throughout that there is no contiguous subsequence of more than ℓ buckets containing only faulty servers. This means that even though perhaps we have examined more than ℓ servers between one that reports i as ID and

```
1:  for $i := 1$ to $n$:                        // (1–2): Initialize buckets
2:      $list[i] :=$ null
3:  for $i := 1$ to $n$:                        // (3–7): Sort servers into buckets,
4:      if $list[server[i].\mathrm{ID}] =$ null     // thus creating linked lists.
5:          $last[server[i].\mathrm{ID}] := server[i]$
6:      $server[i].next := list[server[i].\mathrm{ID}]$
7:      $list[server[i].\mathrm{ID}] := server[i]$
8:  $i := 1$                                    // (8–11): Find the first nonempty bucket.
9:  while ($list[i] =$ null) do
10:     $i := i + 1$
11  $start.next := list[i]$; $startindex := i$
12: $i := n$                                    // (12–15): Find the last nonempty bucket.
13: while ($list[i] =$ null) do
14:     $i := i - 1$
15: $endindex := i$
16: for $i := startindex$ to $endindex - 1$// (16–22) Initialize the remaining
17      if ($list[i] \neq$ null)                  // next pointers.
18:         $j := i + 1$
19:         while ($j \leq n$ and $list[j] =$ null) do
20:             $j := j + 1$
21:         if ($j > n$) break
22:         $last[i].next := list[j]$
```

Fig. 5. Finding a correct server: initialization

one that reports $i + j$ as ID, as long as $j < \ell$, we expect the two servers to have verification information for each other. However, it may still happen that the basic algorithm (Figure 3) tries to cross-verify a pair of servers not adjacent in the verification graph. We argue that in such a case, we can drop some of the servers, backtrack slightly and continue the algorithm, and still with high probability find a correct server. The algorithm is presented in Figure 6.

Underlying the implementation is the notion of a *consistent component*. During the execution of the algorithm, whenever the variable *count* is positive, the algorithm knows a set of at least *count* consistent servers that induce a connected subgraph of the verification graph. The algorithm maintains several pieces of information about the current consistent component: the variable *count* stores the number of consistent servers in the component; *correct* stores the head of the consistent component (the latest server added); $firstnon$ and *lastnon*, respectively, point at the first and last element (as ordered by the position in the path) known not to belong to the component; the *next* pointers link the servers known to be inconsistent with the component into a list.

The main actions the algorithm performs are the following: **(1)** Whenever *count* reaches 0, a new consistent component is started. **(2)** Whenever a server p is found to be inconsistent with the head of a component, the last *non-belonging* server (*lastnon*) is linked to p and *lastnon* updated to point to p. **(3)** If the head of the component cannot be checked for consistency against the current server

```
$i := start$; $correct := start.next$; $count := 0$; $firstnon :=$ **null**
**while** ($i \neq$ **null**) **do**
    $i := i.next$
    **if** ($|i.\mathrm{ID} - correct.\mathrm{ID}| > \ell$)
        $count := 0$
        $i := firstnon$
        $firstnon :=$ **null**
        $correct := i$
    **if** $count = 0$ **then**
        $correct := i$
        $count := 1$
    **else**
        **if** $i$ and $correct$ are consistent
            $count := count + 1$
            $correct := i$
        **else**
            $count := count - 1$
            **if** ($firstnon =$ **null**)
                $firstnon := i$
                $lastnon := i$
            **else**
                $lastnon.next := i$
                $lastnon := i$
```

Fig. 6. Finding a correct server

under consideration, the algorithm concludes that the whole component must be faulty (as justified below) and restart the algorithm from $firstnon$.

Note that in the case where more than one server uses the same ID, if one of them is found consistent with the current component, the others servers will be found inconsistent and will consequently be ignored (unless the component is subsequently discarded).

We show (using an amortization argument) that the total number of consistency checks is linear (with a small hidden constant) in the number of servers.

Theorem 2. *If a majority of the servers is correct, then, with high probability, the information of server $p_{correct}$ at the end of the main loop of algorithm of Figure 6 is not tampered with.*

Storage Overhead. Since we use patterns to hash strings consisting of, among other things, other patterns, the size of the pattern required is not immediately obvious. According to the rules for storing verification information, each server will store its share of data (size s), the pattern (size p), the $2(\log n + k)$ hashes (each of size h), and the $2(\log n + k)$ random strings (each of size h) used for the data hashed by verifying servers. Thus the total size of data that needs to be hashed by verifying servers is $s + 4h(\log n + k) + p$. Given a string of size s to be hashed into a string of size h, the pattern is of size $hs^{1/m}$. Hence the size of the pattern p must satisfy the inequality $p \geq h(s + p + 4h(\log n + k)^{1/m}$.

Under the very reasonable assumption that $\sqrt{s} \geq 4h + 6$, it is enough that $p = hs^{1/m}(\log n + k)^{1/m}$ (we omit the derivation for lack of space).

Running Time. It may not be obvious that the removal of a component and another invocation of the algorithm (and, recursively, possibly more than one) can be performed in a total linear number of consistency verifications.

Theorem 3. *The total number of consistency checks made in the algorithm of Figure 6 before a correct server is found is at most* $5n/4$.

In addition to these $5n/4$ checks, another n may be necessary to find all correct servers. Each consistency check takes time proportional to h times the size of the data stored at a server.

Failure Probability. The algorithm could fail in one of two ways: either one of the consistency checks fails with a false positive, or there is a contiguous sequence of corrupted servers. The first happens with probability $(1 - 2^{-m})^h$ for each check, and the second with probability $(1/2)^k$. Since there are at most $(9/4)n$ consistency checks, the total probability of failure is no more than $(9/4)n(1 - 2^{-m})^h + (1/2)^k$. With $h \in \Theta(\log n)$, this probability can be made an arbitrarily small constant.

References

1. Shamir, A.: How to share a secret. Communications of the ACM **22** (1979) 612–613
2. Rabin, T.: Robust sharing of secrets when the dealer is honest or cheating. Journal of the ACM **41** (1994) 1089–1109
3. MacWilliams, F.J., Sloane, N.J.A.: The Theory of error-correcting codes. North-Holland (1977)
4. Krawczyk, H.: Distributed fingerprints and secure information dispersal. In: Proceedings of the 12th annual ACM symposium on principles of distributed computing. (1993) 207–218
5. Alon, N., Kaplan, H., Krivelevich, M., Malkhi, D., Stern, J.: Scalable secure storage when half the system is faulty. Information and Computation **174** (2002) 203–213
6. Alon, N., Kaplan, H., Krivelevich, M., Malkhi, D., Stern, J.: Addendum to "Scalable secure storage when half the system is faulty". Unpublished Manuscript (2003)
7. Stinson, D., Wei, R.: Bibliography on secret sharing schemes (1998) Available on the Internet at http://www.cacr.math.uwaterloo.ca/˜dstinson/ssbib.html.
8. Boyer, R.S., Moore, J.S.: MJRTY—a fast majority vote algorithm. In Boyer, R.S., ed.: Automated reasoning: Essays in honor of Willy Bledsoe. Kluwer (1991) 105–117
9. Rabin, M.O.: Efficient dispersal of information for security, load balancing and fault tolerance. Journal of the ACM **36** (1989) 335–348