LH*$_{RE}$ with Cached Encryption Keys: A Scalable Distributed Data Structure with Recoverable Encryption

Sushil Jajodia[1], Witold Litwin[2] & Thomas Schwarz[3]

(Work in Progress Report, June 2008)

Abstract

LH*$_{RE}$ is a Scalable Distributed Data Structure, first described in [JLS8]. We designed it to store many records protected through key encryption in a scalable file. As often, key management and in particular preventing the loss of keys is of utmost importance. LH*$_{RE}$ is an LH*-scheme, hence stores records on behalf of clients on any number of servers addressed through the scalable distributed hashing of the record identifiers. LH*$_{RE}$ client encrypts each record using client chosen secret key cryptography. The client also encodes each key using the secret-sharing. Shares are stored at servers, different for each share and randomly chosen. The client chooses the secret size, i.e., the number of shares. An attacker of a key has to locate all the secret-sharing servers. The task appears overwhelming for an LH*$_{RE}$ file, typically on many servers. An authorized user may recover or revoke in contrast rather rapidly every key, lost, corrupted or of a missing encryptor. LH*$_{RE}$ frees in this way the users from the well-known daring chores of the client-side key maintenance.

The LH*$_{RE}$ description in [JLS8] specifies a basic scheme and sketches variants for future work. The basic scheme uses a new encryption key for every record, minimizing the number of disclosed records if the unthinkable happened. The storage overhead of encryption, while different for each variant, is always negligible. The basic scheme has the highest insert, update, delete, and search message count costs. In [JLS8a], we analyze a variant lowering search and update costs. Below, we discuss next variant lowering also record insert and delete message count costs. The overall result is the message count per operation of basic LH*, i.e., without any encryption related message count overhead. For some applications, this may be a compelling advantage. The client caches the encryption key space with, at will, a single key for all the records or with a large number, e.g., a million keys. The key space can scale, as well as the secret size. We discuss the scheme algorithmic, performance and design criteria, including the rationale for choosing the key space size.

1. Introduction

More and more applications can benefit from scalable distributed data structures (SDDSs) for their storage needs. An SDDS stores data on a number of servers, which gracefully adjusts to the data size. Google reference count in thousands shows growing interest in the technology[4]. Example applications, where scalability became a buzzword, involve data management in companies, health systems, personal data servers, remote backup, Web services like Simple Storage of Amazon.com, archive systems, P2P social networks…. Many of these applications have strict confidentiality needs. However, the servers used by the SDDS might not be under the administrative control of the owner of the data or would need to be administered more securely. For example, an owner or administrator (or the "owner" of a root-kited system) can dump and analyze all local data. Secret key encryption provides an efficient tool for protecting the confidentiality of data stored in a possibly hostile environment. However, key management is a well-known drawback to using this technology. Loss and leakage of keys are often prelude to a disaster. Furthermore, records and hence their keys might be long lived, adding to the key management challenge. One current solution is to release keys to some third party escrow system that safeguards the keys and provides key recovery upon request [B3], [PGP4]. This idea has not yet popularly adopted and is currently not easy to use. Another approach uses secret sharing on the records instead of encrypting them, such as in the Potshards scheme, [SGMV7]. Potshards is explicitly designed for long-term archival storage that needs to survive several generations of secret key schemes (e.g. AES replacing DES) without re-encryption. Its drawback is the large storage overhead, which is not acceptable for data that is currently in use and located in storage that is more expensive. In the very last sentence of the generic article on the subject, Wikipedia conjectures that a way out is perhaps a key based encryption of large data objects, with the secret sharing limited to the safety of the key storage [W8]. Keys are usually much smaller than records,

[1] Center for Secure Information Systems (CSIS), George Mason University Fairfax, VA 22030-4444, jajodia@gmu.edu

[2] U. Paris Dauphine, Pl. du Mal. de Lattre, 75016 Paris, France, Witold.Litwin@dauphine.fr

[3] Thomas Schwarz, S.J. Dep. of Computer Eng. Santa Clara University, Santa Clara CA, tjschwarz@scu.edu

[4] 15 000 as by June 15, 2008 for query "scalable distributed data structure" OR "scalable distributed data structures"

hence the storage overhead of secret sharing should greatly decrease accordingly. Current industrial practices advocate encryption by the data host, and not the data owner, unless "data security is paramount", [S8], [C8]. It is questionable how users feel about the resulting lack of control.

LH*$_{RE}$ is a new tool for the needs we discussed above, [JLS8], [JlS8a]. As an LH* scheme, it is scalable, [LNS96], [LMS5], [LSY8]. In LH*$_{RE}$ the user defines encryption, but is relieved from the task of managing keys. LH*$_{RE}$ protects keys by secret sharing. Users can choose from a spectrum defined by the extremes of using a single key per record or one key for all records. Finally, an authorized (trusted) LH*$_{RE}$ client can recover any encryption key and therefore encrypted record, regardless of collaboration by the owner (encrypting party). This allows an authorized party to access the records of an unavailable client or a client to recover its own keys.

LH*$_{RE}$ creates (k+1) shares of keys for each client key. We simply use the now classic XOR-based scheme by Shamir [S79]. LH*$_{RE}$ provides each share with a dedicated primary key and stores the shares using the LH* addressing scheme accordingly. The scheme guarantees that the shares are always on (k+1) different nodes. While records migrate in any SDDS, these migrations will *never* result in shares being located on the same server. While the majority of the SDDS accesses are direct, the scheme also prevents that shares might end up on the same server in transit.

As a result, an attacker bent on reading a record needs to successfully penetrate (k+1) servers, unless the attacker can access the record only and break the encryption. We assume that we can trust the client node [MMJ3], in particular that it is immune against attack including message sniffing. Only a massive series of break-ins will leak confidential data, since in addition LH*$_{RE}$ does not give an indication where the key shares are stored so that the attacker has to break into the vast majority of storage sites.

In [JLS8], we have designed the basic LH*$_{RE}$ scheme and sketched variants for future analysis. The storage overhead incurred is always small, whenever, as usual, keys are small relative to records. The basic scheme provides a new encryption key for every record. Its message count costs associated with record insertion, update delete or search are about (k+2) times those of LH*. In [JLS8a], we have analyzed a variant that lowers the message counts of a search and of an update to those of LH*. The variant uses an additional share, termed private. The client caches this one locally and uses it for the private encoding of the encryption key, stored with the record. An additional benefit is the secret-sharing for the encrypted record itself. If the record was, e.g., a testament, one could decrypt it only by all the heirs together.

In the remainder of the paper, we analyze an LH*$_{RE}$ variant that lowers the message counts of all manipulations of records such as key-based searches, inserts, updates and deletes. The messaging cost is that of LH* and thus optimal in the sense of absence of encryption related message count overhead. The variant uses predefined keys. The client uses secret sharing for each key. It also caches all keys in local storage, preferably in RAM. Unlike in the basic scheme, the same key can encrypt several records. The user has flexibility of choosing between encryption of individual records with about individual keys or using the same key for many, if not all records. This choice is one of balancing *assurance* and *disclosure*. The former measures the expectation that an intrusion of l servers does not disclose data. The letter measures the amount of data disclosed in a successful intrusion. The key space can be scalable, providing, in particular for the desired ratio of records per encryption key used, hence for the desired disclosure amount. The secret size can scale as well. This may help to prevent the assurance deterioration in a scaling file.

We discuss below the file structure and its manipulation, including the algorithms for the key recovery and revocation. We also discuss the performance factors and the file design criteria. We finally indicate the directions we continue to investigate.

The properties of LH*$_{RE}$ with cached encryption keys appear attractive. However, our analysis is still in an early stage. Space issues force us to assume some familiarity of the reader with LH* schemes, as presented, e.g., in [LNS96], [LMS5] or [LSY7].

2. File Manipulation

2.1. File Structure

An LH* file stores data structured into records with primary keys and non-key fields. Records in an LH*$_{RE}$ file are application data records and key shares. It is possible to distinguish these two data types, but we cannot give these details here. Records are stored in buckets numbered 0,1,2… Each bucket is located at a different server (node). Initially, a typical LH* file is created with bucket 0 only, but then grows to more buckets through bucket splits. In contrast, an LH*$_{RE}$ file is created with at least k+1 buckets, i.e. buckets 0, 1, …, k, k+1. In previously analyzed variants, k could be zero or started with k+2 initial nodes to avoid collocating a record with any key share. Starting with more initial buckets enhances the inability of an attacker to find shares of a given key. An LH* file (including an LH*$_{RE}$ file) spreads over more buckets through a series of bucket splits. In each bucket split, about half of the splitting bucket's records move to a new bucket. All applications access records through an LH*$_{RE}$ client node. Each client has a unique ID. Typically, several clients share a file. The client does not store any application data. It's generic LH* role is to manage the dialog with the servers. It's LH*$_{RE}$ specific role is the encryption of data on behalf of an application and the encoding of the encryption keys. We give the details below. The client is also responsible for sending out queries and the records to store. Record search, insert, update, and delete queries are key-based. We use a dynamic hash function h that calculates the record location given a certain number of buckets over which the database is spread.

LH* splits are not posted synchronously to the clients. A client may be therefore unaware of the actual file extent. The address calculation may send in consequence a query to an incorrect (outdated) location. We recall that LH* locates nevertheless any record with at most two additional hops. Also, the great majority of requests reach the correct buckets directly. Moreover, if all nodes are peers (combine the role of server and client) then at most one additional hop suffices [LSY8]. LH* addressing is indeed faster in this sense than that of any other known SDDS and DHT-based schemes[5]. If there are hops, the LH* client gets from the servers an Image Adjustment (IAM) message. IAMs prevent a client from committing the same addressing error twice.

2.2. Encryption Key Caching and Encoding

The client has the *encryption key cache* with the capacity to store $N \geq 1$ keys. The client uses the cached keys, and *only these keys*, in this variant, to encrypt data records. N is user or administrator defined. N can be static, or may scale as we sketch below. The encryption is symmetric, e.g., AES. The client generates the keys. They could be also application defined, yet another goal of future analysis. The client generates key values at random, e.g., using white noise. Specifics of key generation are not part of the scheme; though they are of course important [CJHR5]. Any proven method is welcome. The cache itself is a one-dimensional table **T** [0..N-1]. The client inserts the N keys to the cells in any desired way.

The client caches and encodes every encryption key prior to any use. The encoding uses the secret-sharing into $k + 1$ shares, $k = 1,2\ldots$ The choice of k may differ among the clients. It reflects a client's *assurance* that the disclosure of any data by intrusion into any ($k + 1$) nodes is very unlikely, [MZ8]. Different users may have different estimates of the assurance. Regardless of the approach, higher k usually increases assurance.

In addition to the keys, the client generates (k+1) random and different values $C_1\ldots C_{k+1}$. These values are the primary keys of share records, i.e. the records that contain a share of a key. Given the LH* generic principles, each C_i should be unique, i.e., different from any data record key or share key. The $C_1\ldots C_{k+1}$ are chosen to be placed in different buckets. The client tests therefore each generated share record key for a collision and resolves this by generating a new key. The client formats the k shares into records $S_i = (C_i, T, I, N_i)$. Here, T denotes the offset of the encoded key in **T**. We need the T-field for key recovery, as we discuss below. Next, I denotes some identity of the client, or more generally, any information provable upon request that the future requestor of the record is entitled to access it. The choice of I value and the authentication method are not parts of the LH*$_{RE}$ scheme, e.g., any well-known approach will do. Finally, each N_i is a different white noise, generated by the client. The client forms also share S_{k+1} as (C_i, T, I, E'') with $E'' = N_1 \oplus \ldots \oplus N_k \oplus E$, where ($\oplus$) denotes XOR. Finally, the client sends out each S_i. Since we are using linear hashing, the client calculates $h(C_i)$ based on its current image and sends it to the resulting bucket.

Share record generation, encoding, and storing should be a single atomic transaction. Otherwise, a client might use a key that is not maintained and hence unrecoverable.

2.3. Data Record Insertion

To insert a data record R with primary key C, the client starts with the encryption of the non-key field of R. For this purpose, for $N > 1$, the client applies some hash function h_T mapping every record key C into [0, N-1]. For instance, one can set for $h_T(C) = C \bmod N$ with possibly N being chosen to be a power of 2. Values of N may vary among the clients, but h_T has to be the same for all clients in order to enable key recovery, as we will see below. The encryption is symmetric using the key cached in **T** [T] with $T = h_T(C)$. Afterwards, the client adds the non-key field I to an encrypted R with its identification. It also stores N in as the non-key field. The fields serve the key recovery as it will appear. Finally it sends out the record to server $h(C)$, as usual for an LH* file.

[5] The SDDS concept came from database community in early 90s. P2P ideas came later from network community. The initial addressing schemes used flooding. They were progressively reworked towards the so-called *structured* P2P, e.g., Chord, with more direct, basically DHT based addressing. The network community was basically not aware of the work of the former. A *posteriori* analysis has shown that both concepts are quite close, although some SDDS schemes could be seen as also or rather targeting the grid computing.

Insertions may optionally trigger the *cache scalability* and *secret scalability*. We address these facets of our scheme later on.

2.4. Record Search

To search record R with given key C, the client sends the query to bucket h (C). Provided the search is successful, the client retrieves key E cached in **T** [h_T (C)]. It decrypts R using E and delivers R to the application.

The LH* record search through a scan, exploring the non-key data of every stored data record, obviously does not make sense for LH*$_{\text{RE}}$. The LH* scan operation remains in use only for the key recovery and revocation, as discussed later on.

2.5. Record Update and Delete

The LH*$_{\text{RE}}$ record update involves obviously a search with decryption, unless the update is blind, and re-encryption, being otherwise carried as for LH*. The deletion is carried as for LH*.

2.6. Key Recovery

(Encryption) *key recovery* reconstructs one or more encryption keys from the shares, without *a priori* knowledge of share (primary) keys. Encryptor I may perform key recovery if for any reasons it lost a part of its **T**, perhaps because of memory corruption or hardware failure. Another need can be to decrypt a specific data record, identified by its primary key C, by an authorized client I' other than I. In addition, a given client I may become unavailable – and with it **T**, while another authorized client I' needs to continue to use the data, e.g., in a company or an archival system. The servers should trust or should be able to verify I received. This verification is not a part of our scheme, any well-known technique will do. Otherwise, a specific client of LH*$_{\text{RE}}$ called an Authority, identified as client A, with A trusted by every server, may also start the key recovery, handing it over to I' for termination. The recovery process performs the LH* scan operation.

We recall that LH* scan sends a query Q to all the currently existing buckets, using multicasting or unicasting. The latter may be handled in many ways; each assuring that all file servers get Q and only once, while some servers are perhaps unknown to the client. The scan termination handling the reply unicast messages can be probabilistic or deterministic. Only the latter is of interest here. A deterministic protocol guarantees that the client gets all replies it should. Specifically, to recover given encryption key E, lost in cell T in **T**, I or A issues scan Q with semi-deterministic termination requesting every share with I and T, to be sent back to the requestor or perhaps to I' on behalf of A. The server receiving Q verifies the identity of the requestor. Q brings the matching shares back to I or A, or to I'. The termination protocol counts the messages until (k+1). If the count is not reached after some timeout, the client considers some server(s) unavailable. It resends the scan using the (fully) deterministic LH* termination protocol. This protocol will localize the server(s) to recover. Currently, high or scalable high availability is not designed for LH*$_{\text{RE}}$, but it is already implemented in many different variants for LH* at large [LHS5]. The receiver recalculates E and finishes the recovery, by an update to **T** [T].

Similarly, to recover the key encrypting some record R with given primary key C, the requestor calculates $T = h_N$ (C), after perhaps a search of R not yet at the client, continuing with scan Q as above. The original client knows N, any other requestor uses N value saved in R during the encryption. Finally, to recover all the keys of I, I or A sends out the following scan Q' with semi deterministic termination. Q' requests for the sender or for I', every share with I and $T \leq N'$, with N' such that the flow received does not saturate the client. The successful scan termination occurs iff the client receives (k+1) N' messages. It may happen that $N' \geq N$ in which case Q' is the only scan. Otherwise the client continues with the further scans needed, after looping on key recovery for each T received. It progressively fills up **T**. The final dimension of **T** recovers N, hence it recovers h_T as well.

2.7. Key Revocation

Key revocation means here that for some good reason, the encryptor should no longer decrypt selected data records. Two cases that appear practical are (i) data record R of client I should no more be decryptable through its current encryption key E, and (ii) all the records of client I should no more be decryptable using the current keys. In both cases, the revocation should include the re-encryption of records with a new temporary or permanent key(s). In a company, a specific data record might suddenly need to become unreadable for employee I. Or, employee I was just fired. Or, the laptop with client node I got stolen... In LH*$_{\text{RE}}$ file, key revocation basically consists of (a) key recovery by A or of the new client I' on behalf of A, (b) temporary or permanent re-encryption of R using **T** of A or I'. Notice that case (i) may require A to re-encrypt R with some unique temporary key, to avoid that *I'* gains the side-effect capability to decrypt also other records encrypted by *I* through *E*. If the revocation concerns all the records of *I*, one also deletes all the shares of *I*.

2.8. Cache Scalability

As we mentioned, insertions may optionally trigger the *cache scalability*. Basic benefit is the generation of a small cache with a few keys only for a small file, scaling progressively with the file growth. The process may be further designed so that the client may specify the desired average number r of records per encryption key in the file. Cache scalability lay automatically maintains r, while expanding the cache. The actual number of records per key varies of course, being, e.g., under r initially, but should remains close to r. The feature seems of interest. Unlike a static cache, it keeps the balance between the assurance and disclosure size.

Cache scalability providing this function works for our variant as follows. Given the average load factor of an LH* file that is about 0.7, what is well-known, the current cache size N for the file of M buckets (servers) with the capacity of b data records per bucket, should be about $N = 0.7bM/r$. To start working, assuming that the file has $M = (k+1)$ servers initially, the client sets initial N accordingly and generates N encryption keys. It also uses N for any coming record manipulation. With new inserts, at one point an IAM comes. The client learns that the file has no more M buckets, as in the client image, but, at least, $M' > M$ buckets. The client adjusts the image as in general for LH* scheme. In addition, it scales the cache to $N' = 0.7bM'/r$ cells. Actually, it adds $N' - N$ cells to **T** and appends as many new keys. It finally assigns N to the current, i.e., performs $N := N'$. From now on, i.e., till next IAM, the client uses either the current of N for any encryption of a new record, or the N value found in the record for the decryption. The re-encryption in the case of an update may apply either one.

Notice that the above scheme considers a single process at the client. If there concurrent threads, as usual under current software architectures, one has to add some concurrency management. Consider for instance two IAMs processed concurrently. An exclusive locking of N seems sufficient, but likely, one can do better. We leave this analysis for the future.

2.9. Secret Scalability

An insert may also trigger optionally the secret scalability. The required secret size, i.e., k value, scales then as well, as we mentioned. The number of shares per each encoded key increases accordingly, enhancing the assurance. The investment may be wise for a scaling file. It is easy to see that as the number of file servers grows, the probability of k-node intrusion at least may grow as well, for any fixed k. Let us assume, for instance, some fixed probability of a node intrusion and independence among the servers as, e.g., typically on a P2P system.

To enable the secret scalability, the client sets up some values of M as successive thresholds for increasing k by one. For instance, the client may be poised to increase k by one any time the file doubles. The successive thresholds are then defined as $M_{i+1} = 2M_i$; with $i = 0,1,2….$ and M_0 denoting the initial file size, $M_0 \geq (k+1)$ as we spoke about. The client proceeds then as follows. It waits for an IAM informing that actual M reached or exceeded current M_i. It issues then the LH* scan query requesting shares with $T = 0,1…L - 1$, where L is the scan performance related increment, as N' used for the client keys recovery above. For each T value, the client retains one of the shares received; let us denote it as S_T. The client creates also a new noise N_{k+1}, stores it as a new share S_{k+1} and updates share S_T in the file to $S_T := S_T \oplus N_{k+1}$. It is easy to see that from now on, the secret for each key processed in this way is shared by $(k+1)$ shares. Once done with the current scan, the client requests next scan, for next N' elements of T, till it explores whole **T**.

Notice that the above scheme does not require the entire, possibly quite long obviously, operation to be atomic. The file remains accessible, while the secret size increased for some but yet not all keys. It seems to make sense then to introduce, for the behavioral analysis at least, the concept of the *file secret-sharing level* defined as the minimal k among all the records. Notice also that we ignore again at present the concurrency, this time however both at the client and at the servers. At a glance, at least, the update to share S_T and creation of S_{k+1} should be made atomic. Otherwise, e.g., a concurrent key recovery could end in error. Obviously, one needs some concurrency management with, e.g., 2PC protocol. We leave all these aspects for the future analysis as well.

3. Scheme Analysis

3.1. Assurance and Disclosure

Assurance is here broadly the confidence that an intrusion creating data disclosure will not happen. We discuss later how to measure the (amount of) disclosure if it happened anyhow. Under our assumptions, the intruder (attacker) has only two ways to proceed. The first one is to use brute force to decrypt the data records. Success mainly depends on the strength of the encryption key. Current schemes such as AES are likely not vulnerable to a successful crypt-attack. If the user chooses to have a key encrypts at most a few records only, the success of the hard work would be quite limited. As one increases the number of records encoded with the same key, the possibility of successful attack increases. Having fewer keys may nevertheless potentially benefit the assurance otherwise, as it will appear below. Modeling this trade-off is on our future goals list.

Alternatively, an attacker can collect all k+1 shares of an encryption key E for a given record R and also gain access to the bucket with R. Unlike previous variants, R might be collocated with one of the shares. The shares however are always on different servers, as long, as usual, the buckets may only split. This is because shares are initially placed on different servers and LH* splitting has the generic property of never placing two previously separated records in the same bucket (of course the opposite is not true). Likewise, if a record sent by a client gets forwarded by some bucket, a record sent by the client to another bucket cannot get forwarded through the same bucket. A sever cannot thus, even transitively get the knowledge of two shares of a key. Because of these (fundamental in the context) properties, an attacker will always have to break into at least (k+1) servers to access all shares. Notice that if we allowed buckets to merge, in a heavily shrinking LH*$_{RE}$ file, then in contrast two shares could end up collocated. Heavily shrinking files becoming rare, we omit the analysis of the case for now.

Even if an attacker is the administrator of a node and thus knows the location of a record or of a share, the attacker does not know the location of the other data needed to access the record. Since an SDDS file has usually at least dozens if not thousands of nodes, guessing the remaining k locations is hopeless. Notwithstanding that the shares move with the splits, with the 50 % probability on the average.

All things considered, the disclosure of a specific record in our LH*$_{RE}$ variant should be very difficult. The intruder (attacker) may however adopt the *spider attack* strategy. Namely, to break into any specific (k + 1) servers, to disclose whatever one could find there. The spider attack may be, for instance, appealing to an insider, in charge of some but not all servers of a "server farm". The evaluation of the probability of intrusion into such (k + 1) nodes is beyond the schema. In contrast, we can evaluate the conditional disclosure that such an intrusion may bring. One way is to estimate probability P of a successful disclosure of a record. This one is also the probability that R' and its shares are all at the intruded servers. P provides us further with another measure that is the expected number M of records that the intruder could disclose. M depends on the choice of k, on N and on bucket capacity. One can consider M value as a rationale for the choice of parameter k. Indeed, k should be the minimal value that achieves the expected disclosure size M, where M should be much smaller than one. Larger values of k would create an unnecessary burden of encoding and decoding. They might however be justified if the client considers breaking into more than k+1 servers feasible. The following example illustrates the point.

Consider an SDDS with 100 server nodes containing 1M data records, i.e., b = 10.000 per server on the average. Let it be k = 1 and N = 1. A successful attack (intrusion) breaks into any two servers. The probability that the two servers have both shares R' is $P = 1 / C^2_{100} \approx 1 / 5000$. Since N = 1, the intruder would be able to disclose all the records in these two buckets, i.e., M = 20 000 on the average. Choosing k = 2, lowers P to about 1/30 000. Likely, k = 2 should be the smallest value to choose for k. M remains however the same if the unthinkable occurs. If the same file had fewer servers, e.g., 10 servers only, $k \geq 4$ should rather be a decent minimum. Notice that this property also means that disclosure assurance scales well in our case, since it improves with a growing file.

Consider now that N = 2. In this case, the (conditional) probability P of key disclosure is about double. However, the disclosure size, in the number of disclosed records, is about half. Some users may consider this a better choice, especially for k = 2. Others may apply a larger k that allows them to choose a much larger N for P at least as small. In particular choosing N above (k +1)b reduces the disclosure size to approximately a single record, rivaling previously analyzed LH*$_{RE}$ variants that use a new key for every record.

Following the ideas in [MMJ3] one may formalize the above analysis, to create a design tool for choosing LH*$_{RE}$ file parameters. Such a tool might measure *assurance* more formally than in our current analysis, based on the probability that an intrusion does not disclose any data. Another concept implemented in such a tool could be that of conditional *disclosure* if an attacker intrudes into l servers. We can then estimate the probability of a key disclosure or the expected number of such keys given N and M… Of interest to us are only the disclosures occurring through the key decoding, i.e., the concepts one could tentatively qualify as *decoding* assurance and disclosure. We are working in this direction. Nevertheless, most importantly, it is already clear from our current analysis that both assurance and disclosure (variously measured) scale well.

3.2. Storage Occupancy and Access Performance

Storage for the encryption keys in the file and in the cache should be negligible. It is O ((k+2)N), where the cache accounts for O (N). For AES, e.g., the latter value should be about 32N bytes. With current RAM sizes, this allows for millions of keys. If an LH*$_{RE}$ designer wants the entire cache to fit in the L1 or L2 processor caches in order to achieve high performance, then there is still easily room for thousands of keys. The effective load factor of an LH*$_{RE}$ file, calculated with respect to the data records, should be in practice that of an LH* file with the same buckets and data records. To recall, it should thus be on the average ln 2 ≈ 0.7.

Messaging cost should be – as usual – the dominant factor in the access times for a record. It is proportional to the message count per operations. The costs for the cryptographic operations are proportional to the size of the data records, but should be negligible. Especially, since we use the symmetric encryption. The conjecture needs of course experimental verification.

The same message count costs of both data record insert and searches[6] are those of LH*.

Thus, our variant of LH*$_{RE}$ uses 1 message per insert and 2 messages per search. In case of an addressing error, this increases by 2 at worst. The update and delete message count cost is as for LH* as well. The combination of the negligible storage overhead with the absence of the encryption related messaging overhead for the data manipulation are advantages of this variant over the two others.

To recall, the basic scheme usually had an insert message count of (k+2) messages and a search cost of 2(k+2) messages. Both costs were thus (k+2) time greater. This does not mean that the response time were proportionally slower since the messages can be processed in parallel. The usual update cost is 2(k+2)+1 messages, hence k+1,5 more. The cost of a blind update (which does not notify the originator of the outcome of the operation) was 2k+3 messages, compared to usually one message. Finally, the usual cost of a normal delete is for both schemes that of a normal update. The cost of a blind delete is that of an insert.

The private share variant has the same search, and update message counts as the current one. Notice that the blind update does not make sense for the private share variant; it is cheaper to process it as the normal one. The insert and blind delete costs are in contrast, the basic one. The current variant remains thus several times faster accordingly. The normal delete operation for the private share variant takes advantage from its fast search. It remains nevertheless several times more costly than the current one.

The message count cost of cached keys creation is usually (k+1)N. A single key recovery given T should usually cost only (k+1) unicast messages replying to Q and one multicast or M unicast messages to send Q out towards *all* the M buckets of the file. In the unicast based send out case, it is the dominating cost factor. The client part of these M messages can however be as small as a single message. The client sends a key recovery request to bucket 0 that takes care of forwarding the request to all other servers. Key recovery, given a data record should add up usually two messages. The recovery cost of all keys of a client is $\lceil N / N' \rceil$ messages necessary to distribute the scan requests and (k+1)N unicast messages to bring in the shares. Finally, for the key revocation, one should add to the embedded key recovery cost usually two (unicast) messages per record to re-encrypt.

4. Conclusion and Further work

The LH*$_{RE}$ scheme with cached keys allows us to insert, search, update and delete data records without any encryption related messaging overhead. It has for these operations the same (messaging) performance as LH* itself. This should make this variant of LH*$_{RE}$ faster than the basic scheme for all data manipulations. In turn, the scheme needs storage for the cache, possibly RAM or on-chip processor cache, for best speed. This does not seem to be of much practical importance, as current on-chip caches can store thousands of keys and RAM can store millions. Another relative drawback can be a higher amount of disclosure if, and only if, one chooses to use relatively few keys.

Notice also that our variant presented here encourages higher values of k. A higher value increases indeed the assurance of the scheme at all times. In contrast, it only negatively matters possibly infrequently, when we recover or revoke a key. Unlike for any the data record manipulation under the basic scheme and inserts and deletes under the private-share variant.

As for the previous schemes, our analysis of this one is in early stage. We continue the following directions.

- Deeper analysis. Up to now, our main concern was the design of the algorithm. We now turn the focus more towards the trade-off between assurance and disclosure. The threat analysis should help user deciding on k value, and the encryption key space size to choose, as well as, perhaps, what encryption algorithm to use.

[6] Record searches are here only primary key-based searches, not searches for contents in the record's non-key field(s), obviously.

Next, while we have concentrated our performance analysis on the messaging costs, we should also investigate the encoding and encryption processing overhead and the overall timing one can expect in practice. Likewise, the performance analysis of the key recovery and key revocation has to be detailed. An experimental implementation seems the only practical way to solve the issues.

- Other variants. An important direction is to add up the concurrency to the cache and secret scalability management, as we mentioned. Also, we think about variants overcoming the limitations of our current assumptions, i.e., resistant to network snooping at the client, to a malicious action of the intruder etc.

An orthogonal direction is to incorporate the high-availability into the scheme, to avoid any losses of shares especially. We plan to reuse LH*$_{RS}$, [LMS5]. The way it uses the erasure correction makes the result somehow similar for the key storage to (m, k) secret sharing with $m < k$. Our rationale is also that LH* is inherently faster, especially through fewer hops, than a DHT, e.g., attempted for similar purpose in [MZ8].

Acknowledgments

Interactions with Ethan Miller and Darell Long, Storage Systems Research Center (SSRC) at UCSC, were instrumental to this work. We thank also Angelos Stavrou and Lei Zhang from the Center for Secure Information Systems (CSIS) at GMU for helpful suggestions after the 1st presentation of the scheme. Work by Witold Litwin was performed during sabbatical visits at SSRC and CSIS.

References

[B3] Bishop, M. Computer Security. Addisson-Wesley, 2003. ISBN 0-201-44099-7.
[C8] Cleversafe Opensource Community : Building Dispersed Storage Technology. http://www.cleversafe.org/
[CJHR5] Chua-Chin Wang, Jian-Ming Huang, Hon-Chen Cheng, Ron Hu, Switched-Current 3-Bit CMOS 4.0-MHz Wideband Random Signal Generator. IEEE J.Solid-State Circuits. V.40(1360-1365), June 2005.
[JLS8] Jajodia, S., Litwin, W., Schwarz, Th. LH*$_{RE}$: Scalable Distributed Data Structure with Recoverable Encryption. Work in Progress Report. GMU-CSIS & UPD-CERIA, (May 2008). *Submittted*.
[JLS8a] Jajodia, S., Litwin, W., Schwarz, Th. LH*$_{RE}$ with Private Shares : A Scalable Distributed Data Structure with Recoverable Encryption. Work in Progress Report. GMU-CSIS & UPD-CERIA, (May 2008).
[LMS5] Litwin, W. Moussa R, Schwarz T. LH*rs - A Highly Available Scalable Distributed Data Structure. ACM-TODS, Sept 2005.
[LNS96] Litwin, W, Neimat, M-A., Schneider, D. LH*: A Scalable Distributed Data Structure. ACM-TODS, (Dec. 1996).
[LSY7] Litwin, W. Yakoubin, H., Schwarz, Th. LH*$_{RS}^{P2P}$: A Scalable Distributed Data Structure for P2P Environment. Google TechTalk, June 19, 2007. Video at http://www.youtube.com/watch?v=bcTkFig6kyk[LSY8] Litwin, W. Yakoubin, H., Schwarz, Th. LH*$_{RS}^{P2P}$: A Scalable Distributed Data Structure for P2P Environment. NOTERE-08, June 2008.
[MMJ3] Mei, A., Mancini, L., V., Jajodia, S. Secure Dynamic Fragment and Replica Allocation in Large-Scale Distributed File Systems. IEEE TRANSACTIONS ON PARALLEL AND DISTRIBUTED SYSTEMS, VOL. 14, NO. 9, SEPT. 2003.
[S79] Adi Shamir: How to share a secret. Communications of the ACM, vol. 22(11), 1979
[S8] ENCRYPTION STRATEGIES : The Key to Controlling Data. A Sun Microsystems–Alliance Technology Group White Paper Jan. 2008
[SGMV7] Storer, M., W. Greenan, K., M., Miller, E., L., Voruganti, K. POTSHARDS: Secure Long-Term Storage Without Encryption. 2007 Annual USENIX Association Technical Conference.
[W8] Secret sharing. Wikipedia. http://en.wikipedia.org/wiki/Secret_sharing
[MZ8] Mills, B., N., Znati, T., F. SCAR - Scattering, Concealing and Recovering data within a DHT. 41st Annual Simulation Symposium, 2008.
[PGP4] Method and Apparatus for Reconstituting an Encryption Key Based on Multiple User Responses. PGP Corporation. U.S. Patent Number 6,662,299.