2010 18th Annual IEEE/ACM International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems

# Clasas: A Key-Store for the Cloud

Thomas Schwarz, S.J.
Universidad Católica del Uruguay
Montevideo, Uruguay
tschwarz@calprov.org

Darrell D. E. Long
University of California
Santa Cruz, California
darrell@cs.ucsc.edu

***Abstract*—We propose Clasas (from the Castilian "Claves seguras" for "secure keys"), a key-store for distributed storage such in the Cloud. The security of Clasas derives from breaking keys into $K$ shares and storing the key shares at many different sites. This provides both a probabilistic and a deterministic guarantee against an adversary trying to obtain keys. The probabilistic guarantee is based on a combinatorial explosion, which forces an adversary to subvert a very large portion of the storage sites for even a minute chance of obtaining a key. The deterministic guarantee stems from the use of LH* distributed linear hashing. Our use of the LH* addressing rules insures that no two key shares (belonging to the same key) are ever, even in transit, stored at the same site. Consequentially, an adversary has to subvert at least $K$ sites. In addition, even an insider with extensive administrative privileges over many of the sites used for key storage is prevented from obtaining access to any key.**

**Our key-store uses LH* or its scalable availability derivate, LH*$_{\mathrm{RS}}$ to distribute key shares among a varying number of storage sites in a manner transparent to its users. While an adversary faces very high obstacles in obtaining a key, clients or authorized entities acting on their behalf can access keys with a very small number of messages, even if they do not know all sites where key shares are stored. This allows easy sharing of keys, rekeying, and key revocation.**

## I. Introduction

Many applications can benefit from the inexpensive, scalable storage solutions offered by storage providers in the cloud. These applications often need to maintain confidentiality of data. While storage service providers might be trusted to fulfill their contractual obligations, their maintenance and administrative personnel might be suborned and security procedures insufficiently vetted. In this environment, client-side encryption is an attractive choice, but its downside is the difficulty of key management. Loss, destruction, or disclosure of keys during the lifetime of the data can be disastrous.

A third-party escrow service could safeguard keys and provide key recovery on request [1], [2]. These services are not widely used, perhaps because of legal and technical difficulties. Current industrial practices mainly use server-side encryption, unless "data security is paramount" [3], [4]. The resulting lack of control is unacceptable when the owners of the data and of the storage sites are different entities.

Microsoft's Encrypted File System (EFS) uses a public-private key infrastructure [5]. Files are encrypted, the encryption key is itself encrypted with the public key of the application and with a domain (group) server public key and stored with the file. A successful intrusion into two servers reveals all information.

The research prototype Potshards takes a different approach [6]. It targets assets stored much longer than the average lifetime of any encryption system and eschews encryption at all. Instead, participants use secret sharing to break every datum into several shares stored at different systems, each protected by an autonomous authentication procedure. The price tag is high storage overhead, as every share has the same size as the original data.

A large number of proposals for distributed storage systems exist, such as CFS [7], FarSite [8], Pasis [9], and PAST [10] to name only a few. Common to these systems is that data is stored on machines beyond the administrative and often legal domain of the data owners.

We propose a solution to the resulting security problem that combines client side encryption, Potshard's secret splitting across administrative boundaries, and an adaptation of LH* [11], [12] a scalable, distributed data structure. An asset is protected with a secret key that is split into a number of key shares, which are then stored in the LH* data structure that can use all available storage sites. The LH* addressing algorithm prevents shares of the same key from being stored together. It thus forces an attacker to subvert many sites, while allowing fast access to keys for authorized users, who can share keys, retrieve lost keys, and revoke keys.

The combinatorial explosion resulting from the use of LH* and many key shares provides strong probabilistic guarantees. Even an adversary who can access many sites in the system is very unlikely to have access to one of a reasonably large set of keys. The mere fact of secret sharing provides a hard lower bound on the number of sites needed for an adversary to be successful. In contrast, the client itself can quickly retrieve keys. The client can also delegate its rights to an authority. A central authority or a delegated authority can access keys on the client's behalf, can revoke and replace keys. For example, when an employee has left the employment or a mobile device with keys was stolen. Our work here focuses on the security resulting from key share dispersal. We leave out the design of authentication and authorization procedures for a large set of interacting clients.

Because the deterministic security of our scheme is based on LH* and its variants, the next section gives a short overview

This research was supported in part by the National Science Foundation under award CCF-0937938, IIP-0934401, by the Department of Energy's Petascale Data Storage Institute under award DE-FC02-06ER25768, and by a grant from the Stiles Family Foundation to Thomas Schwarz.

1526-7539/10 $26.00 © 2010 IEEE
DOI 10.1109/MASCOTS.2010.35

of these systems that can be skipped at first reading. We then discuss the key store itself. A detailed analysis of its security is given in Section IV, while Section V draws the consequences for the operations of Clasas.

## II. Background

We use the highly available, scalable distributed data structure LH*$_{RS}$, a distributed version of linear hashing [11], [12], [13], [14]. LH*$_{RS}$ adds scalable availability to LH* and an LH*$_{RS}$ file appears to a client as an LH* file. An LH* or LH*$_{RS}$ file is a collection of records identified by a Record IDentifier (RID). One or more clients insert, update, read, and delete records. The records themselves are stored in buckets, each stored at a different server. Buckets have logical addresses 0, 1 ... $N-1$. We call $N$ the file extent. LH* gracefully adjusts $N$ to the total number of records. The location of a record depends therefore both on the RID and $N$. We first break up $N$, writing $N = 2^l + s$ where the level $l$ is the highest power of two smaller or equal to $N$ and $s$ is called the split pointer (for reasons that will become apparent soon). We define $h_i(r) = r \bmod 2^i$. The family $(h_i)_{i=0,1,2,...}$ is a family of extendible hash functions used for the address calculation. The address $a$, i.e. the bucket where a record with RID $r$ resides, depends indirectly on the current extent $N$ and directly on the current level $l$ and split pointer $s$. We obtain $a$ by

$$\text{if } h_l(r) < s \text{ then } a = h_{l+1}(r) \text{ else } a = h_l(r)$$

If a bucket reports an overflow after an insert, then a special coordinator node creates a new bucket with number $N$. The extent is incremented, but only records currently in bucket $s$ change their bucket address because of this change in file extent. Indeed, about half of them should be now located in the new bucket and the split operation moves them there. Conversely, the merge operation is triggered if a bucket reports an underflow and undoes the last split.

The coordinator does not inform clients of splits and merges. Consequentially, a client can commit an addressing error based on an outdated extent. To deal with misdirected requests, all buckets remember the file level when they were last split or created. Based on this information, a bucket receiving a misdirected request can forward the request, usually directly to the correct bucket. It is known [12] that in the absence of an intervening bucket merge at most two forwards can occur. The correct bucket sends its view of the file extent to the requester who updates his view of the file extent and is guaranteed to never repeat this particular error. After merge operations, it is possible that a client tries to access a record in a bucket that no longer exists. After a timeout or receipt of an error-message, the client resends the request to one of the original buckets.

In addition to RID-based operations, we can scan an LH* file. The scan operation is a Map-Reduce type query to all $N$ buckets, requesting records fulfilling the conditions specified in the query. By forwarding, it is guaranteed to reach all buckets, even those the client did not know about.

As a consequence of bucket size limits and the maximum number of message forwards, LH* maintains constant access times independent of the number of records in the file. LH* does not rely on central routing information, broadcasts of states to clients, or Distributed Hash Tables (DHT).

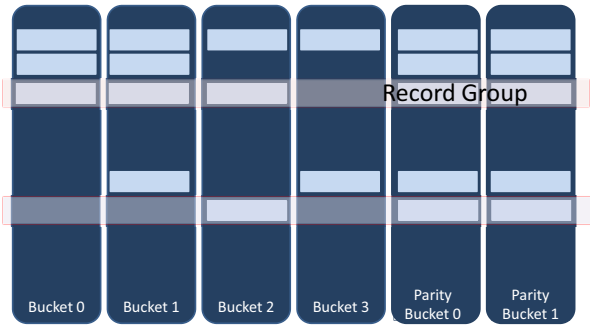


Fig. 1: Formation of record groups for calculating parity records in an LH*$_{RS}$-file

Any large distributed system should offer transparent recovery from site unavailabilities. As we use the LH* structure to store key shares whose loss implies loss to possibly many assets, this need is exacerbated for us. We use LH*$_{RS}$ which groups adjacent buckets into a *bucket (reliability) group* of $k$ buckets. We refer to them as *data* buckets. To each bucket group, it adds one or more *parity buckets*, depending on the size of the file. Records in data buckets are grouped into *record groups* such that each record in a group belongs to one bucket in the bucket group. Figure 1 gives an example with $k = 4$ and two parity buckets. Each parity bucket has a *parity record* in the record group. Its key is the number of the record group, and its payload contains the keys of all data records in the record group, as well as the parity of the data records' payload calculated with a linear erasure correcting code. Not all record groups contain $k$ data buckets. In this case, the parity records are calculated using dummy zero records in place of the missing data records. When a record in a record group changes, the parity records are updated based only on the exclusive-or (delta) between the previous and current version of the changed record and key information. In particular, this process allows us to replace dummy records with real data records. The erasure correcting code is a linear code with optimal erasure correcting capacity. Hence, if there are $p$ parity buckets in a group, then LH*$_{RS}$ can recover from $p$ or less bucket failures. LH*$_{RS}$ uses its particular growth by splits to increase $p$ gradually and thus adjust to the greater need for redundancy in the storage as the number of buckets increases.

## III. Clasas Operations

Clasas (Claves Seguras) uses an (adapted) LH*$_{RS}$-file to store cryptographic keys of clients in a distributed system in the cloud. For protection, keys are broken up by a standard cryptographic secret sharing mechanism [15] into $K$ shares, which are stored at different sites. The storage nodes also store buckets from an LH*$_{RS}$-file, that contains the keys of all clients.

### A. Adjustments to LH*$_{RS}$

To use LH*$_{RS}$ in our setting, we need to make a number of relatively minor adjustments. First, we never allow the number of data buckets to fall below $K$, the number of shares into which we break each share. Otherwise, at least two key shares would be located in the same bucket. Secondly, instead of adjusting the file extent to the number of records stored in the file, we adjust the file extent to be approximately the number of sites available in the storage system. This departure from LH* is motivated by a change in goals. LH* intends to keep buckets close to their capacity to avoid deterioration of access to records in overflowing buckets. We instead are interested in dispersing key information as much as possible. Third, we assume that access to buckets is authenticated and authorized on a per bucket base. We do not discuss the selection of the mechanisms, but argue that maintaining public-private keys per client and site is less arduous than maintaining multiple keys per client. As records in an LH*$_{RS}$-file change location, a client might have to use different authentication methods to access the same record over the lifetime of the record. Also, when a client approaches the wrong bucket for a key share, and the request is forwarded, the final site needs to authenticate the client independently.

Finally, we have to make a change to the high availability implementation of LH*$_{RS}$. If there is a single data record in a record group (as in the second highlighted record group in Figure 1) then the contents of the parity records reveal the contents of the data record. Similarly, a parity record reveals which of the data records in the record group are dummy (zero) records. To avoid this situation, we do not allow bucket groups with only one data bucket. Various mechanisms can be employed, such as having a larger final bucket group. In addition, if we create a record group by inserting a data record that cannot be placed in any other record group, we create dummy records with random contents in all other slots in a record group. If later we want to use this slot for a real key share record, LH*$_{RS}$ performs a normal record update.

### B. Key Share Records

Clients maintain their keys locally in a key chain. Before a client stores a key $C$ in the key chain, it also backs it up in Clasas. Since keys are small, we can use a simple scheme that is not storage optimal. We first generate (cryptographically strong) random numbers $C_0, \ldots C_{K-2}$ and calculate $C_{K-1} = C_0 \oplus \ldots \oplus C_{K-2} \oplus C$ as the last key share. Each key share becomes then the payload of an LH* record, together with fields for authentication and authorization purposes. We also include a field that identifies the client or clients using this key and that points to the location of the key in the key chain of the client(s). Finally, we need to give a unique Record IDentifier (RID) to the key share record. In addition to uniqueness, the RID should not identify the client. Most importantly, the LH* addressing mechanism should place all key shares into the $K$ different data buckets if the extent were $K$. This is the rule that in conjunction with LH* addressing implies the key safety property: No two key shares will ever be placed together in the same bucket however the system evolves. A simple trial and error method based on a cryptographically strong random number generator will yield RIDs with these properties.

### C. Operations

Clients can *recover* keys by using the LH* RID-based access if they can regenerate the RIDs used for the key shares. Alternatively and more likely, they can use the LH* scan operation by searching for all key shares belonging to the client and possibly having a given pointer into the key chain. Similarly, an entity with the correct authentication information can use a scan to recover all key shares belonging to a certain client and thus recover all of the client's keys. In order to revoke a client's keys (e.g. because an employment was terminated or because an application was replaced,) the authority can recover all keys, then search for and access all assets of the client and rekey them.

## IV. Security Analysis

### A. Threat Model

The novelty of our scheme lies in its distributed nature and our threat model focuses on this aspect. In particular, we do not consider direct attacks on a client but concentrate on attacks at storage sites. We focus also on key integrity and confidentiality, and only consider availability briefly in what follows. The greatest threat comes from insiders administering servers in the cloud. These adversaries are likely to have access to more than a single server. They can certainly destroy locally stored assets and the same applies to key shares. We assume that social sanction will prevent them from doing so. For our analysis, we assume individual attacks on servers, but discuss generic strategies against insiders with administrative control over many sites. Finally, we assume that the encryption method used is strong and an adversary needs access to the key as well as the asset.

### B. Single Key Safety

An adversary can only obtain key shares from an intrusion into a server. These key shares can be residing at the server or they can be in transit, for example because the client made an addressing error during the insert of a key share. Recall that we split every key into $K$ shares. Clasas is $(K-1)$-safe, which means that an adversary needs to have access to at least $K$ servers in order to obtain a share. The formal proof of $(K-1)$-safety requires some notation. When a bucket $i$ splits, then the new bucket $j$ (which happens to have address $j = 2^l + i$ where $l$ is the level) is called a descendent. Reversely, we call $i$ an ancestor of $j$. We form a set $D_i$ of all direct or indirect descendents (descendents, descendents of descendents, ...) of a bucket $i$, $0 \leq i < K$ and call $D_i$ the $i^{\text{th}}$ *descendency set*. LH* addressing implies the following properties for files with extent at least $K$:

1) The sets $D_i$, $0 \leq i < K$ are mutually disjoint and every possible bucket is in one $D_i$.
2) A descendent of a bucket in $D_i$ is also in $D_i$.

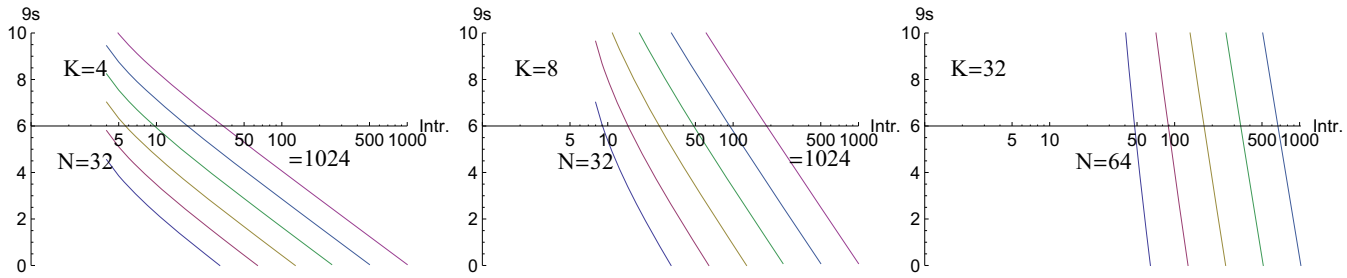


Fig. 2: Nines of Assurance that a single key is not compromised when broken into $K = 4$ (left), $K = 8$, and $K = 32$ (right) shares. The $x$-axis gives the number of randomly intruded sites. The curves correspond to a total of 32, 64, 128, 256, 512, 1024 (left and middle) and 64, 128, 256, 512, and 1024 sites.

3) A RID based query (i.e. a key share insert, update, delete, or read) to a record in a bucket in $D_i$ is originally sent and possibly forwarded only through and to other buckets in $D_i$. In particular, a scan for key shares transmitting its results back to the client uses independent messages for each key share belonging to the same record.
4) If a scan requests reaches a bucket in $D_i$ and the message is forwarded because the client did not know the true extent of the file, then the message only has passed and will pass to other buckets in $D_i$.
5) Records in a bucket in $D_i$ can migrate through split and merge operations, but only to other buckets in $D_i$.

In an LH* file that contracts and expands repeatedly, it could happen that the same server stores various buckets in succession. We add a rule that prohibits this successive hosting of buckets on the same server. Taken together, all these rules imply that key shares are never collocated on the same server, even in transit. They also allow protection against insiders with administrative control over many servers, as long as we can insure that no single administrative domain has direct or indirect control over buckets from all dependency sets.

An LH*$_{\mathrm{RS}}$ gives an attacker an additional opportunity for harvesting key shares as access to sufficient parity buckets gives access to data. Assume a bucket group with $m$ data buckets and $k$ parity buckets. An adversary can gain access to all the data in the group by gaining access to any $m$ or more buckets in the group. According to what we just derived, there are at most $m$ keys ever stored in the bucket group. Using the loss resilience of LH*$_{\mathrm{RS}}$, the attacker has to access at least as many buckets as there are keys.

### C. Assurance

While the safety level $K-1$ ($K$ being the number of key shares into which a key is broken) gives a simple measure of confidence that an intruder cannot read a client's data, it gives only a lower bound on the number of intrusions necessary for an adversary to succeed. Typically, the required number of intrusions is much higher. As usual [16], we define *assurance* to be the probability that an adversary is not successful despite certain gains, namely access to the key shares in $x$ buckets. As sites storing keys also store assets encrypted with these keys, we can calculate *disclosure*, which is the expected ratio of assets to which an intruder now has access.

*1) Single Key:* We first calculate assurance of a single key for a key store depending on the number of sites that the adversary can read. We first assume that there are no parity buckets and that the adversary has somehow managed to intrude $x$ of the $N$ buckets in which key shares are stored. In our scenario, the intruder does not exploit any knowledge about location of buckets and the buckets obtained appear to him to be random. We determine the probability that an adversary has obtained all key shares by a counting argument. There are $\binom{N}{x}$ 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 not containing a key share. Thus, the probability that the adversary obtains a given set of $K$ key shares with $x$ intrusions is

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

The assurance against disclosure of this single key is $q_1(N,x,K) = 1 - p_1(N,x,K)$. As can be expected, the assurance is rather high, even if a large portion of sites has suffered an intrusion. We measure assurance in the number of nines. Mathematically, this is $-\log_{10}(p_1)$. We give an example in Figure 2. There, we display the number of nines of assurance for $N = 32$, 64, 128, 256, 512, and 1024 sites and $K = 4$, $K = 8$ and $K = 32$. We move the $x$-axis to cross the $y$-axis at a point corresponding to six nines. With only $K = 4$ key shares, this level of assurance is only reachable if the LH* file has more than 64 buckets.

For example, nine out of 32 sites intruded still give us six nines of assurance (probability 0.999 999) that the attacker has not obtained all $K = 8$ key shares. The almost even spacing of the curves on the logarithmic $x$-axis shows that the fraction of sites needed for a key disclosure is almost independent of the number of sites. We now investigate this fact. For a given assurance level $a$, we define $x_0 = \min\{x | p_1(x) \geq a\}$ and $F = x_0/N$. $F$ thus measures the portion of sites an adversary has to subvert in order to bring assurance below a certain level. We display $F$ for an assurance of six nines (0.999 999) for various $K$ in dependence on $N$ and confirm that $F$ essentially becomes constant in Figure 3. The percentage

values correspond to the value for $N = 1000$. The convergence of $F$ is easily explained by comparing with an alternative scheme in which key shares are stored in randomly chosen buckets. In the alternative scheme, two key shares can be in the same bucket. Let $\rho$ denote the probability of disclosure of an individual bucket. The assurance against key disclosure in the alternative scheme is simply $1-\rho^K$ and is independent of the number of sites. As the number of buckets increases, the probability of two key shares being in the same bucket goes quickly to zero and the assurance of Clasas converges from above to the assurance of the alternative. We can calculate the critical value for $\rho$ below which assurance falls under a given assurance level. $F$ then converges towards this critical value for $\rho$. For example, the critical value for $\rho$ is 3.16% for $K = 4$, 42.16% for $K = 16$ and 80.58% for $K = 64$, very close to the values in Figure 3. Overall, the dispersal of key shares even over a moderate number of sites is astonishingly effective. The assurance becomes independent of LH* addressing as the number of sites increases.

*2) Multiple Keys:* Assume now that the client has $r$ keys stored in the key store. The probabilities of obtaining two different keys from the data in the same $x$ intruded sites are independent. Consequentially, the assurance is now $q_r(N,x,K) = (1-p_1(N,x,K))^r$. The number of nines of assurance is $\text{nines}_r(N,x,K) \approx \text{nines}_1(N,x,K) - \log_{10}(r)$. We illustrate the influence of $r$ in Figure 4 with the case $K = 8$ and $N = 128$. As the number $r$ of keys increases, the number of sites that an adversary can intrude without assurance falling too low converges only very slowly to $K-1$, and in fact so slowly that clients can easily entrust hundreds of keys to a large storage system with high assurance that not a single one falls into the hands of the adversary.

*3) Single Keys in a LH*$^*_{\text{RS}}$*-File:* A closed form expression for assurance of a single key stored in a LH*$_{\text{RS}}$-file has escaped us. If an adversary breaks into $x$ buckets in a LH*$_{\text{RS}}$ reliability group (consisting of $m$ data buckets and a total of $n$ buckets), she has all of the information contained in the $m$ data buckets if $x \geq m$. If $x < m$, then information in parity buckets is not useful to her.

For our model, we assume that a given site is vulnerable to an adversary with probability $\rho$. To obtain a given key share, the attacker either has to have gained access to the bucket that contained the key share, or to $m$ out of the $n-1$ other buckets, or to both. If $\mathscr{B}(x,N,p)$ is the cumulative distribution of the binomial distribution, i.e. if $\mathscr{B}(x,N,p) = \sum_{v=0}^{x} \binom{N}{v} p^v (1-p)^{N-v}$, then the probability of the second mode is $\mathscr{B}(m,n-1,p)$ and the probability of obtaining a given key share is accordingly

$$\begin{aligned} p_{\text{share}}(\rho) &= \rho + (1-\mathscr{B}(m,n-1,\rho) - \\ &\quad \rho \cdot (1-\mathscr{B}(m,n-1,\rho) \end{aligned}$$

If the number of sites is large enough, then the probability of two records placed in the same bucket is negligibly small. This allows us to approximate the probability of obtaining a given key by

$$p_{\text{key}}(\rho) = p_{\text{share}}(\rho)^K$$

which is a lower bound for the true value. Figure 5 displays assurance, which in this model is independent of the total number of buckets. We notice that the effect of more data buckets in a bucket group is negligible, while of course the effect of key dispersion is rather dramatic.

To gauge the effect of more parity buckets, we can vary the number of parity buckets and with it the tolerance of the file. In Figure 6, we use $m = 8$ data buckets in a bucket reliability group and add parity buckets in pairs of two to the group. As we can see, assurance falls with larger numbers of parity buckets (as our models adds more opportunities for the adversary), but the change is rather benign.

*4) Using Ramp Scheme Sharing for Key Availability:* An alternative to using LH*$_{\text{RS}}$ is direct protection of key shares through a ramp scheme [17]. A ramp scheme splits a secret into $K$ shares of which any $L$ are necessary and sufficient to reconstruct the original key. Since keys are small, we can afford making the shares as big as the original keys. We split a key into $L$ shares (of the same size as the key). We then add $K-L$ parity shares calculated using an optimal, linear erasure code such as a Reed Solomon code. The properties of the code guarantee that any $L$ shares, whether they are original or parity shares, are necessary and sufficient to recover the secret. If an adversary has $L-1$ of the shares, there are exactly as many possibilities for any given additional share as there are keys. Therefore, the adversary cannot derive any information on the key from his knowledge of only $L-1$ shares.

Expanding on the argument for the safety of a single key, we obtain the probability of the adversary gaining access to a single key as the sum of the probabilities of gaining access to $s$ of the $K$ keys, summing from $L$ to $K$. Thus

$$p_{L,K}(N,x) = \sum_{s=L}^{K} \frac{\binom{K}{s}\binom{N-K}{x-s}}{\binom{N}{x}}$$

Figure 7 shows some values. As is to be expected, the assurance is about the same as for LH*$_{\text{RS}}$. The advantage of choosing a ramp scheme is the simplicity of the organization, but unlike LH*$_{\text{RS}}$, there is no native mode of adjusting the availability level $K-L$ to the number of sites.

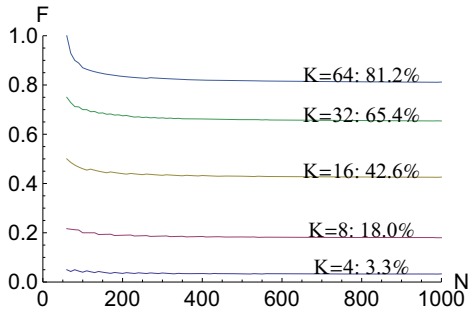

Fig. 3: Ratio $F$ of critical value $x_0$ over number of storage sites $N$ for an assurance of six nines.

### D. Disclosure Size

The *disclosure d* measures the quantity of data revealed by a successful intrusion. More precisely, we define $d$ to be the

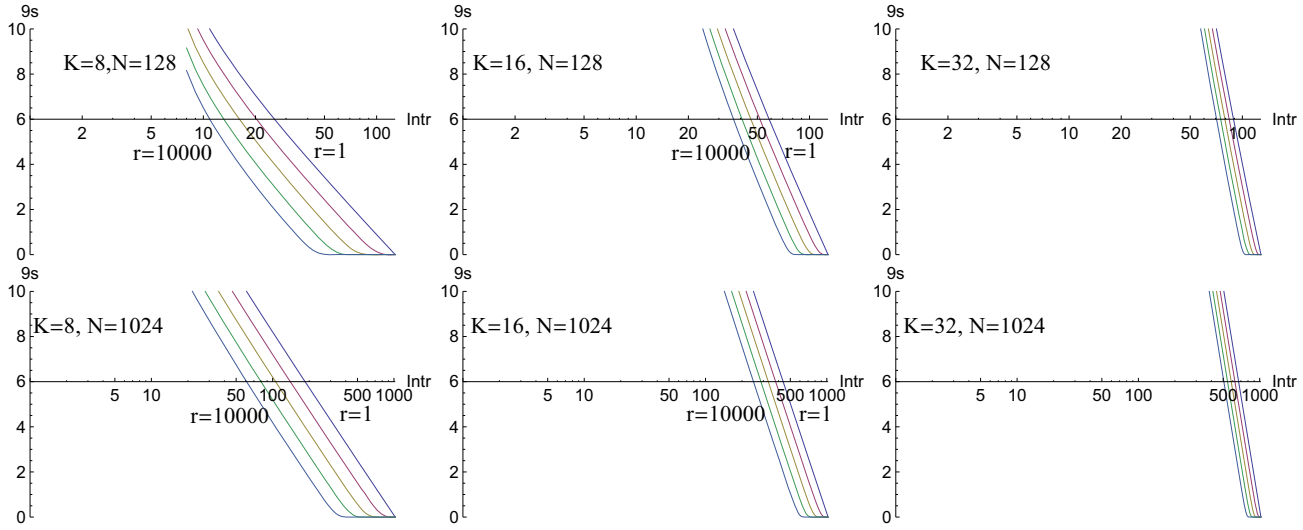


Fig. 4: Nines of Assurance that $r$ keys ($r = 1, 10, 100, 100, 10000$) are not compromised when broken into $K = 8$ (left), $K = 16$ (middle) and $K = 32$ (bottom) shares. The $x$-axis gives the number of randomly intruded sites out of a total of $N = 128$ (top) and $N = 1024$ sites.

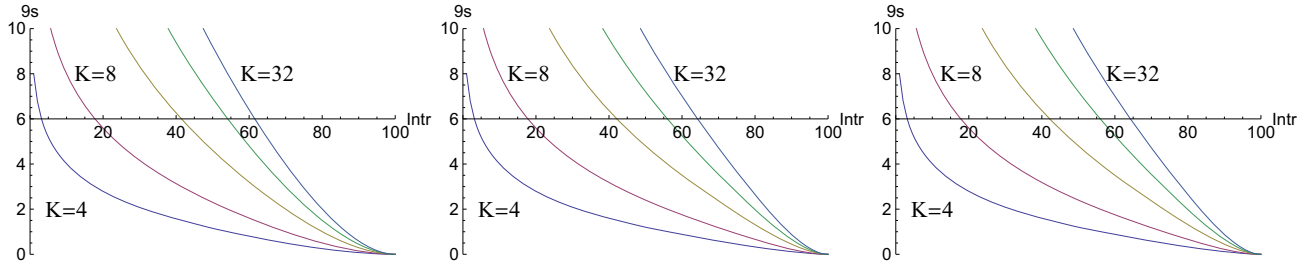


Fig. 5: Nines of assurance of the safety of a single key in LH*$_{\mathrm{RS}}$ file with bucket reliability groups consisting of $m$ data buckets and $n$ buckets total for $m = 4, n = 6$ (left), $m = 8, n = 10$ (middle), and $m = 16, n = 18$. All schemes can tolerate two inaccessible buckets per bucket group.

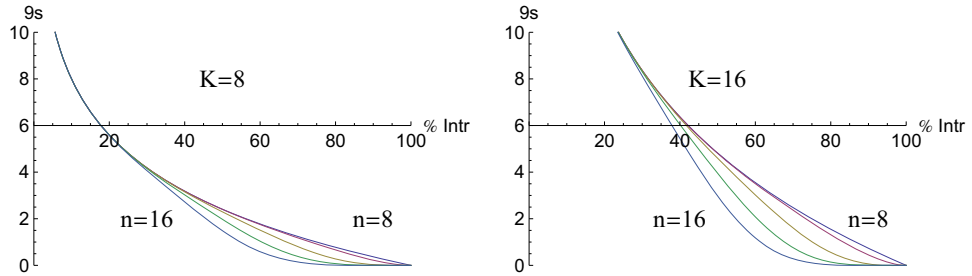


Fig. 6: Nines of assurance of the safety of a single key in LH*$_{\mathrm{RS}}$ file with bucket reliability groups consisting of $m = 8$ data buckets and $n = 8, 10, 12, 14$, and 16 total buckets for a key broken into $K = 8$ (left) and $K = 16$ shares

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. Assume an attacker has intruded into $x$ servers out of $N$ total. This gives the adversary access to about $x/N$ of all encrypted assets, which does her no good without the corresponding keys. She also now possesses a given key with probability

$$p_1 = \binom{N-K}{x-K} / \binom{N}{x}$$

On average, she has obtained a proportion of

$$d(N,x,K) = \frac{\binom{N-K}{x-K} \cdot x}{\binom{N}{x} \cdot N}$$

of all assets. Since this is a proportion, the same expression not only gives the disclosure for a single key but also the disclosure for any number of encryption keys. In particular, *expected disclosure is independent of the number of encryption keys used.* Figure 8 shows that disclosure even for rather substantial break-ins is rather low. We placed the $x$-axis at a level representing $10^{-6}$ disclosure, i.e. 0.000 1 %. Results

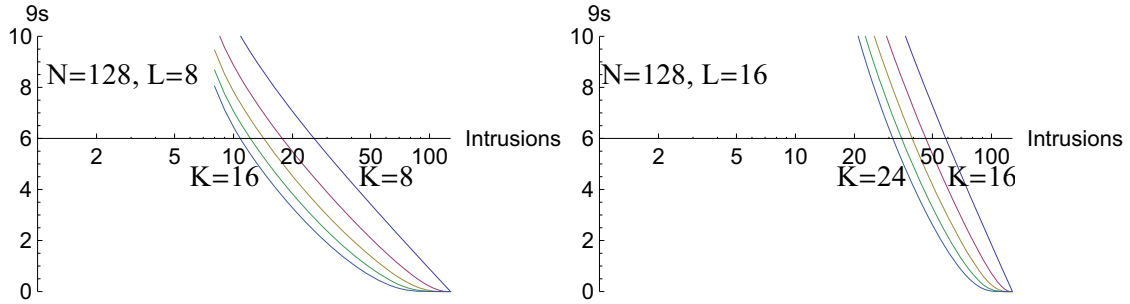


Fig. 7: Nines of assurance against single key disclosure when using a ramp scheme. We use $N = 128$ sites, $L = 8$ (top) and $L = 16$ (bottom) necessary key shares out of $K$ varying from $L$ to $L+8$.

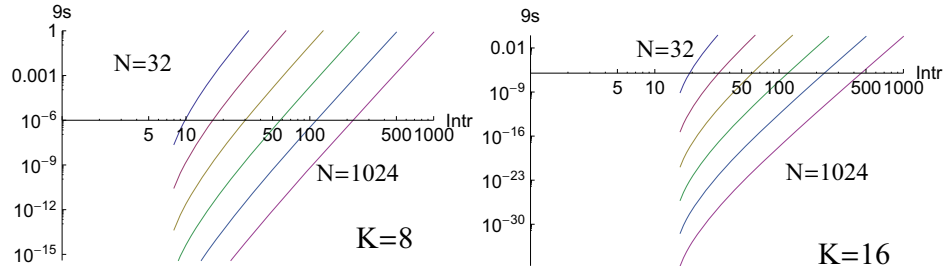


Fig. 8: Disclosure amount with $K = 8$ (left) and $K = 16$ (right), and $N = 32, 64, 128, 256$, 512, and 1024.

for disclosure and assurance closely mimic each other as the respective formulae show.

If we increase the number of keys, but maintain the same total amount of data, the amount of data disclosed in a successful breach becomes smaller. We can capture this notion in what we call *conditional disclosure*. Conditional disclosure resulting from $x$ intrusions is defined to be the disclosure (i.e. the proportion of accessible assets over total assets) under the assumption that the $x$ intrusions resulted in a successful attack, i.e. one where the attacker has obtained at least one key. The probability for a successful attack on a specific key is $p_1 = \binom{N-K}{x-K}/\binom{N}{K}$. Our model implies that obtaining one specific key and obtaining a different one (in the course of the same attack) are independent events. 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 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

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

This is the mean of a binomial distribution divided by P and thus evaluates to

$$E = \frac{r \cdot p_1}{P}.$$

The proportion of assets encrypted with $s$ out of $r$ keys is $s/r$. Consequentially, $E/r$ of all assets are expected to be encrypted with a key that the adversary obtained in the course of the attack. The attack also gained access to encrypted assets themselves, at a proportion of $x/N$. Thus, the conditional disclosure is

$$d_{\text{cond}} = \frac{p_1 \cdot x}{PN}$$

We present a contour graph of conditional disclosures in Figure 9. By increasing the number of keys, we can control conditional disclosure rather well. Increasing the number of keys decreases the expected damage of a successful attack, but also increases the probability of such a successful attack. Ultimately, the choice is a business decision. A minimal number of key limits the chance of a successful intrusion with all of its associated costs in public relations, regulatory fines, and need for compensation for users, but such a failure would probably have catastrophic costs. A larger number of keys increases the probability of a break-in, but limits its expected size. Such a scenario can possibly be addressed by insurance, which traditionally is hard to obtain for protection against very rare, but extremely costly events.

### *E. A Refinement of the Intrusion Scenario*

Our basic intrusion scenario limits the capability of an outside attacker perhaps more than is reasonable to assume. We switch now to a different scenario where the attacker spends a certain effort in order to obtain contents of a given storage site. However, we now allow the adversary knowledge of the LH* structure of the file, in particular of the global state of the LH* file and the locations of buckets. These can be obtained after obtaining access to a client, or through an analysis of the traffic, even if the traffic is encrypted. The adversary now has obtained some advantages: First, if LH*$_{\text{RS}}$ is deployed, the adversary knows which buckets are parity buckets and which ones data buckets. He can then plan on

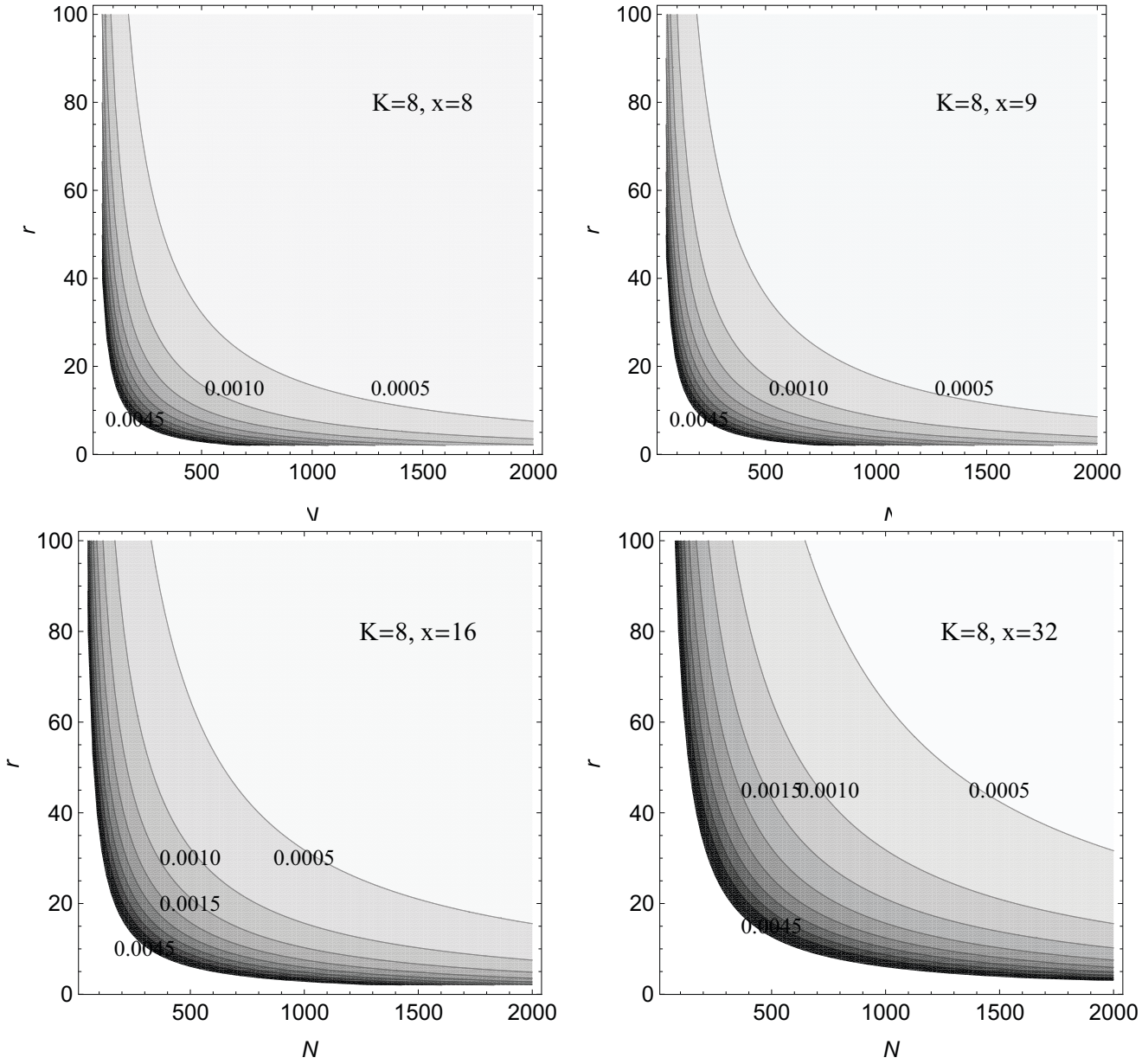


Fig. 9: Contour Graph for the conditional disclosure. We vary $N$, the number of sites, on the $x$-axis and $r$, the number of keys. We chose the number of key shares to be $K = 8$ and we consider $x = 8$, $x = 9$, $x = 16$, and $x = 32$ (right) intrusions. The upper right corner of each graph has close to zero conditional disclosure.

attacking only data buckets, as we now assume. Secondly, by reconstructing descendency sets, the attacker can avoid hunting for a key share that she already obtained. Thirdly, the number of keys in a bucket is only approximately even, if the LH* file extent is a power of two (and the split pointer reset to zero). Otherwise, there are some buckets already split (e.g. Buckets 0, and 1 in Figure 10, which shows the state of an LH* file with 10 buckets) or obtained through splitting (e.g. Buckets 8 and 9) and buckets not yet split (Buckets 2, 3, 4, 5, 6, 7), which contain about twice as many records as the other buckets. The attacker might also have different costs of a break-in, such as exposure when using social engineering, the time spent testing for vulnerabilities, ... The two different sizes of the buckets, the different costs of breaking into them, and their position in descendency sets lead to an interesting optimization problem for the attacker.

We use the file in Figure 10 as an example, even though it uses a very low value of $K = 3$. Assume that the attacker plans on $x$ simultaneous intrusions, of which $x_0$ are directed to buckets in $D_0$, $x_1$ to $D_1$, and $x_2$ to $D_2$. To have any chance of success, all $x_i$ need to be non-zero. We assume that the costs of all attacks are identical and that the adversary looks for

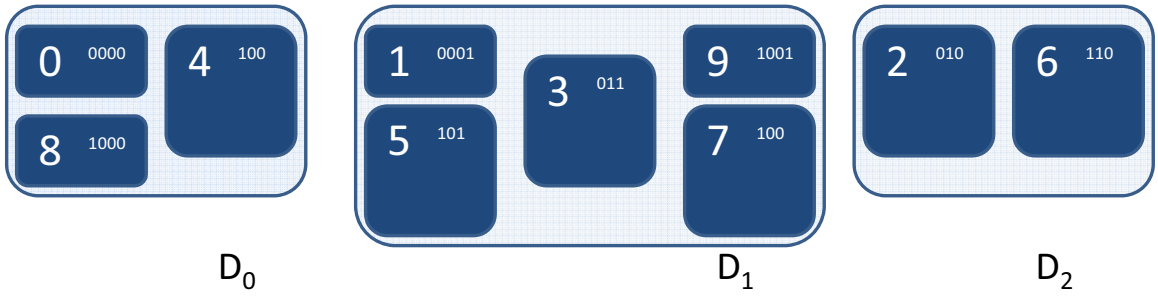


Fig. 10: LH* file with extent 10 broken into descendency sets for buckets 0, 1, and 2.

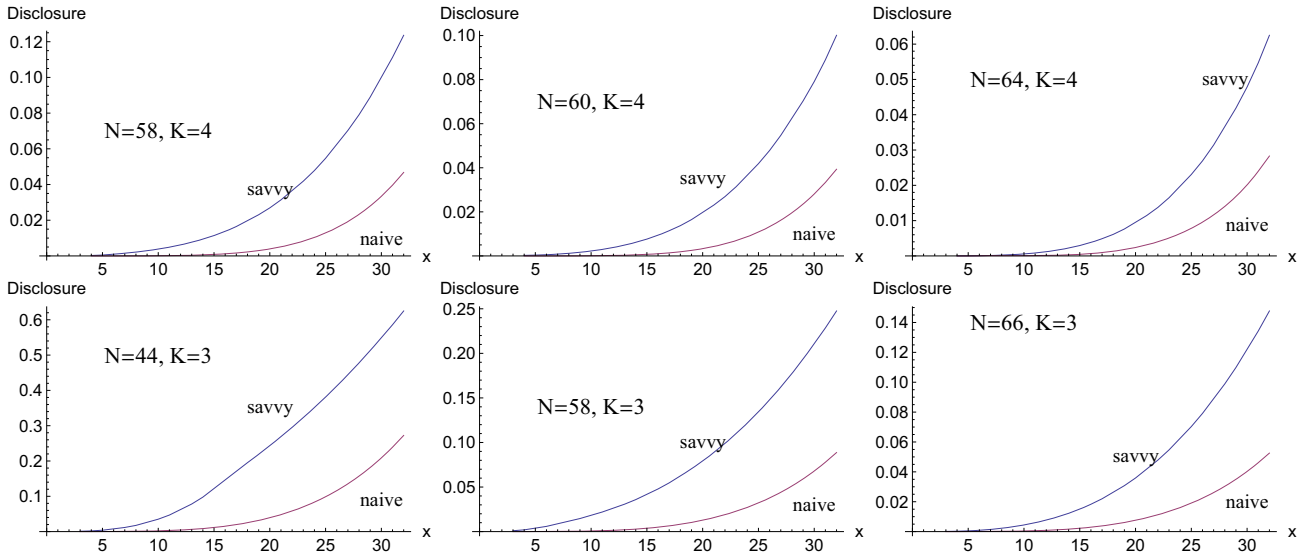


Fig. 11: Disclosure with $K = 4$ and $N = 58$, $N = 60$ and $N = 64$ and $K = 3$ and $N = 44$, $N = 58$, and $N = 66$ for the savvy and the naïve attacker.

a single key. Within her "attack budget" of $x_i$ the adversary does best to prefer large buckets over small ones. With this strategy, we can calculate the chances $p_i(x_i)$ of obtaining the key share in $D_i$ from $x_i$ and the number of big and small buckets in $D_i$. Assume that there are $b_i$ big buckets and $a_i$ small buckets in $D_i$. Then $p_i(x_i) = 2x_i/(a_i + b_i)$ if $x_i \leq b$ and $p_i(x_i) = (b_i + x_i)/(a_i + b_i)$ if $x_i > b_i$. The adversary has to maximize $\prod_{i=0}^{K-1} p_i(x_i)$ subject to $\sum_{i=0}^{K-1} x_i = x$.

We calculated the disclosure for a "savvy" attacker who uses this optimization and an "naïve" attacker who does not. The results are depicted in Figure 11 (note the different scales for the $y$-axis). While the success of the savvy strategy is clearly visible, it is not dramatic. The biggest differences we found result from using values that are not powers of two for $K$ and for values of $N$ midway between powers of two. Our numbers are based on an explicit optimization, but since we did not use sophisticated approximation such as simulated annealing, we only obtained values for the "savvy" adversary for smaller $K$. We conclude that location information is indeed helpful to the adversary, but it does not amount to a serious threat on the scheme.

## V. Operational Considerations

In most storage systems in the cloud, several storage sites belong to the same administrative unit and are vulnerable to insider attack not individually, but in ensemble. When implementing Clasas in such an environment, the insider threat requires careful assignment of buckets to the different entities. Placing data buckets from all descendancy sets into the same administrative domain gives an insider adversary a non-zero chance of obtaining some keys (and hence assets). Placing sufficient buckets (i.e. as many as there are data buckets) from a bucket reliability group into a single domain gives the malicious insider also access to the other data buckets. Unfortunately for us, the creation of bucket reliability groups in LH*$_{RS}$ places buckets from different descendancy sets into the same group. As the assignment of buckets to servers falls to a single entity, the *coordinator*, the solution requires that the coordinator is made aware of administrative boundaries. Even if there are only two administrative domains, the coordinator can assure that for a given $K$, no domain has all of the information needed to retrieve a single key.

Assets are not created equal and neither are the security requirements of a client. Using the simple share generation scheme allows clients to dynamically change the number of key shares for a key. The process is similar to the bucket split. Assume that $K = 2^l + s$ where $l$ is the level of an LH* structure of extent $K$. The client retrieves the key share $C_s$ in $D_s$, which is the key share that would be stored in bucket $s$ in an LH* file of extent $K$. It then creates a random number which becomes key share $C_K$ and calculates $C_s = C_s \oplus C_K$. The client creates a new key share record by copying the authentication information and storing $C_K$ as the payload. It gives the new record a RID that places it into bucket $K$ if the file extent were $K+1$. It reuses the old RID for $C_s$ but places the new $C_s$ in it instead of the old one (if keys are shared, this alone alerts the other clients that the safety level of the key has changed). The number of key shares is now $K+1$. A repetition of the process allows any number of key shares and a reverse operation lowers the number of key shares.

A different variant doubles the access time to key share records in order to almost double the safety of a key by using a master key to encrypt the key shares of a single key or of a group of keys. The master key itself is stored in Clasas, possibly using a larger number of key shares. If the master key is cached at the client, the access times to key shares remains the same.

## VI. Conclusions

Clasas has not been implemented. An implementation for a specific environment would force complete solutions of authorization and authentication and ultimately prove its feasibility.

LH*$_{\mathrm{RE}}$ [18] is a version of linear hashing where all records are encrypted. It stores keys in the same manner as Clasas, but in the LH* structure itself. The security analysis of Clasas, while much more involved than the one presented in [18], also applies to LH*$_{\mathrm{RE}}$. While LH*$_{\mathrm{RE}}$ is an integration of an SDDS with the key store that we presented here, other scalable data structures could also benefit from integrating a key store with the data structure. BigTable [19] and Chord [20] come to mind. While Chord has also disjoint descendency sets, its forwarding algorithm does not prevent different key shares from traversing the same node. BigTable has a meaningful RID whose privacy would need to be protected.

In conclusion, our work here proposed a new type of key store that relies for its security on the splitting of keys into shares and using an LH* structure to disperse the shares in a large distributed system. With the advent of cloud computing, such large distributed systems become possible even for organizations of moderate size. We have shown that key share dispersion is a very powerful tool to thwart an adversary who can only gain access to a number of the total machines involved in Clasas. This type of security depends on the security of the clients themselves and the use of individual access and authentication mechanisms between clients and individual storage sites. Clasas offers a solution to a very difficult problem (how to maintain a large number of keys) based on solving a more tractable problem. While Clasas denies access to keys to an attacker, it allows fast access for clients and authorized entities. Overall, Clasas enables safe and fast client side encryption for clients of cloud storage.

## References


[1] M. Bishop, *Computer Security*. Addison-Wesley, 2003.

[2] PGP, "Method and apparatus for reconstituting an encryption key based on multiple user responses," U.S. Patent 6 662 299, 2004.

[3] "The key to controlling data," White Paper, Sun Microsystems Alliance Technology Group, January 2008.

[4] "Building dispersed storage technology," White Paper, Cleversafe Open-Source Community, www.cleversafe.org.

[5] Using encrypting file system. [Online]. Available: technet.microsoft.com

[6] M. W. Storer, K. M. Greenan, E. L. Miller, and K. Voruganti, "Potshards—a secure, recoverable, long-term archival storage system," *Trans. Storage*, vol. 5, no. 2, pp. 1–35, 2009.

[7] F. Dabek, M. F. Kaashoek, D. Karger, R. Morris, and I. Stoica, "Wide-area cooperative storage with CFS," in *Proceedings 18th Symposium on Operating System Principles*, 2001, pp. 202–215.

[8] A. Adya, W. J. Bolosky, M. Castro, G. Cermak, R. Chaiken, J. R. Douceur, J. Howell, J. R. Lorch, M. Theimer, and R. P. Wattenhofer, "Farsite: federated, available, and reliable storage for an incompletely trusted environment," *SIGOPS Oper. Syst. Rev.*, vol. 36, no. SI, pp. 1–14, 2002.

[9] J. Wylie, G. Goodson, G. Ganger, and M. Reiter, "A protocol family approach to survivable storage infrastructures," in *Proceedings, 2nd Berinoro Workshop on Future Directions in Distributed Computing*, 2004.

[10] A. Rwostron and P. Druschel, "Storage management and caching in PAST, a large-scale, persistent peer-to-peer storage utility," in *Proceedings 18th Symposium on Operating System Principles*, 2001.

[11] W. Litwin, R. Moussa, and T. Schwarz, "LH*RS—a highly-available scalable distributed data structure," *ACM Trans. Database Syst.*, vol. 30, no. 3, pp. 769–811, 2005.

[12] W. Litwin, M.-A. Neimat, and D. A. Schneider, "LH*—a scalable, distributed data structure," *ACM Trans. Database Syst.*, vol. 21, no. 4, pp. 480–525, 1996.

[13] ——, "LH*: Linear hashing for distributed files," *SIGMOD Rec.*, vol. 22, no. 2, pp. 327–336, 1993.

[14] W. Litwin, H. Yakoubin, and T. Schwarz, "LH*RS,P2P: A scalable distributed data structure for P2P environment," in *NOTERE*, 2008.

[15] A. Shamir, "How to share a secret," *Commun. ACM*, vol. 22, no. 11, pp. 612–613, 1979.

[16] A. Mei, L. V. Mancini, and S. Jajodia, "Secure dynamic fragment and replica allocation in large-scale distributed file systems," *IEEE Transactions on Parallel and Distributed Systems*, vol. 14, pp. 885–896, 2003.

[17] G. R. Blakley and C. Meadows, "Security of ramp schemes," in *Proceedings of CRYPTO 84 on Advances in Cryptology*. LNCS 196, Springer, 1985, pp. 242–268.

[18] S. Jajodia, W. Litwin, and T. Schwarz, "LH*RE: A scalable distributed data structure with recoverable encryption," in *International Conference on Cloud Computing (CLOUD)*, 2010.

[19] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber, "Bigtable: A distributed storage system for structured data," *ACM Trans. Comput. Syst.*, vol. 26, no. 2, pp. 1–26, 2008.

[20] I. Stoica, R. Morris, D. Liben-Nowell, D. Karger, M. Kaashoek, F. Dabek, and H. Balakrishan, "Chord: A scalable peer-to-peer lookup service for internet application," in *SIGCOMM*, 2001.