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

Sushil Jajodia
George Mason University
Fairfax, USA
Jajodia@GMU.edu

Witold Litwin
Université Paris Dauphine
Paris, France
Witold.Litwin@Dauphine.fr

Thomas Schwarz, S.J.
Universidad Católica de Uruguay
Montevideo, Uruguay
TSchwarz@CalProv.org

Abstract—LH*$_{RE}$ is a new Scalable Distributed Data Structure (SDDS) for hash files stored in a cloud. The client-side symmetric encryption protects the data against the server-side disclosure. The encryption key(s) at the client are backed up in the file. The client may recover/ revoke any keys lost or stolen from its node. A trusted official can also do it on behalf of the client or of an authority, e.g., to imperatively access the data of a client missing or disabled. In contrast, with high assurance, e.g., 99%, the attacker of the cloud should not usually disclose any data, even if the intrusion succeeds over dozens or possibly thousands of servers for a larger file. Storage and primary key-based access performance of LH*$_{RE}$ should be about those of the well-known LH* SDDS. Two messages should typically suffice for a key-based search and four in the worst case, with the application data load factor of 70%, regardless of the file scale up. These features are among most efficient for a hash SDDS. LH*$_{RE}$ should be attractive with respect to the competition.

Keywords-cloud; client-side encryption; key recovery, SDDS

I. INTRODUCTION

Many applications can benefit from fast, scalable and distributed storage that Scalable Distributed Data Structures (SDDS) offer. An SDDS receives data from client nodes and stores them at server nodes in a cloud, grid or P2P system. It adjusts the number of servers transparently for the application. The servers are usually outside of the control of the data owners, e.g., in a cloud. Many applications suited for SDDS need then to maintain confidentiality of the data on the servers. The cloud host may be usually trusted. It may not be the case however of insiders, having, e.g., routine maintaince access to some servers. Client-side encryption is then usually the most attractive goal. The downside is the difficulty of the (encryption) key maintenance. Loss, destruction, or disclosure can be disastrous. A third-party escrow service can safeguard keys and provide key recovery on request [1], [14]. These services are not widely used, perhaps because of legal and technical difficulties. Current industrial practices mainly advocate server-side symmetric encryption, unless "data security is paramount" [11, 3]. It is questionable how users feel about the resulting lack of control. Microsoft's Encrypted File System (EFS) uses in addition a public key infrastructure, [9]. The application encrypts the encryption key used for a file with the application's public key and adds it to the file header. The strategy is vulnerable to the application private key loss. To keep the files accessible even in such case, EFS may encrypt these keys with a domain (group) server public key and store them also with the files. The downside is that a successful intrusion into two servers may suffice to reveal all data at an EFS node.

The research prototype Potshards takes a different approach [12]. It targets data records stored much longer than the average lifespan of any encryption system. Participants use secret sharing, [10], to break every record into $K > 1$ shares stored at different systems, each protected by an *autonomous* authentication procedure. The price is a high storage overhead, as every share has the same size as the original record.

Instead of secret-sharing a data record for its secure storage, one may encrypt the record and secret-share only the encryption key, [2]. The keys are usually relatively negligible in size. The secret sharing storage overhead becomes negligible accordingly. We propose an SDDS called LH*$_{RE}$ for scalable files in clouds, grids based on this idea. LH*$_{RE}$ uses the client-side encryption, with one or many keys at the client. To prevent the key unavailability at the client or the key theft from the client, the keys are backed up in the file. The client or a trusted official may recover any keys. Likewise, one may revoke a key, rekeying the encrypted records.

The backup stores every key as $K > 1$ shares. Each share resides as a *share record* at a different server of the file extending over at least some $G \geq K$ servers. The file may scale up its extent at will in practice, as well scale down towards G. The share records may migrate while the file evolves, redistributing themselves about randomly over the current file extent. Despite the moves, the scheme guarantees that no two shares of a key end up at the same server, even transitively. The file is always k-safe, with $K = k+1$, i.e., no secret can be disclosed by an intrusion up to k-server wide. The value of k is file owner (creator) defined. Reasonable choices should be between 3 and 63. For larger intrusions, even orders of magnitude larger, into file with extent $N >> G$, LH*$_{RE}$ provides still a high assurance, e.g. at least 99%. The critical intrusion size is about proportional to N, with the ratio, that we call *resilience* (*level*) F of 31-93 % for the above assurance and k values, and for single encryption key used. In other words, e.g., for a file over 2K servers, even the intrusion 1.8K-server wide, should still leave the attacker with only 1% chance of success. Higher assurance for the same k lowers progressively F, which remains nevertheless very substantial. As our analysis shows, even for 0.99999 assurance, we have $F = 0.7$ for $k = 32$ and $F = 0.84$ for $k = 64$. Likewise, more encryption keys lower F a little as well. For instance, the F values similar to the above apply to 0.99 assurance and 1000 encryption keys. The benefit of more keys is the accordingly smaller disclosure, if an intrusion succeeds in disclosing some keys.

The access and storage performance of LH*$_{RE}$ should be in practice about those of LH*, as already hinted, [5, 6]. Below, Section II introduces LH*$_{RE}$. Section III analyses the assurance. Section IV discusses performance factors. Section V concludes the article. Space limitations make us referring sometimes the reader to the technical report [4] for complements.

II. LH*$_{RE}$ SCHEME

A. File Structure and Addressing

LH*$_{RE}$ reuses the file structure and addressing principles of LH* SDDS, as, e.g., in [7] and [5]. Some are modified towards the new goal. We discuss briefly these LH*$_{RE}$ features now. They are crucial for the LH*$_{RE}$ file safety, see Section 3.

An LH*$_{RE}$ file is a collection of records each with the primary key that we call here Record Identifier (RID). A users/application manipulates the file from the client (node). Records are in buckets of some capacity of $b >> 1$ records each. Every bucket is at a different server. Buckets have consecutive logical addresses 0, 1 ... N–1. Here, N is the *extent*, equals to the current number of buckets/servers of the file.

The file scales up through splits triggered by inserts, overflowing the buckets. Every split adds bucket N to the file, hence $N := N+1$ afterwards. If the file should (rarely) scale down, after many deletions, a *merge* reverses the latest splits. A specific *coordinator* component triggers splits and merges. A bucket reports to the coordinator overflows and underflows. The coordinator maintains the *file state* (n, i), where n is the *split pointer* since it points to next bucket to split and i is called *file level*. We always have $N = n + 2^i$ with $0 \leq n < 2^i$. The initial state, say ($\tilde{n}$, $\tilde{\imath}$), defines the minimal and initial extent G. We have $G \geq k+1$, where k is the file owner (creator client) defined as we spoke about. It is likely to be in practice in the range of $3 \leq k \leq 63$, with larger k making the file safer as we discuss in Section 3.2. Notice that if the file has several clients, as it should be usually the case, they all apply the owner's k.

The coordinator sends a message to bucket n requesting the split. It is usually not the bucket that has triggered the split, according to the well-known principle of the linear hashing (LH). The coordinator provides the physical address of the spare node to become the server of the new bucket. The determination of this server includes a specific precaution, we discuss in Section 3. Bucket n recalculates the address of each of its record as h_{i+1} (c), where h_{i+1} is an LH-function, e.g., as simple as $c \bmod 2^{i+1}$. Each split moves about half of the records to bucket $n+2^i$ that is N. After the split, the coordinator updates n to $n+1 \bmod 2^i$. If $n = 0$, then $i := i + 1$.

The major well-known result of the scheme is that the logical bucket address space is always contiguous, without need for any distributed hash table (DHT), otherwise mandatory, [15]. The *correct* address a, where RID c should locate its record is given by LH address calculus as follows:

(1) $a := h_i$ (c); if $h_i(c) < n$ then $a := h_{i+1}(c)$;

On the client side, every client maintains a private image of the file state, say (n', i'). It starts with ($\tilde{n}$, $\tilde{\imath}$), i.e., with the file having the G initial buckets 0,1..G-1. The client calculates the address of a key-based query (search, insert, update or delete) using (1) over the image. The image can be outdated. As any SDDS, the LH*$_{RE}$ indeed does not post splits and merges to the clients. The query can reach an *incorrect* bucket that is not the correct one. In presence of merges, and only then, the extent N' of client image can be $N' > N$ in which case the query can even reach a spare without any bucket. In this case, the client finds an error message returning from the server or no message after a timeout. It then resends the query using the initial ($\tilde{n}$, $\tilde{\imath}$) state, guaranteeing an address within file extent. Every bucket receiving a query can find through inherited LH* *test-and-forward* algorithm whether it is the correct one. If not, it forwards the query to another bucket which does the same. The algorithm is as follows. Here a is the address of the executing bucket and $j = i+1$ used for the latest split of the bucket, retained by each bucket and called *bucket level*:

```
(2) $a' := h_j(C)$ ; if $a' = a$ : exit ;
    $a'' := h_{j-1}(C)$ ; if $a'' > a$ : forward to $a''$ ; exit ;
    forward to $a'$ ; exit ;
```

It is known for LH*, hence is true for LH*$_{RE}$ as well, that in the absence of merges, algorithm (2) delivers the query to the correct bucket in at most two hops. In presence of merges, one more message may suffice.

The correct bucket receiving a forwarded message sends a specific *Image Adjustment Message* (IAM) to the client. The client processes an IAM as LH* does, hence we avoid discussing it here. A more accurate image results from, i.e., with N' closer to N. Same error cannot occur twice.

An LH*$_{RE}$ record consists of the RID and of the non-key fields at Figure 1. In some popular way, the *I*-field identifies the client or application that stored the record. The *T*- field (to be discussed in more detail) identifies the encryption key for recovery and revocation. The *F*-field is a flag indicating whether this is a data record or an encryption (*key*) *share* record we discuss soon. The *P*-field contains either the encrypted data record or the key share.

The LH*$_{RE}$ client manipulates the non-encrypted non-key fields reusing the LH* *scan* operation. A scan is like *Map-Reduce*. The client requests from every bucket all records fulfilling non-key criteria in the query (the Map phase). By multicast or forwarding the scan reaches every bucket and only once, including eventually buckets $N'...N$-1 beyond the client image. Buckets process the scan in parallel (the Reduce phase). They send results to the client only, (LH* scans may aggregate the results over intermediate servers). The rationale is the safety of the LH*$_{RE}$ file (Section 3).

B. Encryption Key Storage and Backup

LH*$_{RE}$ client encrypts every payload before sending it to the file as already mentioned. The encryption follows any strong symmetric encryption scheme. The client may use one or many (encryption) keys. It stores the keys locally,, in table **T**[0,..., t–1] with t being the number of keys in use. We call **T** *encryption key chain*. New keys are added to the end of **T**, dynamically perhaps.

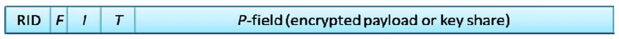

Figure 1: LH*$_{RE}$ Record Structure

The client also backs up each key in the file. The backup uses the secret splitting (e.g. [2, PHS03]). It breaks every key into K=k+1 (*key*) *shares*. Here k is the parameter chosen by the file owner. For every key C to back up, the client first generates k random strings of the same length as C. These become key shares $C_0, C_1 \ldots C_{k-1}$. The last share is $C_k = C_0 \oplus C_1 \ldots \oplus C_{k-1} \oplus C$. We have $C = C_0 \oplus \ldots \oplus C_k$. This calculus recovers C.

The client forms for each share a *share record*. For this purpose, the client initiates the F and I fields as already discussed. Next, the T field gets the index of C in **T**. The P-field gets the share. Finally, the client generates the RID for share. The RID defines through (1) the correct address of the record, where it will get stored. The set of the K RIDs fulfills two conditions. First, any two values are independent. (2) In the minimal file of G buckets, $G \geq K$ we recall, every RID provides a different correct address for its share. These requirements provide for the security of the scheme as it will appear.

C. Data Record Operations

An application inserting a record provides it to the LH*$_{RE}$ client in the non-encrypted form: (RID, D), where D are non-encrypted data. The client retains the RID, say r, sets F to indicate a data record, and fills in the identify information in the I-field. Next, it hashes r over t, e.g. calculates $l = r \bmod t$, puts l into T-field, reads **T**(l) and uses the key found to encrypt D into P-field. Finally, it sends the result to the bucket given by (1) for r over the client image.

Reversely, if the application searches the data record, say R, with RID r, the client searches the record with r in the file. If it gets it, then it explores the T-field, and gets the key from **T** accordingly. It then decrypts P into D field and delivers $R = (r, D)$ to the application.

We skip the processing of a delete and of an update as obvious. Notice however that if keys were added to **T** since the record was last written, the update may end up using a new encryption key.

D. Encryption Key Recovery and Revocation

The client or the authority requests these operations. The key recovery restores the keys from their backup when the original **T** is unavailable. The client issues the scan over I-, T-, and F-fields. The authority does the same, but also specifies the client to receive the result. Each server searches all the records with given I, and with F indicating that these are key shares. It send them all (directly) to the specified client. The client groups the shares received by T-field. Each group has K records, obviously. The XOR operation over P fields in each group recovers every key. The recovered keys are reinserted into new **T**.

The client or the authority may alternatively *revoke* some keys. The need can be caused by a theft of the client. Alternatively, a terminated employee should not retain the data access capabilities. The key(s) to revoke are first retrieved from **T** or recovered from the file. An obvious scan recovers then all data records encrypted with each key. The client decrypts each P-field, re-encrypts the P-field with a newly created key and re-inserts the record into the file. See [4] for details skipped here.

III. SECURITY ANALYSIS

A. Threat Model

LH*$_{RE}$ stores the records in the cloud; we disregard (here) attacks on clients, as outside the cloud. We focus on attacks on the servers. The main threat should come from insiders, with access to some server(s), usually at most k or to a number lesser than the file extent. The attacks are on individual servers (buckets), one by one. Larger attacks are less and less likely. In particular,, no intruder is able to break into all the servers of the file in the cloud. Next, we assume the industrial strength encryption. The only practical way for the attacker is then to break some secret, recovering a key from its shares. Having the key, the attacker attempts to disclose the records it encrypts.

Furthermore, we consider that the intruder only tries to read the data. S/he does not seek to maliciously modify the code. We finally disregard the network snooping. In [4], we discuss LH*$_{RE}$ variants for relaxed models, e.g. with the snooping.

B. Key Safety

LH*$_{RE}$ is k-safe. We recall that it means that, regardless of the file expansion from G, an intruder has to break then into more than k servers to find all shares of some key. The intruder can capture a share by either finding it in the bucket of the intruded server or by obtaining it in transit. The latter happens when a client creates a key and the share record gets forwarded. The scan delivering the share records directly to the client does not create this problem.

To prove k-safety, it suffices to show that no expansion could lead to two shares of a secret at the same address, correct one or during the forwarding. The formal proof is in [4], because of space limitations. Here we provide an informal presentation. The buckets created by splits of bucket a are *descendents* of a. Bucket a is their ancestor. Recursively, descendants of descendants etc. are also (indirect) descendents. For every bucket a in the minimal extent G, bucket a and the descendants form the *descendent set* D_a. Each D_a is ordered by the (unique) order of creation of its elements, defined by the split pointer n trips. In fact, it is tree structured with levels defined by the successive values of i, while the file scales up. According the LH splitting principles in Section 2, we have for any a, $D_a = \{a, a+2^i, a+2^{i+1}, a+2^i+2^{i+1}\ldots\}$, with i being the file level for the state (a, i), when bucket a splits for the first time. Observe from Section 2 that i is the minimal integer so that $a < 2^i$.

Example: Let it be $k = 3$ and $G = 4$. The four ancestors are buckets 0, 1, 2, and 3. We have $D_0 = \{0, 4, 8, 12, 16...\}$, $D_1=\{1, 5, 9, 13, 17...\}$, $D_2=\{2, 6, 10, 14, ...\}$, and $D_3=\{3, 7, 11, 15, ...\}$. Consider now that we choose $G = 6$, so to make the life of an intruder little harder, since the four shares, even initially, are randomly somewhere between six buckets. We now have $D_0 = \{0, 8, 16,...\}$, $D_1=\{1, 9, 17, ...\}$, D_2 and D_3 as before, but also $D_4 = \{4, 12\ldots\}$ and $D_5 = \{5, 13\ldots\}$.

A share record may potentially move from a splitting ancestor to any of its direct descendants. Then, it may move to any indirect one, following the above discussed order and resulting paths in the tree. These are the only possibilities for the moves, regardless of how far N could scale up from G. In other words, D_a is the correct address subspace for address a. The LH*$_{\text{RE}}$ splitting principles in Section 2 imply on the other hand that these subspaces partition the entire address space {0, 1, 2…}. As we stated finally, the client chooses RIDs of the share records of a key so that they are all in different buckets among 0…G-1. Hence, regardless of N, the splits can never move two of them into the same bucket.

An attacker breaks however not directly into buckets but into the servers. Hence, any server should get during its life for the file only one bucket or, at most, buckets within the same D_a. Otherwise the safety could obviously get destroyed during merges disallocating the servers followed by splits reallocating them. The discussed precaution is (easily) enforced by the coordinator, as we said in Section 2.

Finally, we did not elaborate in Section 2 on the test-and-forward algorithm (2), because of the space limits. We prove in [4], but one may see from (2) rather easily, that the one or two hops at most it creates, also follow only addresses in the D_a with a being the correct address of the RID in the query in the file extending over G buckets only. Hence, for any key share record, any client image and any $N > G$, the eventual forwarding cannot traverse or reach the bucket and server with some other key share record of the key already there, or having transited through. No key or data operation can thus violate the k-safety of LH*$_{\text{RE}}$ file.

C. Assurance

LH*$_{\text{RE}}$ file safety concept gives a simple measure of confidence that an intruder cannot read or write any data records. It gives the lower bound $K = k+1$ on the number of intrusions (intrusion extent) necessary for any success. We show now that, in practice, even a much higher extent usually still gives only a little chance to the intruder of disclosing any secretes. For the proof, we reuse the popular concept of *assurance*, [8]. We formalize it here as the probability that an intrusion into x out of N buckets does not disclose any keys. Assurance depends on x, on N, on t and on k. We first aim at the average number of keys obtained by the intruder. We calculate it first for the use of single key, then - of multiple ones. Afterwards, we analyze the *disclosure* that we define as the expected amount of records decrypted by the intruder.

1) *Single Key:* We first calculate the assurance in a file with a single key. We use our basic threat model; hence the attacker does not know which bucket is located where. Otherwise, an attacker that has found a key share record in a bucket with number in D_i no longer needs to look for this key share in buckets with numbers also in D_i. We assume that the intruder has gained access to x out of N buckets. We know that $K= k+1$, of these buckets contain a key share record and need to calculate the probability that all of these K buckets are among the x accessed buckets. We determine the probability by counting. There are then $\binom{N}{x}$ ways to select the x buckets that the intruder broke into. If the attacker has intruded into all K sites with key shares then there are $\binom{N-K}{x-K}$ ways to select the remaining $x - K$ buckets intruded into but without key share. Thus, the probability that the intruder obtains a given set of K key shares with x intrusions is

$$p_1(N, x, K) = \binom{N-K}{x-K}\binom{N}{x}^{-1}.$$

The assurance against disclosure of a single key is $q_1(N,x,K) = 1 - p_1(N,x,K)$.

Figure 2 plots q for a file with 32, 64, 128, 256, 512, and 1024 server sites. We calculate assurance for four (top) and eight key shares. Since we would often be given a required assurance (expressed in number of nines), we drew the x-axis at the 99.999% (five nines) level. Even for moderately large files the required number of intrusions has to be much larger than k+1. We call the maximum number of sites whose intrusion does not let assurance fall under a given level the *critical number* x_0. For example, a file with extent N = 512 and k = 7 has critical number 289 for two and 124 for five nines assurance. The ratio $F = N/x_0$ is almost constant, as can be seen from the almost even spacing on the logarithmic x-axis. For larger numbers of key shares, we give the critical point x_0 of attacked sites up to which the assurance remains higher than a certain value (Table 1). We give values for K = 32 and K = 64 key shares, with N ranging from 512 to 8192.

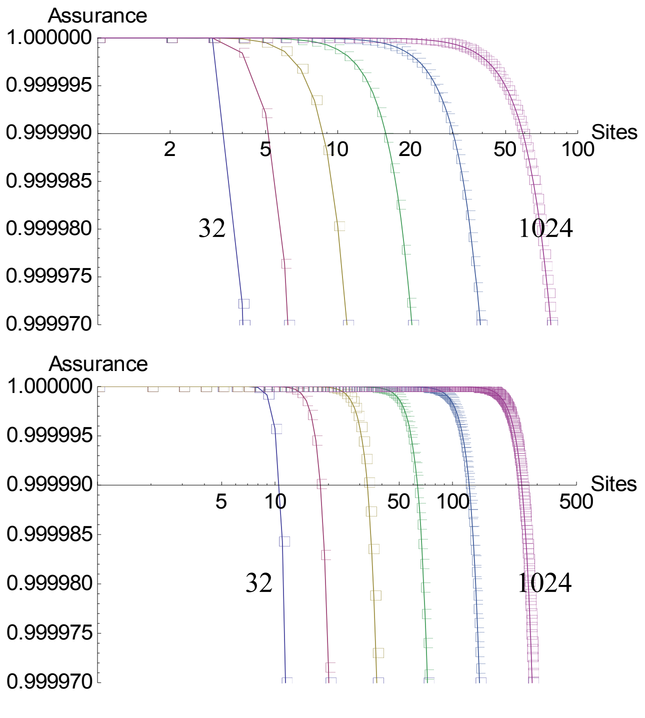

Figure 2: Assurance in an LH*$_{RE}$ file with 4, 8 key shares (top, bottom), hence k = 3, 7, extending over 32, 64, 128, 256, 512, and 1024 servers. The x-axis gives the number of intruded servers.

	K=32		*K*=64	
N	x_0 2 nines	x_0 5 nines	x_0 2 nines	x_0 5 nines
256	223	183	240	219
512	445	362	478	433
1024	888	719	955	860
2048	1775	1433	1908	1716
4096	3549	2863	3813	3426
8192	7096	5721	7625	6848

Table 1: Critical number of intruded sites (out of N) with assurance falling below the two nines / five nines level if more sites are intruded.

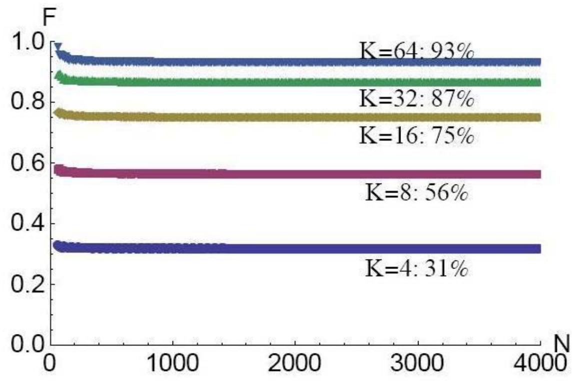

Figure 3: Values of F for K=4, 8,16,32,64 and two nines.

Figure 3 plots further the ratio $F = x_0/N$ found through simulations for various k's and up to rather large N = 4K. As one sees, the ratio appears surprisingly constant, thought in fact it falls slightly when N grows. The formal rationale for this nice behavior is under investigation. We call F the *resilience* (level) of the file and we say that the file is F-resilient. E.g., it is 0.93-resilient, regardless of N, for the two nines assurance and K = 64 and at least 0.31-resilient already for K = 4, at Figure 3. In both cases, the single-key LH*$_{RE}$ file should thus in practice easily resist to intrusions orders of magnitude larger than k, as we have promised. For instance, if the user wishes the file to be 63-safe, (i) s/he creates it with $G \geq 64$ buckets and single encryption key, then (ii) e.g., if the file scales up to 4K buckets, the intruder has at most 1% likelihood to get any keys, as long as s/he did not managed to break into more than 0.93*4K = 3720 buckets (Figure 3). Clearly an exploit for a well managed cloud. It is also clear that if the file resilience is the primary concern, it also pays to choose a smaller bucket capacity b, increasing the file extent N.

2) Multiple Keys: The assurance against retrieval of one key out of r keys is $q_r = (1-p_1(N,x,K))^r$ reflecting the lowering of assurance when using multiple keys. Figure 4 and Figure 5 plot q_r for r = 10 and r = 100, respectively, with a two nines threshold. In comparison with Figure 2 they show the effect of more keys. Nevertheless, the threshold of successful attacks is still very high. For instance, with a single key, K = 4, and N = 64, a value of x = 21 still does not press assurance below the 99% mark. For r = 10, this value decreases to about 13, and for r = 100, to 8. This is still more than twice the safety rating of k = 3. Of course, while the chance of a breech increases with r, the extent of the breach diminishes as most successful attacks will only yield a single key given a high ratio of N/k. We also calculated assurance for higher values of K and give the results for K=32 and two nines in Table 2. We notice that for N large enough the ratio of critical point to N remains almost constant, though slightly falling. For K =64 and two nines, these ratios are between 0.919 and 0.898 for r= 10, and between 0.882 and 0.867 for r=100.

The curves show that increasing r with N is secure. For example, Figure 2, for $k = 8$ and $r = 1$, the assurance for $N = 32$ remains above our threshold of 0.99 for ≤ 20-bucket wide intrusion. For the same x, Figure 4 shows that if the file reaches the extent of about $N = 48$, we can increase r to 10. The assurance remains above the threshold, while the expected gain for a successful intruder decreases by ten. Similarly, Figure 5 shows that if N increases to about 55, we can increase the number of keys used to 100. The quantity of records becoming accessible to the attacker shrinks again by 10.The curves illustrate also that larger values of k compensate for larger r. For instance, Figure 4, in the case of $r = 10$ and $N = 64$, four shares put the assurance above the threshold for a 12 bucket intrusion, while $k = 8$ shares allow a 29 bucket intrusion, an increase of x by about 2.5.

We also calculated assurance for $K = 32$ and $K = 64$ and observe, Table 2, very high assurance and only very slightly varying resilience level F for each r. F decreases for greater r, remaining nevertheless impressively high: over 70% for $K = 32$ and 84% for $K = 32$, for our largest $r = 1000$ keys.

D. Disclosure size

Disclosure d measures the quantity of data revealed by a successful intrusion. More precisely, we define d to be the expected proportion of records revealed by an intrusion into x servers. As we will see, d does not depend either on the distribution of data records to buckets nor on the distribution of records encrypted with a certain key. The attacker has intruded into x servers each with a bucket, has harvested all key share records, and is now in possession of any encryption key for which s/he has gathered all key shares. S/he possesses now a given key with probability:

$$1 - q_1 = \binom{N-K}{x-K}\binom{N}{x}^{-1}$$

Since on average, s/he has obtained a proportion x/N of all data records encrypted with this key and since s/he needs encryption key and data record to obtain access to an application record, the attacker obtains on average the following proportion of all application records encrypted with this key:

$$d(N,x,k) = \binom{N-K}{x-K}\binom{N}{x}^{-1}\frac{x}{N}$$

Since this is a proportion, the same expression not only gives the disclosure for a single key but also the disclosure for a number of encryption keys. In particular,

K=32	r=10	r=100	r=1000
N=256	209	195	183
N=512	415	387	362
N=1024	828	771	719
N=2048	1653	1539	1434
N=4096	3304	3075	2863
N=8192	6605	6147	5722
Range	0.82–0.81	0.76–0.75	0.71–0.70

K=64	r=10	r=100	r=1000
N=256	233	226	219
N=512	462	447	433
N=1024	922	891	860
N=2048	1841	1777	1716
N=4096	3680	3551	3427
N=8192	7357	7098	6849
Range	0.91-0.90	0.88-0.87	0.86-0.84

Table 2: Critical values for K = 32 and 64, two nines, and number of keys r from 10 to 1000.

disclosure does not depend on the number of encryption keys used. We also calculate *conditional disclosure*, defined to be the disclosure (measured again as a proportion of accessible application records over total application records) given that x intrusions resulted in a successful attack, i.e. one where the attacker has obtained at least one key and therefore one or more application records. The probability p for this successful attack is $p = \binom{N-K}{x-K}\binom{N}{K}^{-1}$. We set $q = 1 - p$. The probability of obtaining at least one out of r keys is $P = 1 - q^r$. The conditional probability of obtaining exactly s out of r keys given that we obtain at least one key is $\binom{r}{s}p^s q^{r-s} \cdot P^{-1}$ and the expected number of total keys obtained given that we obtained at least one is $\sum_{s=1}^{r} s \cdot \binom{r}{s}p^s q^{r-s} \cdot P^{-1}$ which after factoring out P^{-1} is the value of the expected value in a binomial distribution and evaluates to *rp/P*. The expected number of records encrypted with s out of r keys is *s/r*.

The expected proportion of LH*$_{\text{RS}}$ data records obtained with x intrusions into N buckets is x/N (even though LH* buckets are not of the same size). The conditional disclosure is given as the expected number of keys disclosed (given that this is a successful intrusion) divided by r and multiplied with the expected number of data records obtained, hence is

$$\frac{xp}{NP} = \frac{xp}{N(1-(1-p)^x)} .$$

Figure 6 plots the resulting (conditional) disclosure d in two contour plots. Each plots d as a function of N and r, for $k = 7$ with $x = 8$ for the left plot and $x = 32$ for the right one. The left plot shows thus the minimal intrusion able to disclose any data. The right one shows, as an example, a four times wider one. Each shaded region is for d equal to or under the value on it at the figure. These range from 0.0001 to 0.0006. As an application of the plots, consider for instance that the user creates indeed a

7-safe file, expected to scale over $N=100$ buckets. The left plot shows that choosing $r=1000$ keys results in $d < 0.0001$. The lucky attacker gets thus access to less than 0.01% of application data. Instead of 8/100 = 8% for a single key. The right plot shows that even the four times wider intrusion still discloses only $d \leq 0.03\%$ of data, instead of up to 32%. Notice the thumb rule matching the intuition: d for r keys is about r times smaller than for a single key. If the file still doubles, reaching $N = 200$, the dashed line shows that d decreases to again to at most 0.01%. As the analysis has shown, the price to pay is a moderately smaller resilience and a larger **T** of course. The d values clearly illustrate nevertheless the potential interest of multiple encryption keys for some users.

4) Refinements of Intrusion Scenario: Our basic intrusion scenario limits the capability of the attacker perhaps sometimes more than s/he will be able to do. By analyzing the data from intruded servers, a savvy attacker can reconstruct the location of at least some buckets. The attacker may then prefer intruding the servers not yet split in the current round. These contain indeed twice as many key shares, by properties inherited from LH*. The attacker may also remove from the target list the servers from the descendent set with an already obtained key share. Such scenarios can be modeled as interesting optimization problems for the attacker, but their analysis requires advance knowledge of LH*. We therefore discuss it in [4]. It turns out that access to discussed information, while useful, does not change much the assurance and resilience figures. We give some thoughts therefore also to even further refinements of the intrusions. For instance, an attacker could triage servers based on the results of vulnerability scan. Likewise, the costs of an attack to a site could depend on the site, etc.

IV. PERFORMANCE

LH* has been implemented and measured [7, 5]. The result apply to the inherited features of LH*$_{RE}$. Especially, since the LH*$_{RE}$ client manages encryption with the storage and processing costs of key shares that should be usually negligible. RID-based operations such as look-ups and inserts of records should have then service times dominated by messaging and the difference to LH* should not be noticeable. It should be the same for the load factor, being thus usually about 70%.

With respect to LH*$_{RE}$ exclusive features, the key recovery time is dominated by the LH* inherited scan time. This one is $O(2N)$ messages. The exact value depends on the number of buckets beyond the client image, in rather complex way. Key revocation time is the longest, being dominated by the messaging for key recovery, then for the retrieval and reinsertion of re-encrypted records. It includes thus at first two rounds of scans with their above complexity. The first is the scan for the key shares. Follows the scan for the records encrypted with the recovered keys. Finally, there are $O(M)$ messages for the re-insert of M re-encrypted records.

V. CONCLUSION

LH*$_{RE}$ provides the client side encryption for scalable distributed hash files in the clouds, with the minimum encryption related overhead. Its novelty consists especially in the secure backup of the client's encryption keys within the cloud. The file should typically resist to intrusions, fully or with high assurance against even very extensive attacks. The disclosure in the bad case may be highly limited. The scheme appears an attractive choice to the current competitors.

Future work will aim on experiments. Also, we will analyze attractive developments, we hint to in [4]. For instance, one may dynamically scale up the secret size. One should be able on the other hand to add the high-availability, protecting against share and data loss, e.g., as in [5]. There are also ways to generalize the threat model. Next, it may be useful to re-encrypt the shares. Finally, one may analyze the portage to other popular SDDSs, especially the DHT based.

ACKNOWLEDGMENT

The work of Sushil Jajodia was partially supported by the NSF grant CT-20013A. Discussions with Ethan Miller & Darrell Long about Potshards & other SSRC projects during the 2007 sabbatical quarter of W. Litwin at UCSC-SSRC triggered the interest to the subject. Comments by P. Valduriez and his ATLAS team to the presentation by W. Litwin at U. of Nantes helped the early stage of this work.

REFERENCES

[1] M. Bishop: Computer Security. Addison-Wesley, 2003. ISBN 0-201-44099-7.
[2] M. Bellare, R. Canettiy, H. Krawczyk: "Keying hash functions for message authentication". Advances in Cryptology: Crypto 96. Lect. Notes in Computer Science v. 1109, 1996.
[3] Cleversafe Opensource Comm.: Building Dispersed Storage Technology. http://www.cleversafe.org/
[4] S. Jajodia, W. Litwin, T. Schwarz, LH*RE: Tech. Rep. 2010.
[5] W. Litwin, R. Moussa, T. Schwarz: "LH*rs - A highly available scalable distributed data structure. ACM-TODS, Oct. 2005.
[6] W. Litwin, M-A. Neimat, D. Schneider: "RP*: A family of order-preserving scalable distributed data structures. VLDB, 1994.
[7] W. Litwin, M-A. Neimat, D. Schneider: LH*: A Scalable Distributed Data Structure. ACM-TODS, (Dec. 1996).
[8] A. Mei, L. Mancini, S. Jajodia: "Secure dynamic fragment and replica allocation in large-scale distributed file systems. IEEE Tr. Parallel and Distr. Systems, v. 14(9), 2003.
[9] Microsoft TechNet. Using Encrypting File System.

http://technet.microsoft.com/enus/library/bb457116.aspx#EIAA

[10] A. Shamir: "How to share a secret." CACM, 11, 1979

[11] "Encryption strategies: The key to controlling cata". A Sun Microsystems–Alliance Technology Group White Paper Jan. 2008

[12] M. Storer, K. Greenan, E. Miller, K. Voruganti: "POTSHARDS: Secure long-term storage without encryption. Annual USENIX Ass. Tech. Conf., 2007.

[14] Publication: "Method and apparatus for reconstituting an encryption key based on multiple user responses". PGP Corporation. U.S. Patent Number 6,662,299.

[15] I. Stoica, R. Morris, D. Liben-Nowell, D. Karger, M. Kaashoek, F. Dabek, H. Balakrishnan: "Chord: A scalable peer-to-peer lookup service for internet applications. SIGCOMM'01.

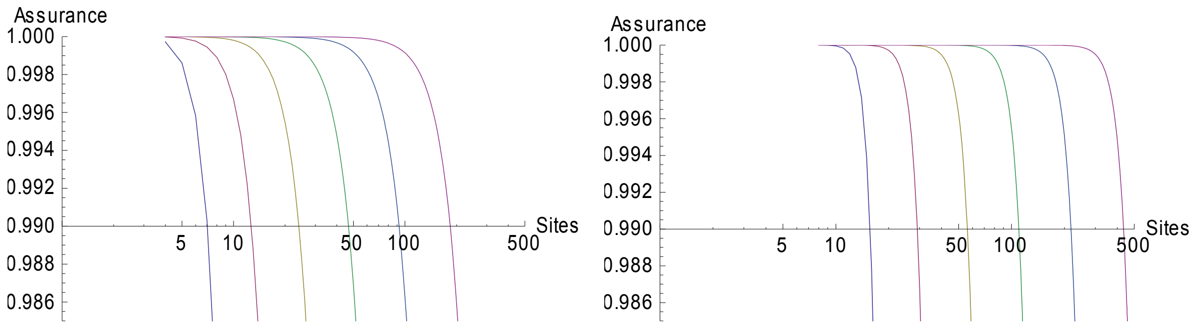

Figure 4: Assurance in LH*$_{RE}$ file with 10 keys and 4 and 8 (left, right graph) key shares, hence k = 3,7 respectively, extending over 32 (left curve), 64, 128, 256, 512, 1024 (right curve) servers. The x-axis is the intrusion size.

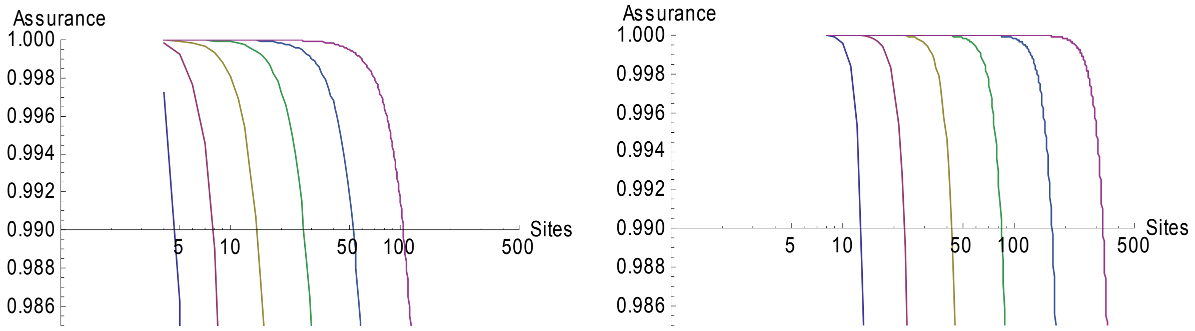

Figure 5: Assurance in LH*$_{RE}$ file with 100 keys and 4 and 8 (left, right graph) key shares, i.e., k = 3, 7 respectively, extending over 32 (left curve), 64, 128, 256, 512, 1024 (right curve) servers. The x-axis is the intrusion size.

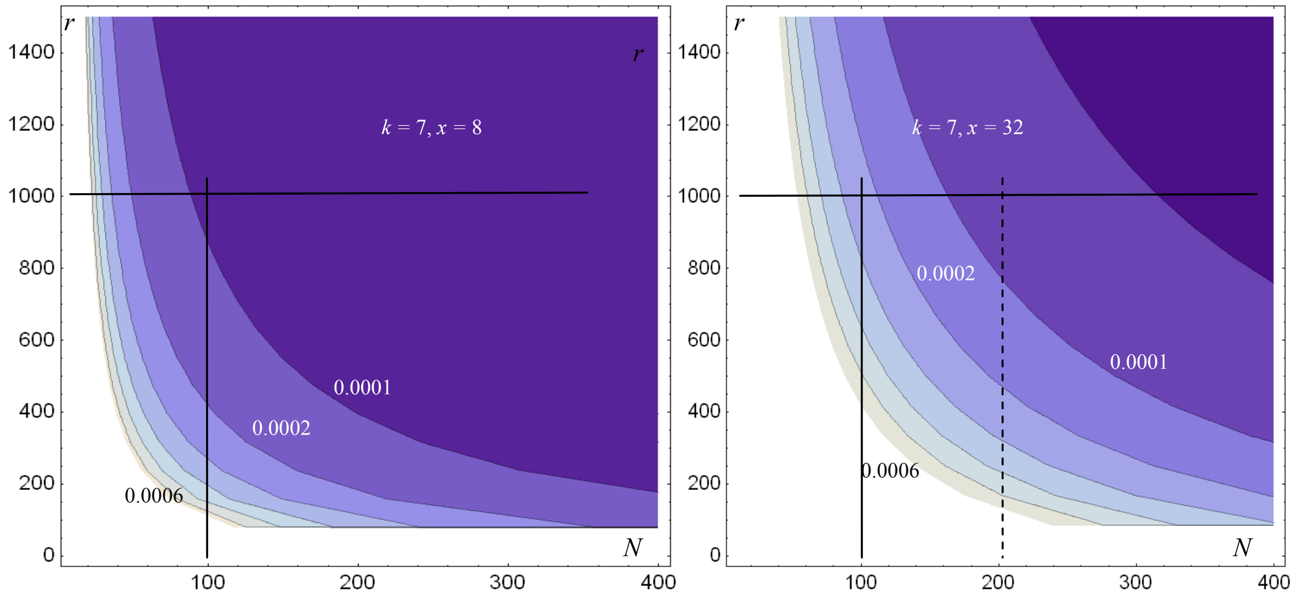

Figure 6: Disclosure for file with k = 7, under 8 and 32 intrusions. We vary N from 20 to 400 and r from 1 to 1500.