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

Sushil Jajodia
Center for Secure Information Systems (CSIS)
George Mason University
Fairfax, VA 22030-4444,
Jajodia@GMU.edu

Witold Litwin
Université Paris Dauphine,
Pl. du Mal. de Lattre,
75016 Paris, France,
Witold.Litwin@Dauphine.fr

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

ABSTRACT

LH*$_{RE}$ is a Scalable, Distributed Data Structure (SDDS) based on LH*. It is designed to store client data in the potentially hostile environment of cloud, grid, or even P2P computing and protects data confidentiality by client side encryption. Its novel feature is safe backup of the encryption keys used within the LH*$_{RE}$-data structure itself. This allows administrators to recover and revoke keys using normal SDDS operations. All keys used are broken into *k*+1 shares (where *k* is a freely chosen parameter) and the shares are distributed among the servers such that LH*$_{RE}$ remains *k*-safe, i.e. such that an attacker has to successfully subvert more than *k* servers in order to obtain a key and be able to read client data. Even for intrusions several times wider, assurance (the probability of obtaining a key and hence gaining the ability to read data) is typically high and disclosure (the amount of data readable) after a successful attack is typically low. Storage costs and performance of LH*$_{RE}$ are about the same as for LH*. Overall, LH*$_{RE}$ combines client side encryption with safe key management and thus an attractive alternative to third party escrow or server-side encryption.

General Terms

Algorithms, Reliability, Privacy

Keywords

Linear Hashing, LH*, Scalable Distributed Data Structures, Encryption, Database as a Service.

1. INTRODUCTION

Many applications can benefit from the fast, scalable and reliable storage in distributed main memory or distributed storage devices that Scalable Distributed Data Structures (SDDS) can offer. An SDDS stores data records coming from client nodes on a number of server nodes. The number of servers adjusts gracefully to the size of the data. SDDS schemes are intended for P2P, grids and clouds. The latter become very important as seen by the emergence of Windows Azure, Simple Storage by Amazon, GoogleApps, etc. The reference count for SDDS (15000 as of June 15, 2008 on Google) shows that SDDSs enjoy significant interest. Many potential SDDS applications need to maintain data confidentiality, as they are stored in a potentially hostile environment. No unauthorized administrator of a participating system and especially no attacker to a system storing SDDS data should be able to read the data. Examples of such applications include data in companies and organizations, health related data, remote backup services in the cloud, grids, archive systems, P2P social networks, etc.

For any such application, client-side encryption using a secret key is a very attractive choice for protecting the confidentiality of its data. The down-side is key maintenance. The key is needed for the lifespan of the data and its loss, destruction, or disclosure can be disastrous. A third-party escrow service can safeguard keys and provide key recovery on request [B03], [PGP04]. These services are not widely used, perhaps because of legal and technical difficulties. Current industrial practices use server-side encryption, unless "data security is paramount", [S08], [C08] and EFS, [MTN09]. It is questionable how users feel about the resulting lack of control. With EFS, an application encrypts the encryption key with the application's public key and stores the result in the header of the encrypted data record (a "file" in EFS terminology). EFS thus requires public key infrastructure and files become unrecoverable once the corresponding private key is lost. To protect against this scenario, the encryption key can be additionally be encrypted with a domain (group) server public key that is stored as another header field in the file. The domain private key is stored on some server. A successful intrusion to only two servers reveals all records in the system.

The research prototype Potshards takes a different approach [SGMV07]. It targets data records stored much longer than the average lifespan on any encryption system. It achieves data confidentiality by secret sharing. Participants break every record into several shares stored at different systems. The price tag is a high storage overhead, as every share has the same size as the original record.

Database-as-a-service has become a popular paradigm where an encrypted client's database is stored at an external service provider. A first solution of this problem was presented in [HMI02, HILM02, HIM03]. Different measures for evaluating the robustness of indexing techniques against inference attacks have been studied in [D&al05]. In [A&al06,A&al08], the authors address the problem of access control enforcement, proposing solutions based on selective encryption techniques for incorporating the access control policy on data.

In what follows, we describe an SDDS called LH*$_{RE}$ that provides client-side data confidentiality. It is a variant of well-known LH*, the distributed version of Linear Hashing [LMS05, LNS94, LSY07, LSY08]. A Record Identifier (RID), a. k. a. a primary key, identifies each record. As the number of records increases or decreases, the resulting LH*$_{RE}$ stretches over more or less servers of the file buckets (one per server). This adjustment proceeds by bucket splits and merges. RID-based operations – retrieving, inserting, deleting or updating records – have messaging times independent of the file size. Typically, a client locates a record going directly to the storing server. At worst, there are two more hops among the servers, but only one additional hop is necessary in a P2P environment [LSY08]. These properties are unique to LH*.

The LH*$_{RE}$ client encrypts every record before it inserts the record at a server. The application or the client chooses the encryption key. Any strong symmetric encryption method such as AES will do. The client can use one, several, or many encryption keys. It keeps each key, but also stores each key in the LH*$_{RE}$ file itself using secret splitting e.g. [BCK96]. Each key is broken into $K = k + 1$ shares. Here k is a file parameter, called the *file safety level*. The choice of k offers a trade-off between the strength of data protection and the costs of key recovery and replacement. We use a basic secret sharing algorithm [PHS03] to generate shares of keys and store each of them in a record, called a *share record*. Using secret sharing on keys instead of records (á la Potshards) incurs usually negligible storage overhead.

LH*$_{RE}$ guarantees that as long as the file has at least K servers, two key share records (of the same key) never reside or even pass through the same server, however the file changes through bucket splits and merges. An attack at a server has to break the record encryption (which we assume to be impossible) or has to recover the encryption key from all K shares. We call the file k-safe as it can withstand k intrusions. In a file stretching over many servers, the number of server intrusions needed by the attacker to gather all k shares of a particular key is of course much higher (Section 4). In contrast, a legitimate client or an authorized administrator can recover all keys through routine LH*$_{RE}$ operations. They can similarly revoke any key by rekeying all records encrypted with that key. The client can also adjust the number of keys with the file size. This can help control the damage resulting from a potential large scale intrusion. The properties of our scheme offer an attractive alternative to current practices.

The next section describes the basic LH*$_{RE}$ algorithmic. Section 3 analyzes security. We define our thread model, prove k-safety under this model, analyze assurance and determine the likely disclosure in the case of a successful attack. Section 4 discusses briefly some variants of the basic scheme. These add capabilities or target other SDDSs than LH* as the basis. We conclude in Section 5. Space limitations force us to assume that the reader has some familiarity with SDDS and LH* in particular. Good references are [LNS96], [LMS05], [LSY07]. Appendix A recalls the properties of LH* that we need.

2. The LH*$_{RE}$ Scheme

2.1 File Structure and Addressing

With respect to its internal structure and its manipulation by an application, an LH*$_{RE}$ file is a normal LH* file [LNS94]. First, any LH*$_{RE}$ file is a collection of records. *Client nodes* (*clients* for short) store records in an LH*$_{RE}$ file on a behalf of one or more, possibly distributed applications. The records are stored in buckets (with logical addresses 0, 1 ... N – 1), each of which resides at a different *server node* (*server* for short). N is called the *extent* of the file. Clients in LH*$_{RE}$ have the additional role of encrypting and decrypting records. They manage the encryption keys autonomously as we will see below. When a file is created, the creator specifies the file safety level k and an initial file extent $G \geq k$.

As clients insert more records into the file, one of the existing buckets will eventually overflow. The bucket reports this event to a specific *coordinator* node. The coordinator then initiates a split operation which appends a new bucket to the file (with number N) and moves about half of the records of a designated bucket to the new bucket. The bucket that has been split is usually not the one that reported the overflow. Similarly, a bucket underflow prompts a merge operation which undoes the last split. Merges may shrink the file extent back to the initial value G.

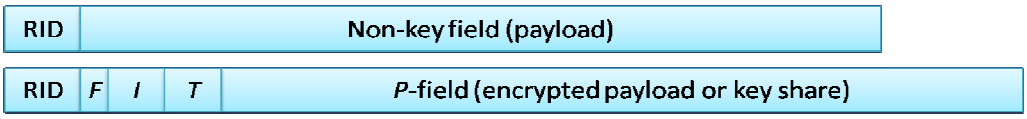

Figure 1: LH* (top) and LH*$_{RE}$ (bottom) Record Structure

An LH* record contains a unique (primary) key called the *Record IDentifier* (RID) to distinguish it from the encryption keys. The RID together with the file state determine the current bucket in which the record resides. The remainder of the record is made up of the non-key field, which contains the application data. LH*$_{RE}$ records add three more fields to the LH* record (Figure 1). The *I*- field identifies the application that has inserted the record and can also contain authorization information, though authorization and authentication are not parts of the scheme. The *T*-field (to be discussed below) identifies the encryption key and is necessary for key recovery and revocation. Field F is a flag

indicating whether a record is a *data record* or a (key) *share record*. Finally, the *P*-field contains encrypted application data of a LH* data record or the key share for a share record.

A record in the LH* file is stored in the *correct* bucket, i.e., with the address given by the *linear hash* function (LH-function) applied to its RID. The LH-function result depends on the current file extent *N*. The coordinator does not push updates to *N* to servers or clients, which might therefore store an outdated *view* of *N*. In this case, the client can make an addressing error. If the client's view of *N* is smaller than the true *N* and the discrepancy results in an error, then the server receiving a request (retrieval, insert, delete, or update) for the false bucket is guaranteed to have a better view of the file extent and can therefore forward the request to a bucket closer to the true one. In fact, at most two forwards can occur. The server with the correct bucket sends an *Image Adjustment Message* (IAM) to any client with an addressing error. The client uses the IAM to update its view. If the client has a view of *N* that is larger than the true value (as the result of one or more merges), then the client can send a request to a server with non-existing bucket. A time-out or normally an error message informs the client who then resends the request to the server that would have hosted the record when there were only *G* servers and sets its view of the file extent to *G*. The operation proceeds then as usual, probably with forwarding and an update to the client's image.

An LH* file application can also request a *scan* operation from the client. The scan searches in parallel over every bucket and retrieves all records matching given conditions on the non-key contents. For LH*$_{RE}$ the scan is obviously limited to non encrypted data. It is only used for operations related to encryption keys.

2.2 Operations on Data Records

The application interacts with LH*$_{RE}$ as with any other LH* structure. Its local client translates application records to LH*$_{RE}$ records and hides the structure of LH*$_{RE}$ records. Given a LH* data record (Figure 1) with RID *r*, the client generates the associated LH*$_{RE}$ record (with the same RID) as follows.

When the client creates an LH*$_{RE}$ data record from an LH* record, it retains the RID *r*. It sets the *F*-field to indicate a data record. The *P*-field receives the payload encrypted with a key. The keys are stored in an *encryption key chain* organized as a table **T[**0,…,*t* –1**]**. The client selects a specific key based on the RID and the table size, e.g. as **T**[*i*] with $i \equiv r \bmod t$. It stores *i* in the *T*-field. For the encryption, we can use any strong, fast, symmetric method. Finally, the client identifies the creating application in the *I*-field that can also be made to contain access rights data.

The key chain can be extensible. The client generates the keys using a cryptographically secure random number generator.

Reversely, given an LH*$_{RE}$ data record, a client uses the *I*-field to determine the generating application and determine access rights. It then uses the *T*-field to determine the encryption key. With the key, it decrypts the *P*-field, yielding the LH* payload. The RID remains the same.

For an insert, a client creates the LH*$_{RE}$ record and inserts it as any LH* record. For a record lookup, the client uses the RID to calculate the address. The correct bucket sends the data record, if found. The client recreates the LH* record and hands it to the application. Updates and deletes are processed analogously.

2.3 Encryption Key Operations

2.3.1 Key Backup

The client backs up every key in the key chain **T** at the servers. It uses secret splitting to create *k*+1 share records of each key. The client creates shares for an encryption key *C* as follows. It first generates (*k*–1) random strings of the same length as *C*. These become key shares $C_0, C_1 \ldots C_{k-1}$. The last share is $C_k = C_0 \oplus C_1 \oplus C_2 \oplus \ldots \oplus C_{k-1} \oplus C$. We recover *C* from the shares just by XORing all shares. The client also needs to generate a RID for each key share. The set of RIDs needs to fulfill two conditions: (1) Knowing the RID of one (or more) shares should not allow any conclusions on the RID of other shares. (2) In a file with extent *G*, the *k* shares need to be stored in different buckets. The requirements on share records provide the central property for the correctness of our schema: No two shares or messages including a share *ever end up in the same bucket* as long as the file retains at least *G* buckets. Otherwise, we could not guarantee *k*-safety. We prove the property in Section 3. Finally, no key should be used for encryption that has not already been backed up.

2.3.2 Key Recovery

Key recovery is the process of recovering encryption keys from backup. The operation can become necessary for a variety of reasons: A user may lose the laptop that served as the client. A company may need to recover the key(s) of an abruptly departed employee. In an emergency, a patient health data may need to be decrypted without his (otherwise required) consent. A court order may mandate data decryption for a variety of reasons, etc.

Key recovery uses the LH* scan operation over the *I-, T-*, and *F*-fields. Each server finds all records that are shares, created by the application, and possibly have a specified offset into **T**. The latter can be used to only recover specific keys or to partition a large recovery process. The client sorts all key shares by the key of which they are shares and XORs the payload in order to recover the key.

In addition to key recovery, we also allow recovery by certain authorized sites. This fact needs to be known to all servers or (more easily) encoded in the *I*-field. In this case, the key recovery operation can specify that the recovered keys are sent to a different client.

2.3.3 Key Revocation

In some cases, a client or an authority needs to revoke one or more keys. This need can be caused by a theft of a laptop with key chain or if access by an employee needs to be terminated, etc. First, the key to be revoked might need to be recovered. Using a scan based on the *I-, T-,* and *F*-field, all

records encrypted with the key to be revoked are recovered, their *P*-field re-encrypted with a newly created and backed-up key that takes the place of the revoked key in the key chain, and reinserted into the LH*$_{RE}$ file so that all servers replace the previous version of these records.

2.3.4 Scalable Number of Keys

Many keys for a small file are burdensome, but for a larger file, the expected disclosure of records in case of a successful attack shrinks as the number of encryption keys used increases. LH*$_{RE}$ allows for an increase in the number of keys *t* used by a client. To limit disclosure, keys should be used for encrypting about the same number of records. One could use a key only for a certain number of records and then create a new one or one could use a linear hashing scheme that places records encrypted with the same key into a virtual bucket. During a split of the virtual bucket, about half of the records would be assigned a new virtual bucket, which triggers rekeying these records with a newly created key. Merging these virtual buckets and removing the last key created in this scheme seems to be hardly of any practical interest.

3. Analysis

We first define our threat model. Next, we focus the analysis on the safety and assurance of an LH*$_{RE}$ file. We then analyze the storage occupancy and access performance.

3.1 Security Analysis

3.1.1 Thread Model

The novelty of LH*$_{RE}$ is tied to its distributed character. We are therefore not interested in the local effects at an intruded client, as any such intrusion poses the same dangers as in a stand-alone environment. We assume that authentication and authorization of record access at servers successfully isolates an intruded client from the data belonging to other clients. This leaves one advantage that an intruder to a client can gain (in addition to control over the application and its data), namely information on the file state and the location of buckets known to the client. Since this analysis is a bit more involved than space allows, we have restricted discussion to a technical report [XX09]. We also assume that snooping network traffic is impossible, e.g. because of the use of Virtual Private Networks (VPN). Furthermore, we assume that the coordinator is secure and that the information that clients receive in an IAM about the location of buckets is correct. (For example, servers receive coordinator signed certificates about bucket locations from the coordinator.) In this paper, we use a *basic threat model*, in which an attacker can gain access to one or more servers capable of hosting buckets, without knowing initially which bucket they host. The attacker is then limited to finding key shares at this server, either in storage or in transit. We further assume that a server stores at most one bucket and does not host a different bucket after it has once hosted a bucket. This situation could otherwise arise from a history of splits and merges.

3.1.2 Key Safety

LH*$_{RE}$ is *k*-safe, which, as we recall, means that an intruder has to break into at least K (= k+1) servers in order to find all key share records belonging to a certain key. The intruder can capture a key share by either finding it in the bucket of an intruded server or by obtaining it in transit. For example when a client creates a key and stores the key share records in the file, they are not send necessarily directly to their final destination but can take an additional hop and occasionally even two hops. The proof of *k*-safety requires some notation. When a bucket i splits into a new bucket i located at the same server and a bucket j, then we call i an ancestor of j and j a descendent of i. Recall that an LH*$_{RE}$ file has a minimum extent of $K := k + 1$ and an initial extent of G, $G \geq k + 1 = K$. We define a descendent set D_i, $0 \leq i \leq k$, to be the set of all bucket numbers of descendents of a bucket i, descendents of descendents of i, descendents of descendents of descendents of i, ... when the file had an initial extent of k.

Examples: Assume that $k = 3$ (and hence $K = 4$) and set $G = 4$. The "original" four buckets are those numbered 0, 1, 2, and 3. The LH* rules give $D_0 = \{0, 4, 8, 12, ...\}$, $D_1=\{1, 5, 9, 13, ...\}$, $D_2=\{2, 6, 10, 14, ...\}$, and $D_3=\{3, 7, 11, 15, ...\}$, which in this case are the set of integers equivalent to 0, 1, 2, and 3 respectively modulo 4.

LH* rules (see appendix A) imply the following, (assuming that the file extent never falls below k:

- The sets D_i are mutually disjoint and their union is the set of natural numbers.
- A descendent of a bucket with number in D_i has always a bucket number in D_i.
- An ancestor of a bucket with number in D_i has always a bucket number in D_i.
- A RID-based query (insert, delete, retrieval, edit) for a record in a bucket in D_i is always sent to a bucket in D_i even if the client has an outdated view. If it is forwarded, then the message only passes through buckets with number in D_i.
- A scan results in queries to all existing buckets, according to the view of the client. If a query to a bucket with number in D_i results in a scan query forwarding, then the forwarded message is sent to a bucket in D_i.

By virtue of how key shares receive a RID, each one of all key shares of a newly created key are placed in a bucket in a different set D_i. Key shares can migrate to other buckets, but only to and from another bucket in the same set D_i. A RID-based query might be routed through another bucket, but only to one with number in the same set D_i in which the key share resides. Similarly, a scan query (which contains information about a key) is only forwarded from a bucket in D_i to another bucket in D_i. As a result, an attacker can only gather all key share records belonging to the same key if he has intruded into at least one bucket with number in D_i (but might need to intrude all buckets in D_i for this information). This implies *k*-safety, as an intruder needs the encryption key to access the contents of the record. If a client sends a scan request to buckets 0, 1, ... , k, then these scan requests will be

forwarded to all buckets, but in a manner where the scan request to i only gets forwarded within D_i. Therefore, scan based operations can be performed without any danger of revealing encryption key data to other buckets.

3.2 Assurance

LH*$_{RE}$ file safety (by electing and adjusting k) gives a simple measure of confidence that an intruder cannot read or write any data records. However, it gives only a lower bound on the number of systems a successful attacker would have to intrude. Typically, the required number of intrusions is higher. We define *assurance* to be the probability that an intrusion into x out of N buckets does not disclose any key. Assurance depends on the number of intrusions x, the number of buckets (each located at a given server), the number of keys, and obviously, on the choice of k. For x, we only count intrusions to servers containing a bucket. We can use assurance calculations to obtain the average number of keys obtained by the intruder and from it *disclosure*, the expected amount of records now accessible to the intruder.

The conclusion of our analysis below is that a typical LH*$_{RE}$ file even with a very low parameter of k and a moderate number of buckets has surprising (to us) high assurance and low disclosure. Adjusting the number of encryption keys a client uses does not change the expected portion of keys revealed, that is, disclosure does not depend on the number of keys revealed. We define *conditional disclosure* to be the proportion of records disclosed given that the attacker received a key. Conditional disclosure describes the expected amount of damage in a bad case. Our analysis reveals it to be controllable by adjusting the number of keys.

3.2.1 Single Key

We first calculate 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, ($K = k + 1$), 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, which is also the number of possible attacks. For an attack to be successful, the attacker has to have attacked all K sites with a key share. Of all his attacks, $x - K$ went to sites that did not have a key share, of which of course there are $N - K$. The number of successful attacks is hence $\binom{N-K}{x-K}$. 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 $q_1(N,x,K) = 1 - p_1(N,x,K)$.

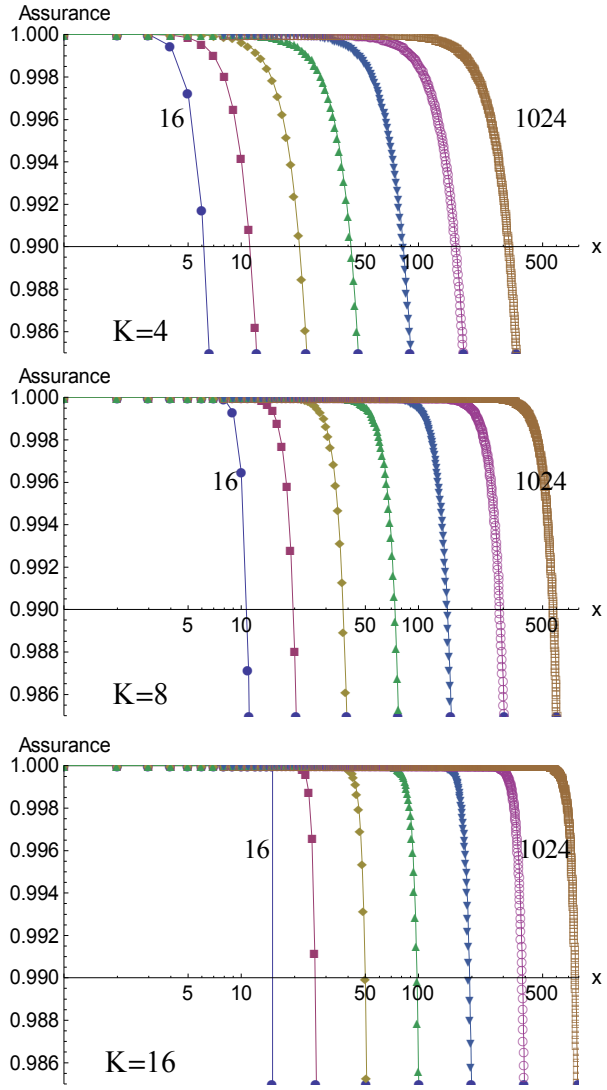

Figure 2: Assurance in an LH* file with K = 4, 8, and 16 key shares (top to bottom) extending over 16, 32, 64, 128, 256, 512, and 1024 servers. The x-axis is chosen to show the 99% (two nines) assurance level.

Figures 2 and 3 give the assurance for an LH* file with K = 4, 8, and 16 key shares and 16, 32, 64, 128, 256, 512, and 1024 sites. We plot the value of $q_1(N,x,K)$ and vary the number x of intruded sites. Since we would often be given a required assurance (expressed in number of nines), we drew the x-axis at the 99% (two nines) in Figure 2 and at the 99.999% (five nines) level in Figure 3. Even for moderately large files the required number of intrusions has to be much larger than x. For example, for a file with extent $N = 512$ and K = 8, the intrusion needs to be to ~ 300 sites to have the assurance fall below the 99% level. The almost even spacing of our curves on the logarithmic x-axis show that the ratio of intruded sites over total sites for assurance below a certain level is almost constant, though slightly decreasing with N.

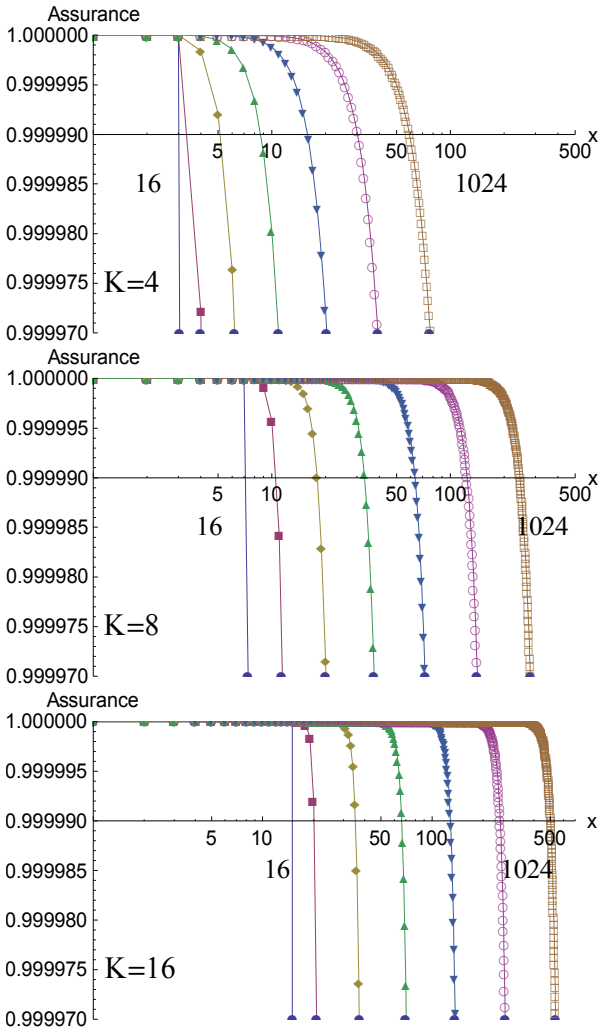

Figure 3: Assurance in an LH* file with K = 4, 8, and 16 key shares (top to bottom) extending over 16, 32, 64, 128, 256, 512, and 1024 servers. The x-axis is chosen to show the 99.999% (five nines) assurance level.

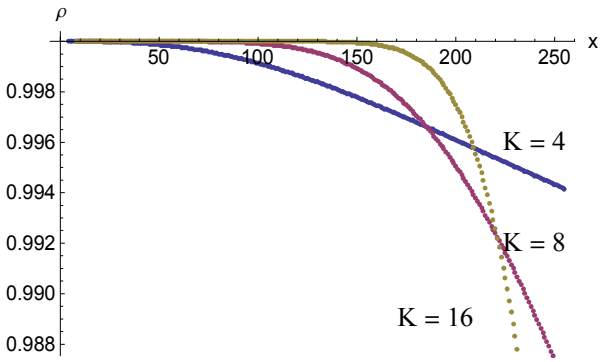

Figure 4: Ratio ρ of assurances with random placement over assurance with the LH*$_{RE}$ placement scheme for N = 256 sites, K = 4, 8, and 16.

An alternative placement scheme to LH*$_{RE}$ is to just distribute key shares randomly among the sites. In this case, the probability of obtaining one specific key share is x/N, the probability of obtaining all K is the K^{th} power of this value, and the assurance under this random placement scheme (given x)

$$p_1^{[\text{random}]}(N,x,K) = 1 - \left(\frac{x}{N}\right)^K$$

We measure the difference by calculating the ratio $\rho = p_i/p_i^{[\text{random}]}$. Figure 4 gives the result for N = 256 and K = 4, 8, and 16. As we see, the effect of the LH*$_{RE}$ scheme is always beneficial, though not pronounced for small number of intrusions, as the probability of having two keys located by chance at the same site is small. As x increases, the benefices of our scheme increase. As K increases, ρ stays closer to 1 for smaller values for x, but falls faster as the number of successfully site attacks increases. This probabilistic evaluation shows that LH*$_{RE}$ placement of key shares is superior, but hides that LH*$_{RE}$ gives guarantees.

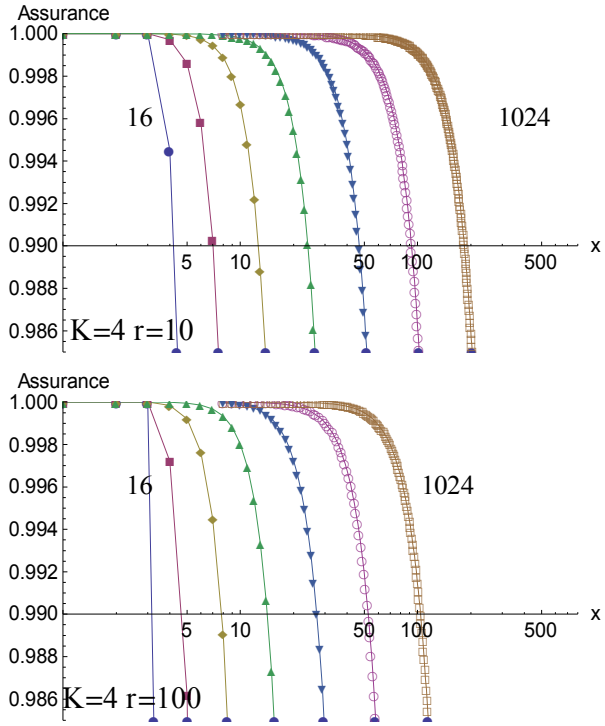

Figure 5: Assurance in an LH* file with K = 4 and r = 10 and r = 100 keys. We vary N from 16 to 1024. The x-axis shows the two nines assurance level

3.2.2 Multiple Keys

The assurance against retrieval of one key out of r keys is $q_r = (1 - p_1(N,x,K))^r$. Certainly, having multiple keys increases the chances of an attacker to gather at least one key after intruding into x sites. Figures 5, 6, and 7 show the effects of changing r. As was to be expected, assurance drops, but still remains quite high.

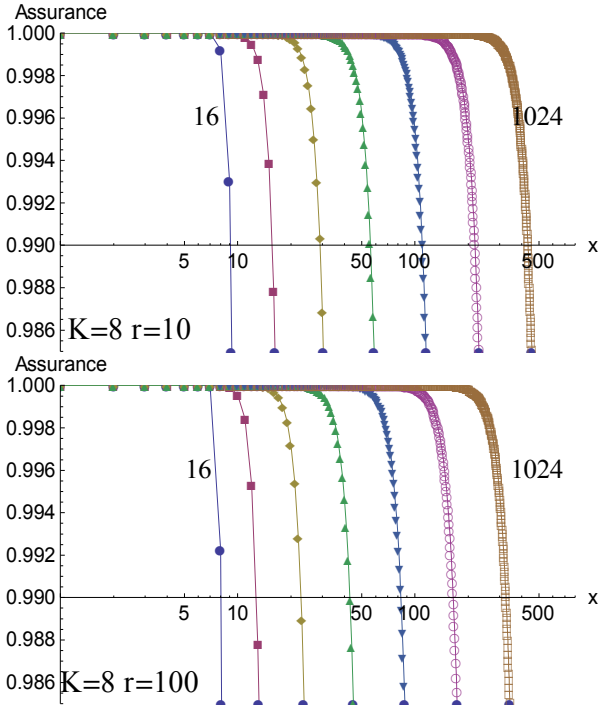

Figure 6: Assurance in an LH* file with *K* = 8 and *r* = 10 and *r* = 100 keys. We vary *N* from 16 to 1024. The x-axis shows the two nines assurance level

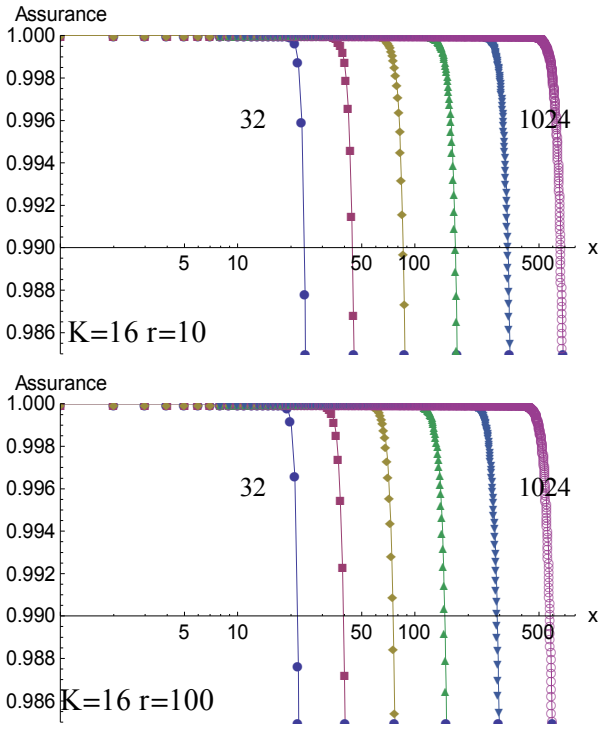

Figure 7: Assurance in an LH* file with *K* =16 and *r* = 10 and *r* = 100 keys. We vary *N* from 16 to 1024. The x-axis shows the two nines assurance level

We notice that the influence of *r* diminishes with increasing *K*.

3.2.3 Disclosure size

The *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 she has gathered all key shares. She possesses now a given key with probability $1 - q_1 = \binom{N-K}{x-K}\binom{N}{x}^{-1}$. Since on average, she has obtained a proportion *x/N* of all data records encrypted with this key, and since she needs encryption key and data record to obtain access to an application record, she obtains on average a proportion of

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

of all application records encrypted with this key. 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, expected *disclosure does not depend on the number of encryption keys used.*

As the number of keys increases, the number of records protected by a given key gets smaller and the amount of data disclosed in a breach becomes smaller. We capture this notion in what we call *conditional disclosure*. Conditional disclosure is defined to be the disclosure (measured again as a proportion of accessible application records over total application records) under the assumption 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 for a successful attack on a specific key is

$$p_1 = \binom{N-K}{x-K}\binom{N}{K}^{-1}$$

Our model implies that obtaining one key and obtaining another one (through the x attacks) are events independent of each other. The probability of obtaining at least one out of *r* keys is

$$P = 1 - (1-p_1)^r$$

We notice that

$$P = \sum_{s=1}^{r}\binom{r}{s}{p_1}^s(1-p_1)^{r-s}$$

The conditional probability of obtaining exactly *s* out of *r* keys given that we obtain at least one key is

$$\binom{r}{s}\frac{{p_1}^s(1-p_1)^{r-s}}{P}$$

and the expected number of total keys obtained given that we obtained at least one is

$$E = \sum_{s=1}^{r}\frac{s\binom{r}{s}{p_1}^s(1-p_1)^{r-s}}{P}$$

Now, this is the number of expected values of the binomial distribution divided by *P*. Thus, *E* evaluates to

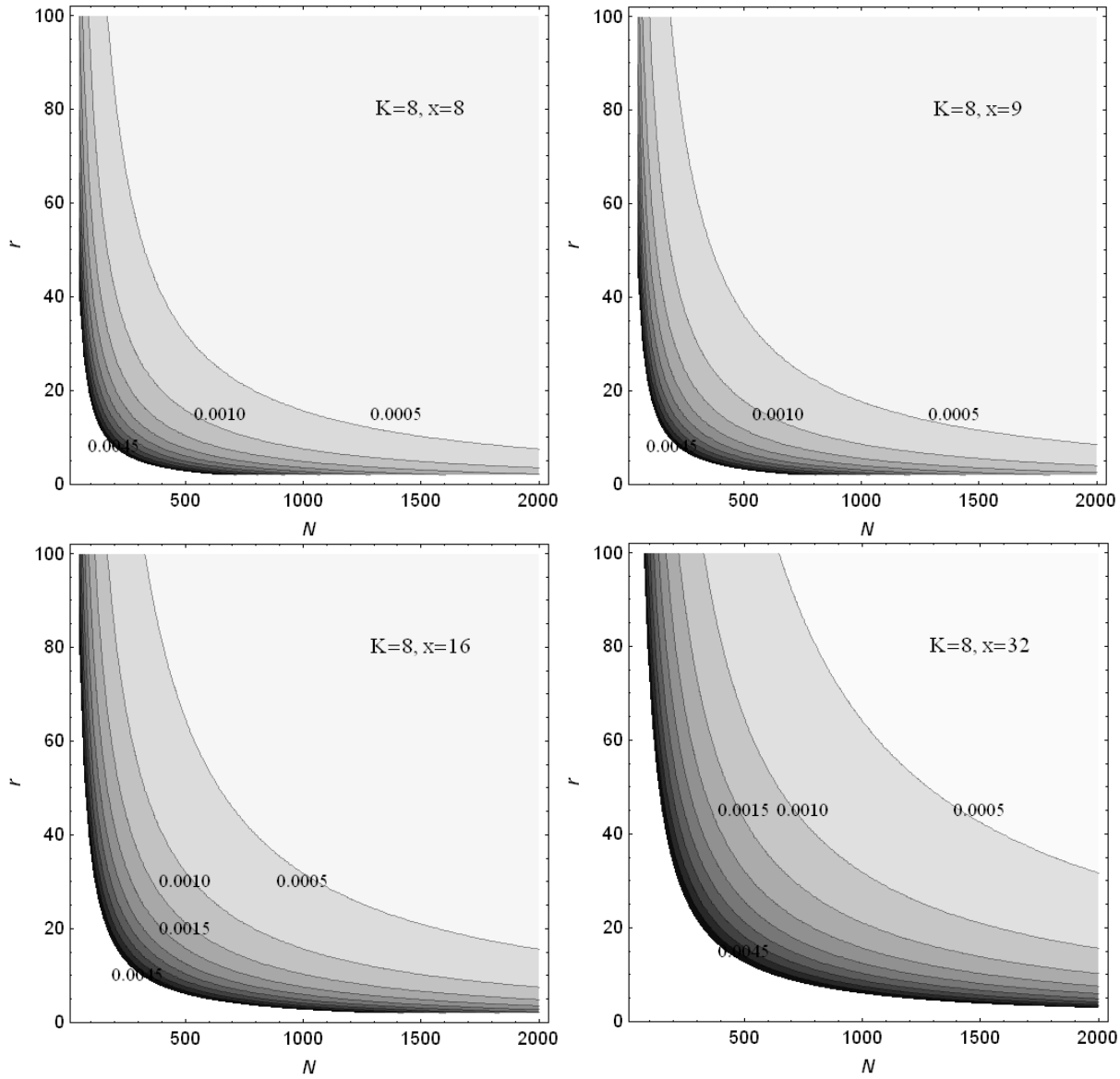

Figure 8: Contour Graph for the conditional disclosure. We vary *N*, the number of sites, and *r*, the number of keys. We set *K*, the number of shares to 8, and show figures for *x* = 8, 9, 16 and 32 intrusions. The upper right corner of each picture has close to zero conditional disclosure.

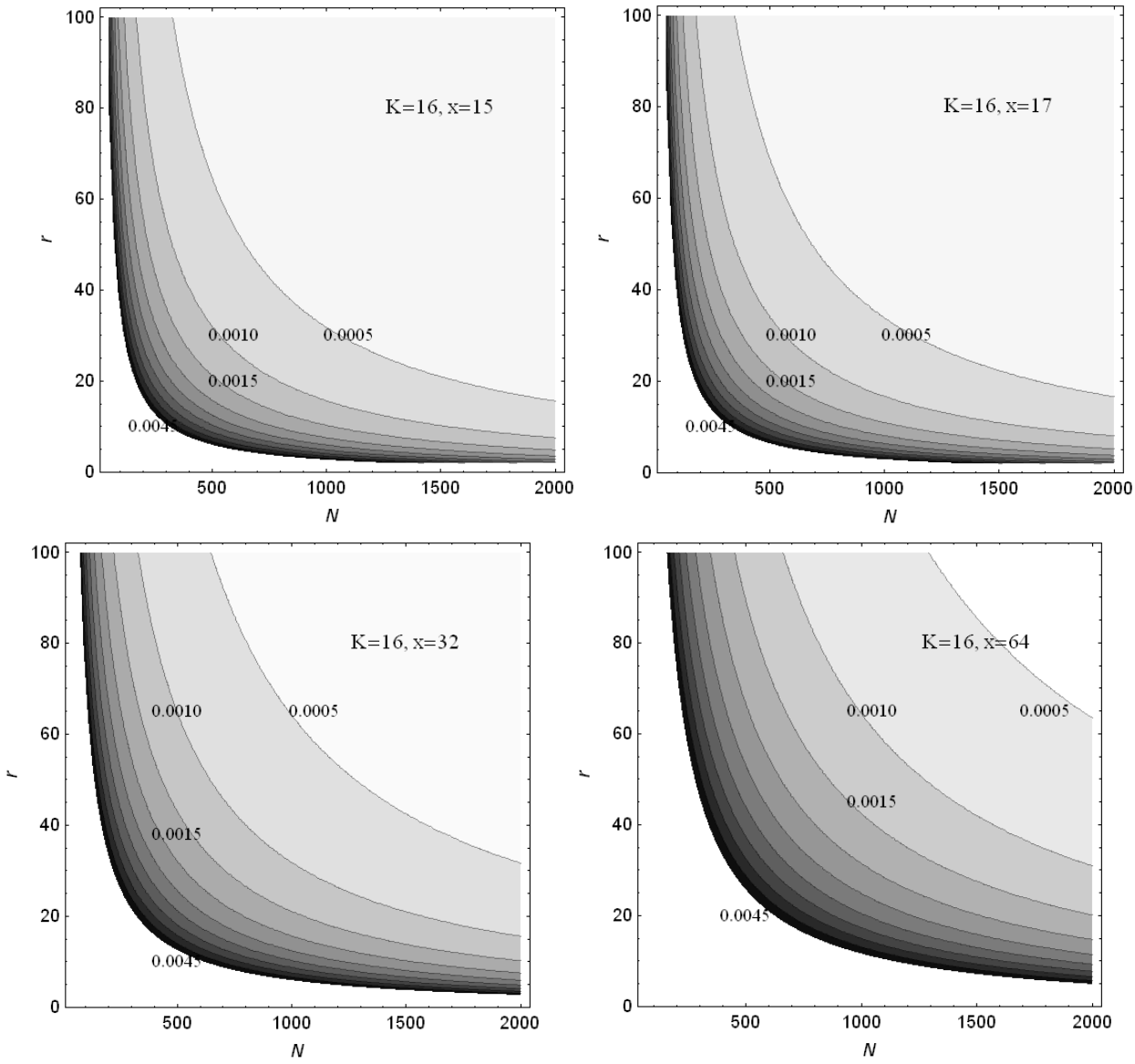

Figure 9: Contour graph for conditional disclosure for $K = 16$.

$$E = \frac{r\, p_1}{P}$$

The proportion of records encrypted with s out of r keys is s/r. Consequentially, E/r of all records are expected to be encrypted with a key that the attacker now possesses. The expected proportion of LH*$_{RS}$ data records obtained with x intrusions into N buckets is x/N (even though of course the expected number in LH* buckets depends usually on the bucket number). 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

$$\frac{p_1 x}{PN}$$

Figures 8 and 9 plot the conditional disclosure for selected values of K in a contour plot. There, we tread the conditional disclosure as a function of N and r. The shaded regions correspond to those areas where the conditional disclosure is less than a given value, ranging from 0.0005 to 0.005 (half a percent of all records revealed).

Our results in conjunction with the independence of the expected disclosure shows that adjusting r allows us to very effectively control the expected amount of disclosure assuming that a disclosure happened. We pay for this control in the higher number of incidents of disclosure. This type of trade-off between incident rate and incident effect might become very useful for obtaining or pricing an insurance policy to cover the effects of an intrusion.

3.2.4 Refined Measurement of Disclosure Costs

Up till now, we have tacitly assumed that the disclosure of any record costs the storage provider the same. However, this might often not be the case. We now consider a scenario where there is a fixed costs associated to any disclosure of records in addition to a costs per record disclosed. For example, assume a health care provider, credit card processing center, or a university registrar service center that store their data in a LH*RE data structure on many servers. Entities of these types are regulated and often forced to publicly acknowledge each breach of privacy of data. In addition, they are likely to incur costs in mitigating the damage for each person involved, which is usually proportional to the number of records revealed. We model this scenario by assigning a value of 1 to the maximum damage done, by assigning a value α, $0 < \alpha < 1$, to the fixed costs of a disclosure. Correspondingly, each of the total s records that is disclosed costs $(1 - \alpha)/s$. Assuming r keys of which i have been disclosed, the proportion of records disclosed is i/r of the x/N records that the attacker can access. Thus, the disclosure costs for $i > 0$ is $\alpha + (1 - \alpha)\frac{i\, x}{r\, N}$ and the expected disclosure cost is

$$\sum_{i=1}^{r} \alpha + (1-\alpha)\frac{i\, x}{r\, N}\binom{r}{i} p_1(N,K,x)^i (1 - p_1(N,K,x))^{r-i}$$

Figure 10 gives the results calculated for $K = 8$, $N = 100$, and $\alpha = 0$, 0.1, 0.5, and 1 depending on the number of intrusions

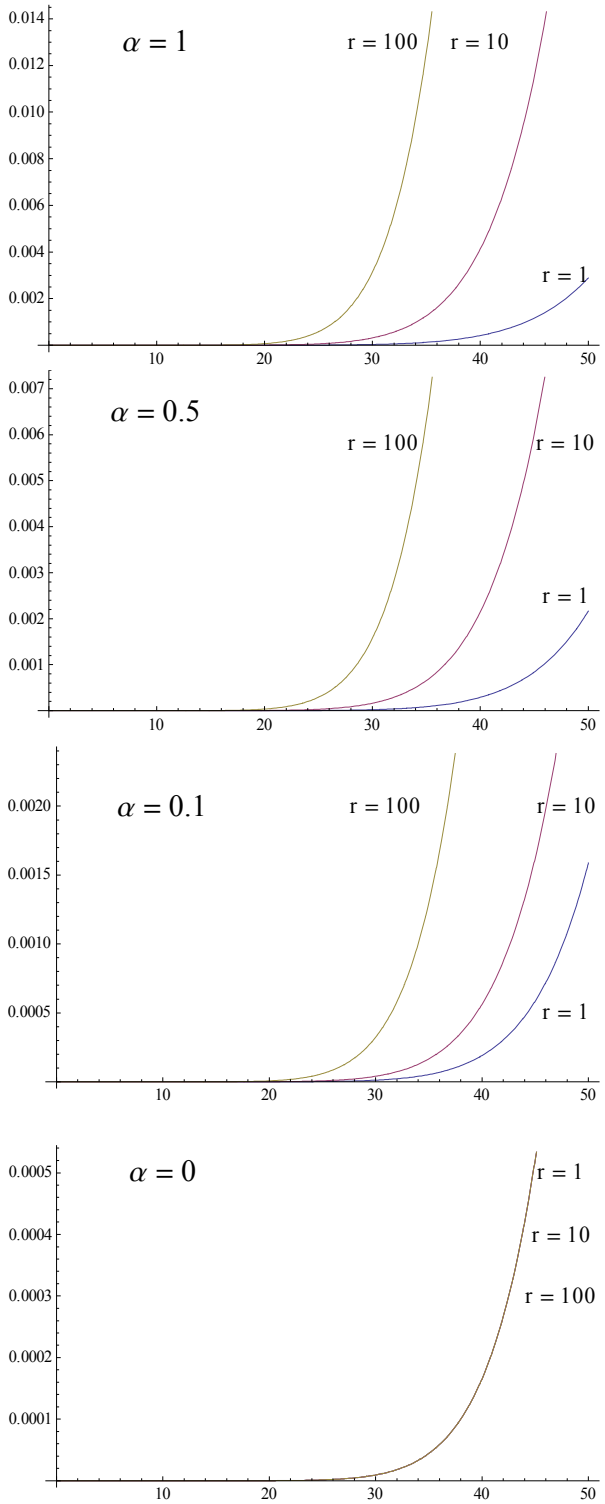

Figure 10: Refined Disclosure Proportion for $N = 100$, $K = 8$, $\alpha = 0$, 0.1, 0.5, 1, and x (x-axis) varying between 0 and 50. Notice the different scales on the y-axis

x. We can see how disclosure increases with increasing α, but also how the number of keys r becomes more important. Not too surprisingly, with this cost model, a single key should be used.

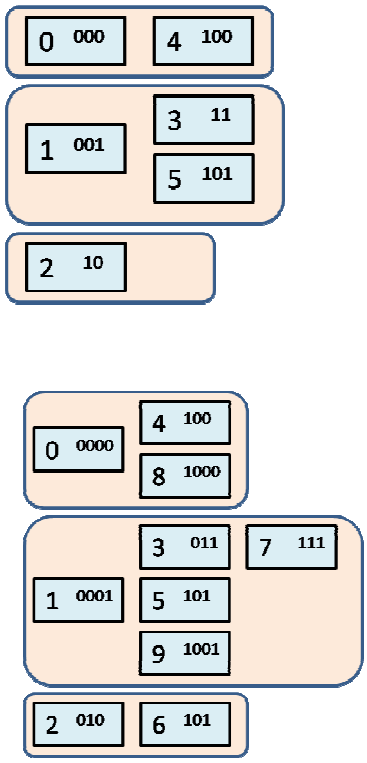

Figure 11: Small LH* structures with 6 buckets (top) and 10 (bottom) buckets broken into descendent sets for buckets 0, 1, and 2

3.2.5 Refinements of the Intrusion Scenario

Our basic intrusion scenario limits the capability of the attacker perhaps more than is reasonable to assume. Take for example an intruder that has taken over a client and now "owns" all its data. This data happens to be stored in a LH*$_{RE}$ file with data not accessible by the client. The attacker wants to gain access to some of this data as well. The attacker has additional information not available to an attacker under our basic thread model. First, some LH* buckets are about twice as large as the others (namely those not yet split in the current round of splits). These contain more key share records. Second, the attacker knows bucket numbers and therefore descendent sets. If the attacker needs a specific encryption key and has already found a key share record, he knows that other buckets in the same descendent set cannot contain other key shares and therefore remove them from his target list. All this can be modeled as interesting optimization problems for the attacker, but its treatment requires advance knowledge of LH* properties, relegated to Appendix A. It turns out that access to size information or even location information, while useful, does not dramatically change the picture. We observe that a true attacker would triage servers based on the results of a vulnerability scan. The costs of an attack to a site depend on the site and vary among sites, making our model less interesting. However, modeling different degrees of vulnerability is difficult since assigning any numerical values to sets of sites is simply too arbitrary and development of a generic theory of how to model vulnerability of individual sites certainly beyond the scope of our work here.

Thus, the scenario we consider here is the following. The attacker has obtained a complete map of the installation that tells the attacker where which bucket is located. We assume that breaking into any site costs the attacker a constant amount, either in money, or in work, or in a combination of both.

We first use the examples given in Figure 11 to illustrate the advantages that the attacker can gain from his/her knowledge. Both examples use $K = 3$ (a somewhat unrealistically low value, but good enough to explain the principle). Figure 11 gives the descendent sets for the original buckets 0, 1, and 2. In addition, we label each bucket with the binary number representing h_{level}, the hash function evaluated at the given level. In the top of the figure, Bucket 3 has label 11, which means that membership of a RID in this bucket is evaluated at level 2. We notice, that Buckets 3 and 2 contain about twice as many records as the remaining buckets, because the split pointer is 2 and they have not yet been split in the current epoch. Similarly, at the bottom, the split pointer is at 2 and Buckets 2, 3, 4, 5, 6, and 7 have twice as many records (on average), whereas Buckets 0 and 1 have been split into Buckets 8 and 9 and thus contain about half as many records as the remaining buckets.

If N were to fall to 3, then a share would be found in bucket 0, 1, 2 each. If there are only 6 buckets, then the key share originally in bucket 2 would still reside in Bucket 2. Therefore, an attacker *has* to intrude into the site with Bucket 2. In the second example, to obtain the key share originally in Bucket 0, the attacker has to attack Buckets 0, 4, and 8. Bucket 4 contains about half the records in the descendent sets, whereas Buckets 0 and 8 contain the other half. Therefore, the attacker *should* intrude into Bucket 4 with priority.

Let us consider the attacker's optimization problem in Figure 11 in more detail. Assume that the attacker plans x intrusions, of which x_0 are directed to the descendent set of Bucket 0, x_1 to the descendent set of Bucket 1, and x_2 to the descendent set of Bucket 2. To have any chance of success, all values need to be non-zero. Additionally, $x_0 \leq 2$ and $x_1 \leq 3$. We can measure putative success by the probability to receive a single key. If $x_0 = 1$, then the attacker can choose either Bucket 0 or Bucket 4 for the attack and receives key share 0 with probability $p_0 = ½$. For $x_0 = 2$, the probability is $p_0 = 1$. For $x_1 = 1$, the attacker chooses Bucket 3 giving a probability of $p_1 = ½$ to obtain the key share. Adding to x_1 involves

attacking Buckets 1 or Bucket 5 or both, resulting in p_1 = ¾ for $x_1 = 2$ and $p_1 = 1$ for $x_1 = 3$. Finally, x_2 has to be 1, given a success probability of 1. The attacker's optimization problem given a "budget" x of sites to attack is to maximize $p_0(x_0) \cdot p_1(x_1) \cdot p_2(x_2)$ for $x = x_0+x_1+x_2$.

In the general case, we can capture the probability of obtaining key share i in a function $p(i,x_i)$. Assume that there are a buckets in D_i of which b are "big", i.e. not split in the current LH-splitting round. Then

$$p(i,x_i) = \begin{cases} 2x_i/(a+b); & if\ \ x_i \le b \\ (b+x_i)/(a+b); & if\ \ x_i > b \end{cases}$$

For example, in the lower example of Figure 11, D_1 has $a = 5$ and $b = 3$ as Buckets 3 and 5 are big. $p(1,1)$ = ¼ since the attacker will attack one of the two big buckets, that yield him 2/8 of all records in the descendent set. $p(1,2)$ = ½ and $p(1,3)$ = ¾ reflecting attacks on two or all three big buckets. Buckets 1 and 7 only contain 1/8 of all records in the D_1, therefore $p(1,4) = 7/8$ and $p(1,5) = 1$.

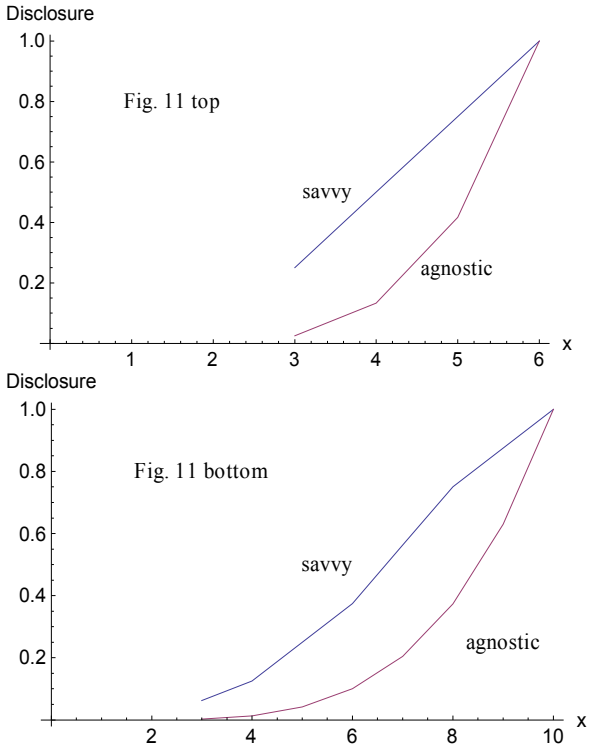

Figure 12: Difference in disclosure between the savvy and the agnostic attacker for the Example in Figure 11.

Figure 12 finally gives the result between the savvy attacker, defined to be the one taking location information into account and optimizing her attack plan accordingly, and the agnostic attacker, who does not use or does not have this information available.

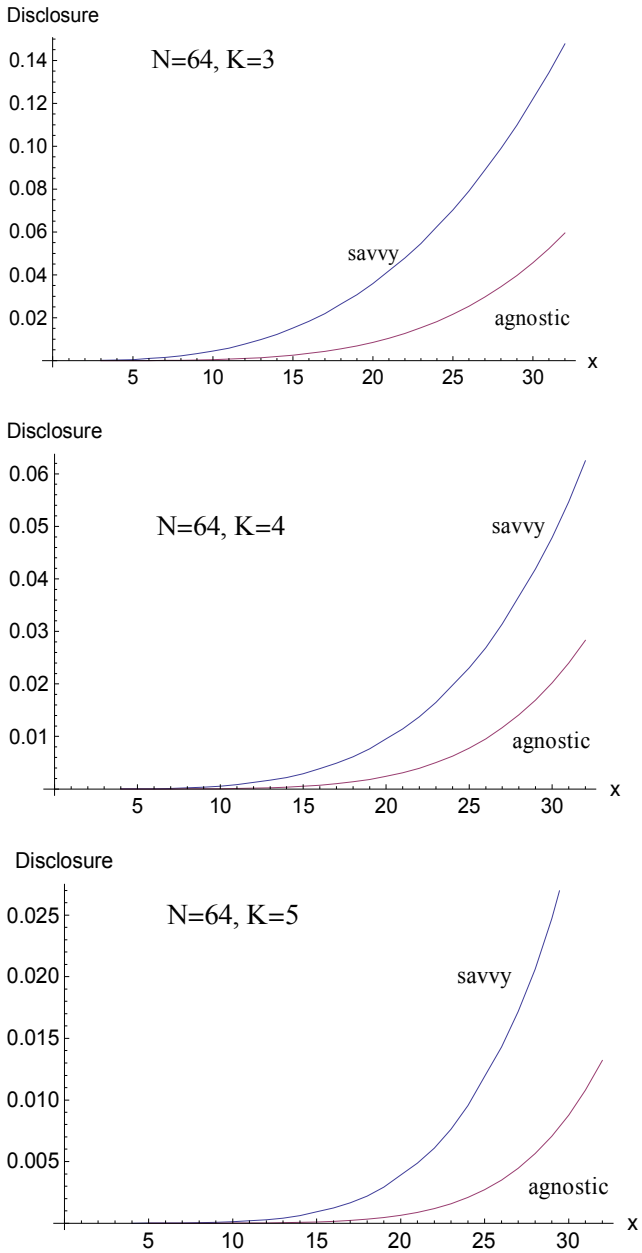

Figure 13: Disclosure amount differences between savvy and agnostic attacker for $N = 64$ and $K = 3, 4, 5$

We now investigate the difference between the savvy and the agnostic attacker for a variety of parameter values. Let the reader beware of the change of scale on the y axis. We give results in Figure 14.

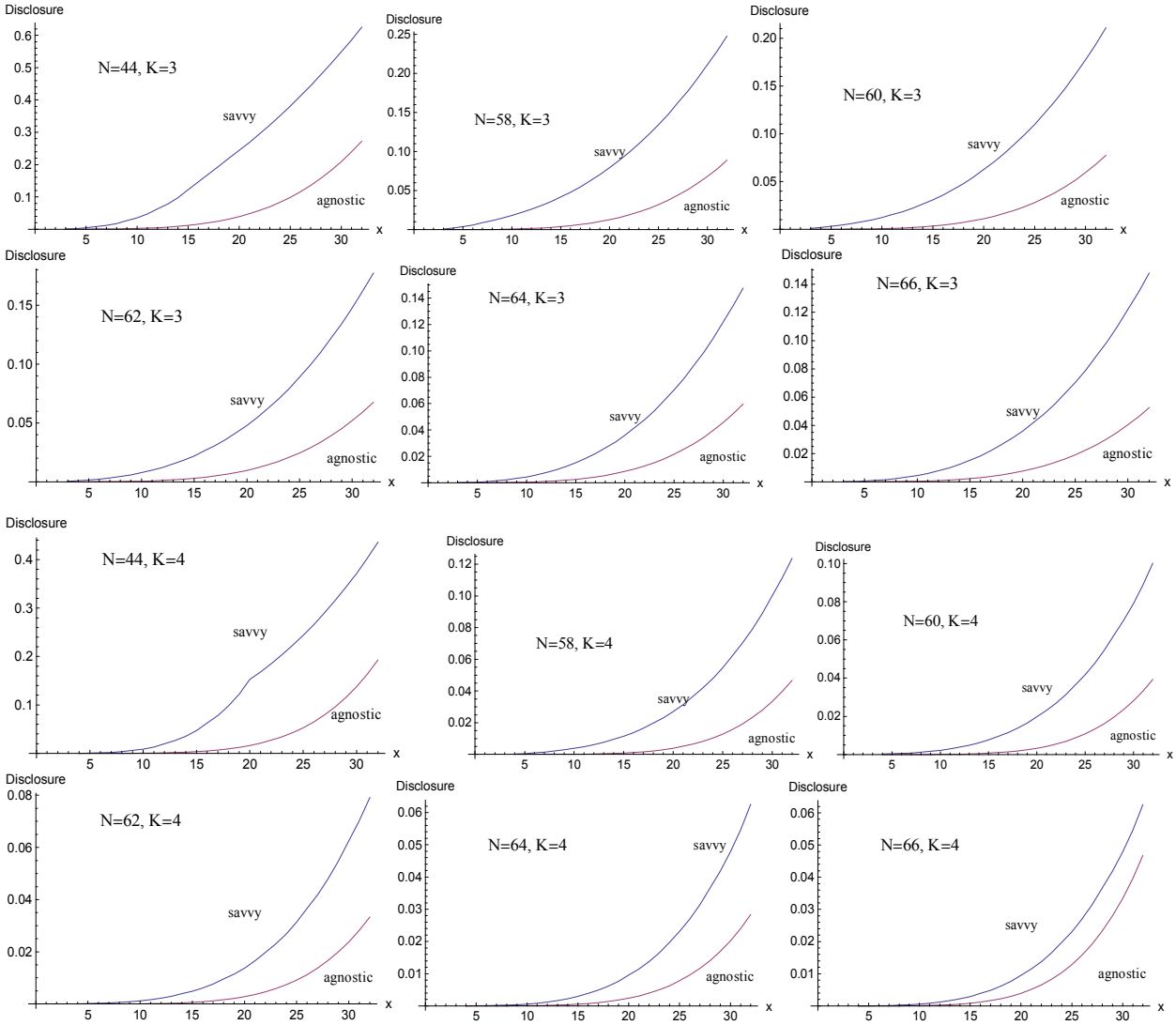

Figure 14: Disclosure differences between savvy and agnostic attackers

The differences between the savvy and the agnostic attackers depend on the number of big buckets versus small buckets, but even more on the ability to avoid to inadvertently allocate all resources to a single dependency set.

3.3 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 $O(N)$. An AES key has 32B, which allows for millions of keys in current RAM and for thousands if the key chain has to fit in the L1 or L2 cache. The effective load factor of an LH*$_{RE}$ file should be in practice very close that to an LH* file of same extent with the same number of data records. We recall that the average load factor of such an LH* bucket is $\ln(2) \approx 0.7$.

Access time to a record is dominated – as usual – by messaging costs and therefore proportional to the number of messages per operation. Cryptographic operations take time proportional to the size of data records. With symmetric encryption they should be negligible. This conjecture still stands in need of experimental verification.

The message costs of record insert or RID-based search are those of LH*. Thus, our variant of LH*$_{RE}$ uses typically one message per insert and two messages per search. In case of a (typically rare) addressing error, this increases by two at worst. The update and delete messaging cost each as for LH* as well.

The message count cost of encryption keys creation is usually $(k+1)N$. A key recovery message cost is that of an LH* scan. It thus consists first of one multicast to send out the scan, or of N unicast messages for this purpose. The timing of the latter depends on the scan processing strategy. Assuming that the client diffuses the scan itself to every bucket in the image, the eventual difference between the maximal number of splits some bucket has encountered according to the client image and the actual one determines the number of rounds of the scan send-out [LNS96]. The message count does not change, but the timing does. The reply from the servers costs at best N unicast messages to the client. Some with shares and some being simple acks, we recall. All together, key recovery costs thus one multicast and N unicasts or $2N$ unicast messages in total, at best. The actual count can be higher depending on the actual implementation.

The scan cost dominates the key revocation cost for a larger file. Assuming only unicast messaging, a scan operation proceeds at best in $2N$ messages to retrieve the records and the key shares of the key to revoke. A practical implementation of this phase should invoke several scan operations, each costing at least 2N messages. This phase is followed by at least $2(k+1)$ messages per key to rewrite the shares, i.e. if **T** has size t, then this amounts to $2t\ (k+1)$ messages. Finally, there are r messages to write r re-encrypted records, where the upper limit for r is the number of records in the file. If however the client only revokes a few private keys for a few records, then the messaging costs is dominated by the $2N$ messages per round in a scan operation needed to find the records. As the rewriting of key share records can now be done in parallel with the scan, the overall timing should decrease towards the time for the scan and the rewrite of the records.

4. Variants

We now sketch some variants of the basic LH*$_{RE}$ scheme. In technical reports [XX09a] and [XX09b] we already discuss two variants in detail. One uses a distinct key for every data record. The other provides a data record with its encryption key encoded through a *private share* which contains the key and only distributed to entities with access rights to the record. We now discuss further variants.

4.1 Safety Level Managed by Client

We expect a typical LH*$_{RE}$ client to have an exclusive set of encryption keys, for instance, because there is only one such client. Even if the client shares the encryption key with other clients, its shares can be made private. This allows the client to set a private client *safety level k'*. For a single client, simply $k' = k$. Otherwise, $k' = k$ only initially. Each client creates and maintains $K' = k' + 1$ shares and adjusts k' according to its policy based on its view of the file extent. (In fact, the same client could have different k' and different policies for different record classes encrypted with mutually disjoint sets of keys.) This *safety policy* of the client uses the perceived view of the file extent. Whenever the client receives an IAM, it calculates whether the policy triggers a change in the number of key shares.

We recall that the encryption key is the XOR of the payload of the key shares. To increase the number of key shares for a given key from K to L, the client retrieves the RIDs of all existing key shares, and at least some shares themselves. Each of the K key shares is located in a different of the K buckets of a file of extent K. Therefore, they would be also located in different buckets of a file of extent L. It is therefore possible for the client to find additional $L - K$ RIDs for the new records such that each of the L old and new key shares would be located in a different one of the L buckets of a file of extent L. The client then generates random $L - K$ new key shares and adjusts one or more of the retrieved key share records so that the XOR of all shares remains the same, namely the encryption key. Finally, the client stores the new and changed key share records. This operation proceeds in parallel for all encryption keys to which the policy applies.

If the file shrinks, the client safety level might be equal or larger than the number of buckets. The client is unaware of this situation until an IAM informs it. Now, the client has to shrink the number of key share records from K to L. It turns out that in a file of extent L, each bucket contains at least one key share. If there are multiple key shares in a (hypothetical, since the actual file extent might be larger) bucket, then the client retrieves the key shares. The client then XORs these key shares themselves and creates a new key share record with the RID of one of them. Obviously, both operations maintain the two important invariants of key share records:

The XOR of all key shares is the encryption key; and, if there are K key shares, then they would reside each in a different bucket of a file with extent K.

4.2 Adapting to Other LH* Schemes

It is useful to incorporate the high-availability into the scheme, especially to avoid any losses of shares. The simplest way is to use LH*$_{RS}$, [LMS5]. Its use of erasure correction protects keys in a similar manner to (k, l) secret sharing with $l > k + 1$, [PHS3]. Alternatively or simultaneously, key share records might be made more resilient to the unavailability of a bucket by using these advanced secret sharing schemes. Furthermore, there are no obstacles to applying LH*$_{RS}^{P2P}$ for a P2P variant. It offers the advantage of minimizing message forwarding [YS09].

4.3 Generalizing to Other SDDSs

Correctness of LH*$_{RE}$ is based in the disjointness and other properties of descendent sets of some initial set of buckets. Other well-known SDDSs have the same or similar properties. This is the case for RP*$_S$ [LNS94], which allows for range queries, and the more recent BigTable [S&al01] which has a very similar basic structure to RP*$_S$. Chord [S&al01] partitions using the principles of consistent hashing also has disjoint descendent sets. The first obstacle for adapting LH*$_{RE}$ principles is the forwarding algorithm that needs to change to prevent different key shares to traverse the same node. The second obstacle for RP* and BigTable is the protection of the record keys, whose confidentiality needs to be protected.

5. Conclusion

The LH*$_{RE}$ scheme allows an application to manipulate data records without any encryption related messaging overhead. It offers insert, search, update and delete operations with the same (messaging) performance as LH* itself. The storage requirements are similar. The storage overhead for the encryption keys at the client and at for the secrets at the servers may be negligible, e.g, may be as low as a few dozens of bytes at the client and the servers, all together. Successful attacks on the servers need to be massive because of k-safety. The applications can adjust easily this parameter dynamically. The assurance analysis has shown that the file is resilient against intrusions several times larger than k. For instance, a 100-bucket file, thus rather smaller file by today standards, may offer at least the 5-nine assurance for an intrusion into up to 40 buckets, simply by choosing k = 7 as file safety level, i.e. by choosing eight as the number of shares per key.

We have derived formulae and numerical results that offer the file administrator guidance in selecting these parameters. All together, the analysis encourages higher values of k. The costs of a higher safety level only matter in the infrequent case of key revocation and reconstruction.

References

[A&al06] M. Anisetti, C. Ardagna, V. Bellandi, E. Damiani, S. De Capitani di Vimercati, P. Samarati: OpenAmbient: a Pervasive Access Control Architecture, Emerging Trends in Information and Communication Security (ETRICS), 2006.

[A&al08] C. Ardagna, M. Cremonini, S. De Capitani di Vimercati, P. Samarati: A Privacy-Aware Access Control System, Journal of Computer Security (JCS), vol. 16(4), 2008.

[B03] M. Bishop: Computer Security. Addisson-Wesley, 2003. ISBN 0-201-44099-7.

[BCK96] M. Bellare, R. Canettiy, H. Krawczyk: Keying Hash Functions for Message Authentication. Advances in Cryptology: Crypto 96. Lecture Notes in Computer Science Vol. 1109, N. Koblitz ed., Springer-Verlag, 1996.

[C&al06] F. Chang, J. Dean, S. Ghemawat, W. Hsieh, D.. Wallach, M. Burrows, T. Chandra, A. Fikes, R. Gruber: Bigtable: A Distributed Storage System for Structured Data. OSDI'06

[C08] Cleversafe Opensource Community: Building Dispersed Storage Technology. http://www.cleversafe.org/

[D&al05] E. Damiani, S. De Capitani di Vimercati, S. Foresti, P. Samarati, M. Viviani: Measuring Inference Exposure in Outsourced Encrypted Databases, Workshop on Quality of Protection (QoP), 2005

[HILM02] H. Hacigümüs, B. Iyer , C. Li, S. Mehrotra: Executing SQL over Encrypted Data in the Database-Service-Provider, SIGMOD 2002.

[HIM03] H. Hacigümüs, B. Iyer , S. Mehrotra: Ensuring the Integrity of Encrypted Databases in the Database-as-a-Service, Data and Applications Security XVII (DBSEC) 2003.

[HMI02] H. Hacigümüs, S. Mehrotra, B. Iyer: Providing Database as a Service, Proceedings of ICDE 2002.

[LMS05] W. Litwin, R. Moussa, T. Schwarz: LH*rs - A Highly Available Scalable Distributed Data Structure. ACM-TODS, Sept 2005.

[LNS94] W. Litwin, M-A. Neimat, D. Schneider: RP*: A Family of Order-Preserving Scalable Distributed Data Structures. VLDB-94.

[LNS96] W. Litwin, M-A. Neimat, D. Schneider: LH*: A Scalable Distributed Data Structure. ACM-TODS, (Dec. 1996).

[LSY07] W. Litwin, H. Yakoubin, T. Schwarz: 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

[LSY08] W. Litwin, H. Yakoubin, T. Schwarz: Th. LH*$_{RS}^{P2P}$: A Scalable Distributed Data Structure for P2P Environment. NOTERE-08, June 2008.

[MMJ03] A. Mei, L. Mancini, S. Jajodia: Secure Dynamic

Fragment and Replica Allocation in Large-Scale Distributed File Systems. IEEE Transactions on Parallel and Distributed Systems, vol. 14(9), 2003.

[MTN09] Microsoft TechNet. Using Encrypting File System. http://technet.microsoft.com/en-us/library/bb457116.aspx#EIAA

[PHS03] Pieprzyk, J., Hardjono, TH., Seberry, J. Fundamental of Computer Security. Springer, 2003, 673.

[S79] A. Shamir: How to share a secret. Communications of the ACM, vol. 22(11), 1979

[S08] ENCRYPTION STRATEGIES: The Key to Controlling Data. A Sun Microsystems–Alliance Technology Group White Paper Jan. 2008

[SGMV07] M. Storer, K. Greenan, E. Miller, K. Voruganti: POTSHARDS: Secure Long-Term Storage without Encryption. 2007 Annual USENIX Association Technical Conference.

[MZ08] B.N. Mills, T. F. Znati, SCAR - Scattering, Concealing and Recovering data within a DHT. 41st Annual Simulation Symposium, 2008.

[PGP04] Method and Apparatus for Reconstituting an Encryption Key Based on Multiple User Responses. PGP Corporation. U.S. Patent Number 6,662,299.

[S&al01] 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.

[XX09] Anonymous Authors: Technical Report 2009

[YS09] H. Yakouben, S. Sahri: LH*$_{RS}^{P2P}$: A Fast and High Churn Resistant Scalable Distributed Data Structure for P2P Systems. International Journal on Internet Technology and Secured Transactions, (to appear).

Appendix A: Generic LH* Scheme

A.1. File Structure

As for any LH* file, an LH*$_{RE}$ file is a collection of records that belong to a collection of applications in a distributed or networked system. Each node with an application contains an LH* *client* that translates between the application(s) and the distributed storage layer that LH* implements. The records themselves are stored at *buckets*, each located at a *server*. A record contains a unique (primary)key (the RID in this paper) and non-key field(s).

The records at the servers are stored in *buckets* numbered 0, 1, 2... $N - 1$. The number of buckets is also called the *file extent*. A bucket number functions as a logical *address*. Every bucket is located at a different server. A file is created with G buckets 0, 1, ..., $G - 1$. In the case of LH*$_{RE}$, G has to be greater than k, the file safety level. As the file grows, a *split* operation appends a new bucket. We discuss the details below together with the opposite operation that shrinks the number of buckets in the file.

LH* addressing is based on the notion of a family of hash functions h_i. Let $\boldsymbol{C}$ be the set of all possible RID values. A family of hash functions is then a set $\{h_i \mid i = 0, 1, 2, ...\}$ of functions

$$h_i: \boldsymbol{C} \rightarrow \{0,1, \ldots\ 2^i G - 1\}$$

with the property that

$$h_{i+1}(c) = h_i(c) \text{ or } h_{i+1}(c) = h_i(c) + 2^i G.$$

We assume that both alternatives are taken with equal probability. The most popular hash functions are based on taking remainders, namely $h_i(c) = c \bmod 2^i G$.

Every LH* file has a specific component, called the *coordinator*. The coordinator resides at some dedicated node or is a distributed entity. For LH*$_{RE}$, the coordinator does not reside at any server. The coordinator keeps the *file state,* based on the file extent. It is stored as a couple of integer values (l, s), $l,s = 0,1\ldots.$, called *file level* and *split pointer* respectively. The meaning of these names will become clear shortly. LH* addressing

Next, every LH*$_{RE}$ file has a specific component called *coordinator*. The *coordinator* resides at some dedicated node, other than the servers and client nodes. The coordinator keeps the *file state*. This is a couple of values (l, s), $l,\ s = 0,1\ldots.$, called *file level* and *split pointer* respectively. Initially, $(l, s) = (0,0)$. The following *linear* hash function (LH-function) h defines the *correct* bucket $h(c)$ where the record with RID c resides for file state (l,s):

$$\text{if } h_l(c) < s \text{ then } h(c) := h_l(c);\ \text{ else } h(c) := h_{l+1}(c);$$

To avoid bottlenecks, the coordinator does not push file state changes to the clients. Clients therefore use LH* addressing with a sometimes outdated view of the file state. To help route resulting queries directed to a false bucket, each bucket maintains a parameter, called the *level j*. The initial buckets are all created with level $j = 0$.

Every bucket has a fixed capacity b to store records. Even a bucket at capacity can still receive additional records for storage. Such a bucket *overflows*. It stores the additional record(s) in an overflow storage area. It also informs the coordinator of the fact. Upon receiving such a message, the coordinator initiates a split operation. The well-known principle of linear hashing is not to split the overflowing bucket but to split the bucket pointed to by the split pointer s. The principle allows to minimize state information (essentially equivalent to the file extent N) and an overflowing bucket will eventually be split. In more detail, the split operation starts by creating a new bucket, with bucket number N. (In LH*$_{RE}$, the hosting server was selected well in advance so that the same server can only store a predetermined bucket.) The coordinator then sends the split request to bucket s together with the new address of N. It turns out that always $N = s + 2^l G$. Bucket s then recalculates the correct bucket address of all its records using h_{j+1}, where j

is the bucket level. The RIDs c of all records in Bucket s have either $h_{j+1}(c) = s$ or $h_{j+1}(c) = N$. Accordingly, the record remains in bucket s or is send to bucket N. Finally, Bucket s increments its level j and Bucket N receives the same level. After the split has been completed by Bucket s, the coordinator updates the file state by executing

$$s := s+1;\ \text{if } s = 2^l \text{ then } s := 0,\ l := l+1;$$

Since the hash functions are uniform, every split moves about half of the records to the new bucket. The progression of buckets to split is 0, 1…G-1; 0, 1…$2G$ – 1; 0…2^iG – 1; 0, 1…

While most files only grow, LH* also let the file extent shrink if many deletions lead to underflow at buckets. The mechanism is similar. If the number of records falls below a certain threshold (such as b/3), then the *underflowing* bucket reports this fact to the coordinator who initiates a merge. A merge basically undoes the split. We skip the details. Using merges and splits, an LH* file reaches an average storage occupancy of about 70%.

Every server needs to know some physical addresses of other servers, as it will need to forward some queries to other buckets and as it will have to upgrade some clients view of the file. The coordinator has all addresses of existing (and in LH*$_{RE}$ future) buckets. Obviously, a server receives the addresses of all buckets that have been split from the bucket residing on the server. This information is sufficient to allow forwarding, but in order to make updates of client images more efficient, the coordinator provides all bucket servers with the addresses of all servers in use when a bucket splits.

A.2 Record Addressing

LH*$_{RE}$ uses this calculation of the correct bucket given a RID for the *search, insert, delete* and *update* operations. This calculation is only exact if clients have the same file state as the coordinator. However, maintaining file state at clients that might not be active for a long time and even maintaining a list of all clients overloads the coordinator. Therefore, LH* allows a discrepancy between the correct view of the file state (by the coordinator) and by clients and servers. Each client contains a local *image* of the file state. The initial image is (0,0), and the only physical addresses known to the client are those of the initial G buckets. The client uses its image (s', l') to calculate the bucket address of a record and its lists of bucket location to find the correct server.

As the file grows and shrinks, all client images, even if currently correct, become outdated. As a result, a client can make an addressing error. Every server receiving a query verifies therefore whether it has the correct bucket for this RID. If not it forwards the query to a bucket (likely the correct one). The receiving bucket uses the same algorithm.. The servers use the following *Test and Forward* algorithm. Here a is the address of the executing server:

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

Finally, the correct bucket receiving a forwarded query sends an *Image Adjustment Message* (IAM) to the client. The IAM contains the triplet (j, a', j') where a' and j' are respectively the address and level of last traversed bucket, say a'. The client updates then the image as follows.

$i' = j' - 1$; if $j < j'$ or $a > 2^{i'}$: $a := a'$ endif
$n' := a + 1$; if $n' \geq 2^{i'}$: $n' := 0$; $i' := i' + 1$ endif

As a result, no client errs twice in the same way. The IAM contains also the server addresses that the server has, but the client, based on the error committed, does not. The analysis shows also that the addressing scheme has a maximum of two forwards, but that the vast majority of queries usually goes directly to the correct bucket, [LNS96].

In case of a merge, the client might have a view of a file with extent larger than the true extent and therefore send queries to servers with non-existing buckets. Such a query either is not answered at all, or with an error message. In this case, the client resets its image to (0,0), i.e. to an extent of G, resends the query to a bucket calculated with this new file view, and typically receives an IAM that moves its view of the extent closer to the actual file state.

A.3 Scans

A client uses a scan operation to retrieve all records matching a query on the contents of the non-key values. This limits its usefulness for LH*$_{RE}$ almost exclusively to scans looking for key share records. A client initiates a scan using multicast or unicast messages to all buckets it knows of. The servers process the scan locally and deliver the records satisfying the query to the client. If the client uses unicast, then the possible difference between view and actual file state causes certain buckets to be missed. Servers therefore need to check for client view and their view of the file state and forward the scan message to buckets not in the view of the client. An alternative implementation of scan uses forwarding almost always. In this case, the client only sends the scan message to buckets 0, 1, ... , G – 1, i.e. uses an artificial view of (0,0). In more detail, forwarding is based on comparing the view of the file state by the client (included in the scan message) with the bucket level, determine whether the view has the same level l' as the bucket level j and if not, forward the scan message to buckets $s + 2^{l'}G$, $s + 2^{l'+1}G$, ... $s + 2^{j-1}G$. These are all descendents, namely the buckets that have been split recently from the receiving bucket recently. See [LNS96] for an easy algorithm.

During the *collection phase*, every server sends all local records matching the query to the client. There are two possible termination protocols. The *probabilistic termination protocol* has only servers reply to the scan message if they have relevant messages. The client therefore has to use a time-out to determine whether all messages have arrived and has no means to diagnose lost messages. The *deterministic termination protocol* (used for LH*$_{RE}$) requires a reply from every server and allows the client to ascertain that all servers have responded. We achieve the latter by having all responding buckets include their level in the response. This allows the client to obtain the correct file state as a side

effect. If the client misses responses from servers, the client resends, restarts the record, or as a final resort, contacts the coordinator.

A.4 Descendent Sets

We defined the descendent set D_i to be the set of all (number of) buckets that are directly or indirectly split from one of the initial buckets $i \in \{0, 1, ... G–1\}$ and stated a number of properties of these descendent sets. First, we claim that $D_i = \{i + Gj \mid j \in \mathbf{N}\}$, the set of all natural numbers $\equiv i$ modulo G. If $x \in D_i$, then the bucket split from Bucket x has number $x + 2^jG$, where j is the bucket level of Bucket x. Therefore, the bucket numbers of buckets split (directly or indirectly) from Bucket i form a subset of D_i. Since all possible bucket numbers have to make up the set $\mathbf{N}$ of natural numbers and since the D_i, $0 \le i < G$, form a partition of $\mathbf{N}$, it follows that they have to be identical. A merge operation merges a Bucket x with a Bucket $x + 2^jG$, which again implies that merge operations take place within D_i. A scan operation will be send either to Buckets 0, 1, ..., G–1 or to a superset. When scan queries are forwarded, they will only be forwarded to buckets split from the receiving bucket, so that scan operations stay also with sets D_i. Finally, the rules for forwarding RID based queries only allow forwards to descendents of the bucket that they were sent to. Thus, if a given key share record is currently located in a Bucket x with $x \in D_i$, then any file state view by the client has the query sent to Bucket i or to a Bucket y that descended from Bucket i. Thus follow the claims made in the main body of this paper.

Attackers intrude servers, and not buckets. If we allow merges (and not every LH* structure allows for merges because it assumes an always growing file) then servers might see various buckets stored on them. This causes a problem for addressing that can be fixed, but only by changing the LH* addressing operations, but it also would destroy our security assumptions, since the buckets could belong to different descendent sets. To exemplify the problem assume $G = K = 3$ and a file with extent 3, 4, 3, 4, 5, 4, 3, 4, 5, 6, 5, ... A server might obtain Bucket 4 the first time that the file grows, but looses it with the next merge operation. When the file starts growing again, the server hosts Bucket 5. It loses the bucket again, but gets Bucket 6 in the next round of growth. The same server could thus store all three different key shares of a given key. We therefore have to adhere to the LH* scheme literally, only storing a given bucket on a fixed server. LH*$_{RS}$, the scalable high availability variant of LH*, achieves high availability by using erasure correcting codes to generate parity buckets and then allows buckets on an unavailable server to be reconstructed on a spare server. LH*$_{RE}$ security requires special care with these migrating buckets.