Erratum: 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

1 Introduction

The protocol presented in the paper that appears in the DISC 2004 proceedings relies on a transitivity lemma that does not necessarily hold. The lemma states that if p is a correct server, and q and r are two other servers such that p verifies q and q verifies r, then r's data should be correct. Serge Fehr pointed to us that this does not hold if q and r are incorrect and collude. In fact, while q's verification will be correct with high probability, if q and r are corrupted, then q can give its own data and verification information to r beforehand so that r can calculate different data that is guaranteed to be verified by q's verification information.

Also, Ronald Cramer pointed out that our adversary model is not the strongest for secret sharing (it is the strongest adversary for secure storage though). In the secret sharing model, one can distinguish between *rushing* and *non-rushing* adversaries [1]. A rushing adversary may wait for the correct players to show their shares before showing the shares of the faulty players. It is not clear to us that Rabin [2] makes this distinction between the model we use and the stronger model of [1], but the protocol that she presents clearly tolerates the stronger model as shown in [1]. In [1], Cramer et al. prove a lower bound for the rushing adversary model which basically states that in any one-round solution that tolerates a rushing adversary, each server must verify $\Omega(n)$ other servers. The rushing adversary model is not relevant to the secure storage problem because all servers send their data over private channels to the reader and faulty servers cannot see the data and verification information of correct servers.

In this erratum, we change the transitivity lemma and the protocol for recovering the file. The new protocol solves all the questions that were posed in the paper. Its space requirement is identical to that of the protocol in the proceedings, but its running time is slightly higher. It is important to note that the

* 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.

main ingredients of the solution did not change, but our solution does not use the fast majority algorithm.

On the other hand, we also have an extension for secret sharing that tolerates a rushing adversary. This extended solution requires multiple rounds, but it overcomes the lower bound of [1]. We do not describe this extended solution in this erratum.

2 Full Information Verification

We first modify how full verification is done. Instead of calculating *right* and *left* verification information that are tied together (right verification information verifies left verification information) we calculate *right* verification information and left verification information independently[1].

0: **for** $i = 1$ **to** n **do**
1: $V[i, 0] =$ Share of Secret
2: **for** $i = n - 1$ **downto** 2 **do**
3: **for** $j = i + 1$ **to** n **do**
4: $V[i, j] = H(V[j], R[j])$
5: $R[j, i] =$ the random bits computed in line 4 to prevent leakage

Fig. 1. Full Right Verification

The *right-verification* information of a server p is verification information that p keeps for servers to its right. The right-verification information of a server p_l verifies *all* the verification information of a server p_r to its right. We say that p verifies q if the verification information of p verifies the information at q. The algorithm for calculating the verification information is shown in Figure 1. 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 $R[j]$.

We define *left verification* information in a similar way. The only difference is that it is calculated from right to left.

2.1 Simple Reconstruction

In Rabin's [2] solution, every server verifies every other server. So, a simple reconstruction scheme will keep only those shares that are verified by a majority of servers (and therefore by a correct server). This guarantees that all correct servers will pass the check and their information can be used in recovery and also

[1] We could have kept the full verification as it appears in the proceedings version, but we wanted to emphasize that they need not be linked together

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 a server verifies another server). Our goal is to reduce the number of verification checks to $O(nl)$, where $l \in o(n)$.

2.2 Transitivity of verification.

Here we prove the necessary transitivity properties.

Lemma 1 (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.

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.

Lemma 2 (Transitivity). *Let p, q, and r be three servers that appear in that order on the verification line. If p and r are correct and p verifies q, then q verifies r with probability at least $1 - (1 - 2^{-m})^h$.*

Proof. q would not verify r if its information is incorrect. But the information of q is incorrect with probability at most $(1 - 2^{-m})^h$.

It should be clear that these lemmas do not rely on the direction of verification and that one can prove two similar lemmas for left verification. We omit the details.

2.3 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 verifies s_j, $j > i$ if and only if $j - i \leq \ell$. The resulting graph is called the *ℓ-th power of a directed path* on n vertices and is denoted by P_n^ℓ. We calculate the right verification information as shown in Figure 2. We calculate the left verification information in a similar way.

Since each server is verified by at most $\ell \ll t$ other servers , an adversary could corrupt all of the servers that verify a given server which makes the information of the now-*isolated* server useless even if it is correct. To prevent this, the distribution phase hides the ordering of the servers in the line by giving each of them a random, unique and private ID. The IDs are used during reconstruction.

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

Fig. 2. Right verification on an ℓ-path

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 also 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 server i are faulty is less than $2^{-\ell}$. If we choose $\ell = \log(n)+k$, 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.

3 Finding the Correct Servers

For a server i, we define the following sets.

- $Right(i)$: the set of servers that are verified by server i.
- $Left(i)$: the set of servers that verify server i.
- $Left\_Consistent(i)$: the set of servers that verify server i and whose verification information correctly verify server i. Servers in $Left\_Consistent(i)$ are to the left of i.

Figure 3 presents an algorithm that finds servers consistent with a given server. The algorithm has one main loop. In each iteration of the loop, a new server is added if it is verified by all servers already in the component that are adjacent to it (in the verification graph). The claim is that at the end of the loop, if p_c is correct, then *R_correct_component* must contain all correct servers to the right of p_c and the data and verification information of all servers in *R_correct_component* have not been tampered with. We prove the following theorem.

correct_component : doubly linked list of servers
correct_component = NULL
add p_c to component. **for** $i = p_c$ **to** n **do**
  if $Left(i) \cap R\_correct\_component \subseteq Left\_Consistent(i)$ **then**
    add i to *R_correct_component*
    $count = count + 1$

Fig. 3. Finding a right component consistent with p_c

Theorem 1. *If p_c is correct, then at the end of iteration i R_correct_component contains all correct servers in the interval $[p_c, i]$. Also, the data and verification information of servers in R_correct_component is correct.*

Proof. The proof is by induction on i.

Base case: $i = p_c$ and *R_correct_component* has only p_c which is the only correct server in $[p_c, p_c]$.

Induction Step: Consider the *R_correct_component* at the end of iteration i. We need to show that if server i is added to *R_correct_component*, then its data and verification information are correct. Also, we need to show that if server i is correct, then it will be added to *R_correct_component* in iteration i. There are two cases to consider.

1. $i - p_c \leq l$. It follows that all elements in *R_correct_component* are verified by the correct server p_c and that all the data and verification information of servers in *R_correct_component* are correct. In particular, the data and verification information of server i are correct if i is added to the component. If i is correct, then it will be added to the component because the verification information of all servers in *R_correct_component* is correct and server i will be correctly verified by the information of servers in *R_correct_component*.
2. $i - p_c > l$. By assumption, one of the l left neighbors of i, say q, is correct. By induction, q must be in *R_correct_component*.
   - If i is not in *R_correct_component* but is nonetheless added to the the component, then i must be correctly verified by q because q is in $Left(i) \cap R\_correct\_component$. It follows that all the data and verification information of i is correct.
   - if i is correct, then i must be added to *R_correct_component*. In fact, all the data and verification information of servers in *R_correct_component* are correct and therefore q will be verified by all servers in $Left(i) \cap$ *R_correct_component*.

Similarly, we can have an algorithm to find a left component of servers to the left of p_c and that are consistent with it (*L_correct_component*). By an analogous proof, we can show that at the end of the execution, if p_c is correct, then the *L_correct_component* contains all correct servers to the left of p_c. So, if p_c is correct, the union of the *L_correct_component* and *R_correct_component* contains all correct servers.

4 Recovering the File

The results of the previous section provide a way to find a majority of servers whose data is correct if p_c is correct. To recover the file, the reader finds n components (left and right), one for each server in the system. The components of correct servers (whose starting server is correct) contain all correct servers and other servers whose data is correct. So, the file recovered using the components of the correct servers must be identical. To recover the file, the reader recovers the file n times, once for each component, and finds the file that is recovered a majority of times; this is the correct file.

In the secret sharing problem, each player recovers the secret independently of the others, so it can recover the secret using its own component. If the player is correct, its component will result in the correct secret. If the player is faulty, we have no constraints on the secret it recovers.

5 Space Requirements

Every servers verifies $\lg(n) + k$ other servers and the verification structure is identical to that of the protocol in the Proceedings paper. Thus, the space requirements are identical to those of the protocol in the Proceedings. It should be noted that reducing the space requirement is the main goal of the work, so we still achieve our claims with this modified solution.

6 Running Time

In finding the correct components, it takes $\lg(n) + k$ verifications to decide if a server is to be added to the component. In contrast, in the algorithm that appears in the Proceedings, it takes only one verification to decide if a server should be added to a component (the components in this erratum are not identical to the components in the protocol in the Proceedings). So, this introduces a factor of $\lg(n) + k$ to the running time of that algorithm. Also, the component algorithm should be executed n times in the worst case, which introduces another factor of n. Thus, the running time of the algorithm is $O(n \lg(n)$ times slower than that presented in the proceedings.

7 Acknowledgments

We would like to thank Serge Fehr for pointing out that our earlier solution has a serious error. We would like to thank Ronald Cramer for pointing out the distinction between rushing and non-rushing adversaries. Ronald Cramer also pointed out that we could replace our private hashing function with authentication codes (A-codes). As it turns out, our private hashing function is a special case of A-codes and there are A-codes in the literature with smaller space requirements than our private hashing function; using such codes in our solution

would further reduce the memory requirements of our solution. One advantage of our hashing function is that hashing requires s XOR operations to execute, where s is the size of the file. It is not clear to us if there are other A-codes in the literature that can be computed as efficiently and have small space requirements.

References

1. Cramer, R., Damgaard, I., Fehr, S.: On the cost of reconstructing a secret, or vss with optimal reconstruction phase. In: Proceedings of the 21st Annual International Cryptology Conference on Advances in Cryptology, Springer-Verlag (2001) 503–523
2. Rabin, T.: Robust sharing of secrets when the dealer is honest or cheating. Journal of the ACM **41** (1994) 1089–1109