Recoverable Encryption Through Noised Secret

(Electronic[1] Res. Rep. 2012-2-18)

Sushil Jajodia[2], W. Litwin[3] & Th. Schwarz[4]

Wednesday, July 27, 2011, Revised Feb. 25, 2012

Abstract. Encryption key safety is the Achilles' heel of modern cryptography. Simple backup copies offset the risk of key loss, but increase the danger of disclosure, including at the escrow's site. *Recoverable Encryption* (RE) alleviates the dilemma. The backup is specifically encrypted so that recovery by brute force, i.e., without any hint from the key owner, remains possible. We propose RE using a *noised secret*, RE_{NS} in short. To dissuade indelicate attempts, the owner sets up the decryption complexity so that recovery time at the escrow's facility, a computer or perhaps a cluster, becomes inhibitive. The actual recovery fits the time desired by the requestor, using distributed processing over a large cloud. A 10K-node cloud may suffice to recover the key in ten minutes, for the dissuasive set up of seventy days for a single computer. Large public clouds are now available with acceptable price tags. Their sheer size makes the illegal use unlikely. We show feasibility of RE_{NS} schemes possibly of interest to numerous applications.

1. Introduction

Most users want to preserve the data confidentiality. They use high quality symmetric or asymmetric encryption, but thereby incur the dangers of key loss. Usually this happens because of the owner's laziness to do a backup or because of the owner's computer misfortune. On the other tune, a company could face a data loss because of an employee departed with the keys. Health data are often encrypted, but access to may remain crucial, even if the owner is unable to help. Encrypted family's data may need to survive the owner's disappearance with the keys, e.g., while boating in SF Bay. A key may alternatively be a trusted ID, e.g., a Diffie-Helman one, the loss of that being potentially also catastrophic…

A usual approach is to have a backup copy with some escrow (service) or Admin or in a cloud [JLS10]. The Escrow or Admin or the owner simply retrieves the backup when needed. However, even a trusted escrow might reveal malicious or intruded by an adversary. A cloud can be unsafe against intruders. If an illegal key usurper discloses it, a big trouble may result. Consequently, users usually seem to refrain from escrow services. On the other end, there is the fear of the definitive encrypted data loss. All these dilemmas make many rather not using any encryption [MLFR2]. In turn, this is also an invitation for trouble. Compound fear deters in particular many from the data outsourcing. By similar token, many users prefer a low quality authentication, e.g., a weak or repetitive password, despite the risk incurred….

The *recoverable encryption* (RE) in [JLS10] was intended to alleviate all these troubles. An RE scheme recovers private data from their specifically encrypted backup. The encryption is such that in the

[1] No paper edition. The only notification of changes is new "Revised" date.
[2] George Mason University, Fairfax, VA, jajodia@gmu.edu
[3] Université Paris Dauphine, LAMSADE, witold.litwin@dauphine.fr
[4] Thomas Schwarz, UCU, Montevideo, tschwarz@ucu.edu.uy

absence of the owner, the brute-force recovery, i.e., without any additional knowledge of original data, is feasible. It remains excessively difficult or dangerous for illegal attempts. In [JLS10], RE was put into practice for the client-side encrypted data outsourcing to an LH* based cloud file. The private data in the scheme, termed LH*$_{RE}$, was the encryption key. It was backed up as the shares of the secret randomly distributed among the LH*$_{RE}$ data records encrypted using the key, the whole file being spread over many cloud nodes. To recover the key, one needed to collect all the shares. The intruder had to break then into typically excessively many nodes.

Below, we propose a different principle, leading to schemes collectively called *RE through noised secret*, RE$_{NS}$ in short. Here, RE basically concerns a high quality encryption key. The backup storage requires a single computer or cloud node only, unlike for LH*$_{RE}$. The owner may put the backup into a cloud or, alternatively, handle it to an Escrow or Admin. The recovery time *D* through the processing at a single computer (or cloud node) is adjustable at owner's discretion. The owner should estimate *D* long enough to deter the practice, e.g., in order of months or years. To speed up, the recovery may use then a (multi-node) cloud. It speeds then almost linearly with the cloud size. The owner may define in this way some *acceptable recovery time R*, in seconds, minutes or, say, up to an hour. In practice, the owner may set up *D* that *N* is at least in hundreds, thousands or dozens of thousands of nodes. Large clouds become easily available, e.g., Google and Yahoo claim having about 50K-clouds, Azur advertises the availability of millions of nodes…

The owner chooses *D* assuring the number of basic steps of any key recovery procedure large enough to make it somehow costly. The owner may bear in mind also that using a large "army" of nodes should be rather visible, e.g., through many traces in logs. The overall hardship may inversely follow owner's trust to the escrow or in the cloud safety. The owner may end up confident that the cloud cost and the likelihood of being caught through audit are high enough to deter illegal attempts.

Technically, an RE$_{NS}$ scheme hides the key within a *noised* secret. Like the classical secret-sharing, this one consists also of at least two shares formed also "classically". One of these shares is however in addition *noised*. The term means that it is redefined in a specific way making it hidden among a large number *M* of pseudo-random *noise* shares. With very high probability, noise shares other than the noised one do not reveal the secret. The RE$_{NS}$ recovery searches a noised share by brute force. It may need to inspect every noise share and at least half of those on the average. This average corresponds to the case of, so-called, *single noise* per share. It increases with additional *multiple* noising. No brute force method known at present can disclose a noised share faster than RE$_{NS}$ recovery.

The search time on a single computer or cloud node, whether the worst or the average, scales up about linearly with *M*. The owner chooses *M* in proportion to *D* through the estimate of the recovery difficulty as defined by the throughput of the operations at a single computer or cloud node. The noised share search may speed up almost linearly with the cloud size *N*. The recovery time may speed up accordingly, i.e., possibly almost reaching D / N. To achieve the speed up, RE$_{NS}$ schemes partition the search over the nodes. A *static* partitioning has *N* fixed upfront, before the recovery processing partitioning starts. A *scalable* one incrementally adjusts *N* during while partitioning. The static scheme achieves the linear speed-up and the optimal respect of *R*, provided the homogenous throughput. If the owner's estimate of a node throughput is accurate, then *N* is $N \cong D / R$. The recovery may however miss the *R* target over a heterogeneous cloud. A scalable scheme applies then. When the static scheme applies, one can always apply the scalable one instead. However, it should generate then usually a larger cloud for the same *D, R* and node throughput, up to about 30 %. For many, this price may be acceptable for greater flexibility. Especially, since the speed up is then also linear and accordingly better. In sum, the scalable scheme is generally more versatile and appears preferable for a private cloud.

An about 10K-node cloud may in this way provide the speed up from, say, *D* of up to two months to *R* less than ten min. A 100K-node cloud may allow for *R* of less than 1m etc. The average timing accelerates accordingly. For the static scheme, it is respectively *D*/2 and about *R/*2. Here for instance, it would speed up from 1 month to about five min or, even, to 30sec. For the scalable scheme, the latter figure may be the same, but usually both the worst and average times should be faster than *R* and *R*/2 respectively, while the cloud generated should be accordingly larger. For a homogeneous cloud the difference for each factor is about 30%, as we have mentioned. For a heterogeneous cloud, the figures depend on the throughput of nodes involved. The formal calculation appears complex and is among future work goals.

Below we analyze the feasibility of RE_{NS} schemes. We define the basic concepts and the above mentioned basic schemes. We prove the correctness, the safety and other properties we have indicated. We point to variations of the basic schemes for future analysis. We discuss especially the use of multiple noised shares, unlike in the basic schemes. The average timing increases with the number of noised shares towards *D* and *R*. We show that some users may appreciate the feature. While no RE_{NS} scheme is implemented as yet, our study shows that those proposed fulfill the intent and should be of interest to numerous applications.

Section 2 starts with the basic concepts. Next, we discuss the static partitioning and continue with the scalable one. In Section 3 we briefly discuss the use of multiple noises. In Section 4 we address the state-of-the art. Finally, in Section 5, we conclude and discuss the further work.

2. Recoverable Encryption through Noised Secret

The algorithm starts with the processing of the secret data by the owner, producing the (*noised*) backup. We call this step *client-side (recoverable) encryption*. The client sends the backup to the *(backup) server*: an escrow, admin.... that we usually note *E* in what follows. The owner usually accompanies the backup with some access path to the data encrypted using the key. The server-side recovery decrypts the backup, i.e., recovers the key, upon request.

2.1. Client-Side Recoverable Encryption

The only secret data, say *S*, to back up, we consider below, unless we state otherwise, are a high quality encryption key. The highest quality keys at present seem those of AES standard, 256b wide. We basically consider these keys for the examples that follow. Nevertheless, the owner can backup shorter keys as well, e.g., DES ones. By similar token they could be larger, e.g., 500b Diffie-Helman ones, used for key exchange or as trusted IDs [JLS12]. The crucial for safety property is that the client (key owner) node, say *X*, should be able to consider *S* as an apparently randomly generated integer. The owner encrypts any data to remain secret with unproven apparent randomness through such a key. For instance, this can be an ID, a password, a pass phrase, any large file….

To start with the backup, *X* chooses some *dissuasive recovery time* that we usually note *D* below. *X* predicts *D* is large enough to be impractical. Technically, *D* is the owner's estimate of the worst time for an RE_{NS} recovery. The case occurs on a single computer e.g., in the escrow's possession, or a 1-node cloud. The average 1-node RE_{NS} recovery time should be then also rather dissuasive, being equal to $D/2$ at least, as it will appear. Depending on the trust to the escrow and cloud safety, the owner may choose D to last days, months or years.

Next, X fixes the *acceptable recovery time R*, $R << D$. The owner is willing to wait that long at most till the recovery completes. The ratio D/R implies for the owner that the cloud size to use that is presumably $N = D/R + \Delta D/R$. The actual N depends on the accuracy of the owner's throughput prediction. We elaborate on it later on. Δ can be almost zero or somewhere between zero and about 30%. It depends on the RE_{NS} scheme used.

We now denote $w(y)$ the bit width (length) of integer y. X generates a random number, say s_0, where $w(s_0) = w(S)$. Below, we thus have $w = 256$ by default. Next, X calculates $s_1 = S_i$ XOR s_0. Afterwards, X applies to s_0 a good one way hash, say H, e.g., SHA 256 by default in what follows. This produces the *hint*, say h, $h = H(s_0)$. In fact $w(S)$ can be shorter than 256b, e.g., if one uses 128b DES keys. In which case SHA 256 pads S automatically.

X presumes further that a cloud node or the escrow's computer is a typical one according to the present technology, e.g., some 1-core Wintel node. X further knows the parameter termed 1-*node throughput T* we introduce soon. T quantifies the computational speed of a typical computer for RE_{NS} purpose. One measures T per time unit, say per second. It will appear that typically $T \cong 2^{20}$ then. On this basis, X calculates a large integer $M = \text{Int}(DT)$. Next, X randomly chooses some integer m within interval $I = [0, M[$. After that X defines integer f as $f = s_0 - m$. Then, X forms the so-called below *noised share* $s_0^n = (f, M, h)$. Finally, X forms the *backup S'* as $S' = (s_0^n, s_1)$ and sends S' to the escrow.

In essence, X first applies to S the "classical" secret-sharing, creating two (actual) shares s_0 and s_1. It is the common knowledge that $S = s_0$ XOR s_1. Then, X creates a finite *noise space* that is here *interval I*. Each value within I is a *noise*. The numbers $f, f+1 \ldots f+M-1$ become *noise shares*. M quantifies the number of noise shares and becomes *noise width*, Figure 1. The actual share s_0 is one of the noise shares. This is easy to see for I above. One may identify s_0 from the noised one s_0^n through the successful match $H(s_0^n) = h$. This is possible, since for any noise share $s \neq s_0$, the practical collision probability of a good one way hash is zero, hence $H(s) \neq h$. Also, since M is finite, one may generate every noise in the noise space and calculate the hash of every noise share. For I, one may simply loop through the values $0,1 \ldots M$-1.

The share s_0 is the m-th noise share in this order. The value of m is however random and not in the backup. It may be equally likely anywhere in I. The *brute-force* recovery, not knowing m, has to guess it. The RE_{NS} recovery does this then by (hint) *match attempt* $h \overset{?}{=} H(s)$, for every noise share s implied by the guessed m. As it is easy to see, since every noise in the noise space is equally likely, regardless of the way of picking up these noises, any brute-force guessing may need to attempt a match for even every of M noises and $M / 2$ at the average at least. Especially, since H is a one-way hash, so it is not possible at present to determine s_0 as $H^{-1}(h)$. Also, since by general properties of the secret-sharing, a noise share other than the noised one cannot reveal the secret.

Practical values of M appear in hundreds of billions at least, e.g., $M = 2^{40} \div 2^{50}$, given the usual SHA 256 speed, [B11]. These choices should effectively let the 1-node RE_{NS} recovery time that is, to remind, a brute force one, to be typically by far too long in practice. As it will appear, any alternate brute force recovery methods may lead to even longer possible and average timing. As we will elaborate, all this reasonably guarantees to the owner the correctness and safety of *R* choice. The former means that the 1-node brute-force RE_{NS} recovery eventually always returns s_0, but not faster than in the expected timing, providing the correct estimate of *T*. The latter means that no brute-force recovery could proceed faster.

By analogy to the secret-sharing terminology, we call the owner's action above *noised secret-sharing*. One can see that the characteristic difference is that the latter, unlike the former, generates one share, s_0 above; with a n*oised* part. This part consists here at least of the rightmost *w* (*M*-1) bits of s_0. The actual value of the noised part is hidden somewhere within a purposely large noise space, *I* above. The value of a noised share, unlike that of a "classical" one is therefore purposely somehow fuzzy. It is the noise width that defines the fuzziness. As we stressed already, *H* is a one-way hash, not allowing at present to calculate s_0 as H^{-1} (*h*). As we also discussed, for any noise share *s* guessed as the (hidden) noised share s_0, the match attempts are then the fastest practical way to find whether *s* is s_0.

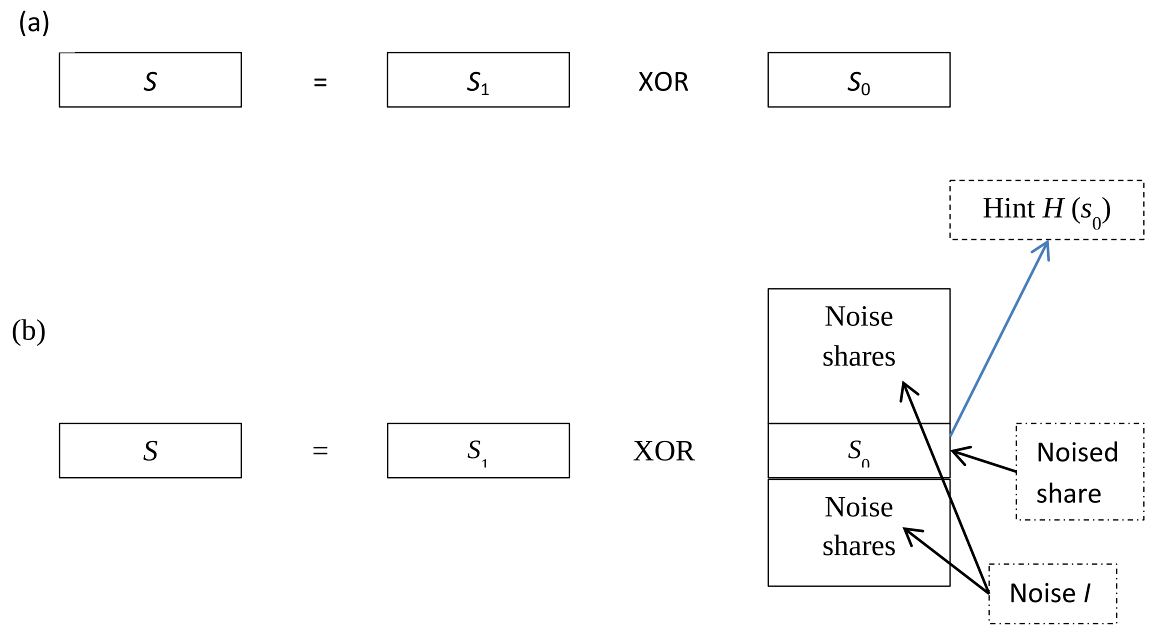

Figure 1. 2-share (a) Secret-Sharing and (b) Noised Secret-Sharing.

Example 1. *X* wishes a backup of some AES key, thus becoming *S* here. Also, *X* wishes *D* to be at least a month, i.e., at least 2^{22}s. Next, *X* fixes *R* to be less than 10m, say $D = 2^9 = 512$s. This lets *X* to expect the use of at least $N = 2^{13}$, i.e., of 8K-node wide cloud. These *D* and *N* seem to *X* big enough to deter an unwelcome recovery.

X also believes throughput *T* of RE_{NS} recovery to be 2^{20} match attempts per second. A match attempt involves the hash, a match attempt $h \overset{?}{=} H(s)$ for some noise share *s* and most often some progress over *S*. Given SHA 256 speed figures, such *T* seems reasonable for a popular about 2 GHz 1-core Wintel node, [B11]. *X* fixes therefore *M* to $M = 2^{42}$. This is the noise width here and $I = [0, 2^{42}[$ is the noise space. Next, *X* produces a 256b long random integer as the actual s_0, calculates *h* and $s_1 = S$ XOR

s_0 as the other actual share. Then, X chooses a random integer within I = [0, 2^{42}[, becoming m. Next, X computes f and forms the noised share s_0^n. Finally, X forms S' and sends it out.

The brute-force attacker of the backup can see in f the bits of s_0 which are perhaps out of the noise. These are here all but the rightmost 42, i.e., the left 214b. The attacker may XOR these with same bits of s_1 recovering these bits of key S. For a high quality key, AES, especially, those bits do not let to infer the noised value of the rightmost 41 bits of s_1 and s_0, hence of the rightmost 41 bits of S besides as well. The only use of the visible bits one can make for the recovery is (i) to guess a noise, then, (ii) try out through a decryption attempt of the actual data whether S calculated that way is the right one. Any such verification we are aware of is by far longer than the hint match attempt. The guessed value m may be equally likely anywhere within I. There are then also "only" M equally likely noise shares. However, we hash over w (s_0) being here 256b. By the property of a high quality one way hash, the value space of H is as large. There is no possibility at present to create a pre-computed inverted file that would have all or most of the hashes for a key. Same is true, more generally, for any other pre-computed decryption method known. If it was possible, an even instant disclosure of the noised share could follow. Altogether, as we already stated, no guessing of the noised share can be at present faster than through RE_{NS} match attempts.

Example 2. Now suppose that the secret data are a DH private ID F, e.g., as used in CSCP, [JLS11] and Section _. This ID is necessary for the data access. A DF number is the 512b long integer. The owner then first chooses a key that becomes S and encrypts F. Then, X backs up S as above. The escrow could be the Admin of CSCP scheme. The owner may accompany then the backup with the encrypted F.

Actually, RE_{NS} backup would remain safe, in the discussed sense, even if the method was applied to F directly. The reasons are similar to those above discussed for AES key. The only alternate method to hint matching would be then the guessing and match attempts through verification in CSCP cloud catalogs. But these can only be slower by far than through hint matching. Thus the direct application of RE is not limited to encryption keys. The result can however remain then also safe or not depending on the case.

Example 3. To illustrate the latter point, suppose now that one attempts as S the string F within the quotes, F = "The quick brown fox jumps over the lazy dog". The choice of I as above could noise the 6-byte substring "zy dog" only. Likely, only seldom folks would not grab the secret from the visible string, without any hint matching. In contrast, it cannot happen, if F is first encrypted through a high-quality key becoming then S.

2.2. Server-Side Decryption

2.2.1. Overview

As discussed, to determine S by brute force from noise shares one has to search over I for a noise share s such that $s = s_0$. For the server, say E, for the secret data under consideration, to determine it through the match attempts as above is the fastest way. For a brute force search, every noise is also equally likely. As we stressed, first, the server needs to explore thus I until the good luck. Next, the server may need to loop through all M-1 values, and M/2 on the average. Assuming the discussed choices of M by X, the chances of a fast success by 1-node recovery are extremely unlikely. More precisely, the probability of discovery after attempting only up to M/*X* noises is 1/*X* [5]. The server considers therefore upfront the actual time to do it by itself too long in practice.

E believes it to be a large value, let us say *D* again, although *E* does not necessarily know it. *E* distributes therefore the match attempts over an *N*-node cloud. The constraint on *N* is that it leads to the recovery time of *R* at worst. There are then two generic RE distribution schemes we qualify respectively of *static* and *scalable* partitioning. For the static partitioning, *N* is fixed upfront of spreading out the RE and does not change afterwards. The scalable partitioning scales up *N* incrementally, while the RE calculus spreads out. The static partitioning appears preferable for a public cloud with nodes known to be identical and possible to allocate to a single client at a time. The scalable one applies there as well, but generates on the average a 30 % larger cloud. In contrast, it may be the only practical on a private or, more generally, rather heterogeneous cloud. The nodes are there usually diverse, with the throughput for a task cumbersome to predict at best.

Upon receiving the backup, the server simply stores it, together with some data that let to identify a legitimate backup requestor. Those data are not the object of our work. When the requestor issues the query for the backup, the requestor sends in particular *R*. For both schemes, the cloud service follows then the same phases whose content however differs. For the static scheme, we begin with; the *Init* phase fixes *N.* Next, the *Map* phase distributes the computation of match attempts over the *N* nodes. Every node continues with the *Reduce* phase, where it searches for the luck. Finally, every node performs the *Termination* phase. This one aims at the ASAP-end of the computation. The Map/Reduce terminology reflects the current popular vocabulary.

2.2.2. Static Partitioning

We call *load L* at a node the number of values among *M* assigned to the node for match attempts. We call (node) *throughput* and note *T* the number of attempts the node supposedly performs per time unit, a second typically as we said. We call node *capacity B*, the size of the bucket of attempts that the node achieves in time *R*, given *T*. We have obviously $B = R*T$. Next, we call (node) *load factor* α the value L / B. Obviously, the node performs (on the average, or for sure …) all its map attempts in time *R* only if $\alpha \leq 1$. The goal of the scheme is to get $\alpha \leq 1$ and $\alpha \cong 1$ at every node participating in the calculus. This goal generates the smallest, hence in this sense *optimal*, cloud respecting *R*. Finally, we say that a node *overflows* iff it has $\alpha > 1$. Notice that our terminology maps purposely our problem towards that of designing a scalable distributed data structure (SDDS) or a file generally [LNS96].

2.2.2.1. Init Phase

When *E* requests the cloud service, *E* gets access to one cloud node. We call this node *coordinator*, *C* in short. *E* sends data $S'' = (t, F, I, R)$ to *C*. *E* hides s_0 to avoid the disclosure of *S* by the cloud, for obvious reasons.

C determines on itself the 1-node load factor, say α^M induced by all *M* attempts. Basically, it knows somehow its *T*, computes trivially *B* from and calculates $\alpha^M = M / B$, since $M = L$ in this case. If by any chances, it happens that $\alpha^M < 1$, *C* performs the (entire) recovery. Otherwise, *C* presumes that its *T* will be also *T* of every other node assigned to the calculation. It then calculates *N* as $N = \alpha^M$. Finally, *C*

[5] For the basic scheme, we call later, with a single noise. For the variant called with multiple noising, this probability can be even much smaller.

requests the allocation of *N*-1 nodes. After that, *C* labels every node of the *N*-node cloud constituted in this way with the unique *node number i* ; *i* = 0,1…N-1. *C* itself becomes node 0.

Example 4. We continue with the assumptions of the previous examples. *C* knows *T* to be $T = 2^{20}$ per second, as *X* supposed besides. It computes *B* from *R* got in *S''* as $B = 2^{29}$ (match attempts per second). *M* received leads to $\alpha^M = 2^{42\ -\ 29}$ = 8K. This is far above $\alpha = 1$ and in fact a quite heavy overflow (overload). *C* requests therefore the allocation of 8K-1 cloud nodes. It labels itself node 0 and numbers the nodes got 1,2…8K-1. It finally advances to next phase.

2.2.2.2. Map Phase

C sends the message, say *D*, to every node of the cloud. *D* contains *S'', i* and request (program) *P* defining the two remaining phases for each node. It also contains some (physical) address of *C*, letting every node to send *C* the positive match message.

Because of the termination protocol, this phase may get interrupted at any time by a message with the result. It then goes to Termination phase at once.

2.2.2.3. Reduce Phase

Every node starts this phase, by executing *P*, immediately after *D* reception. *C* itself starts it in parallel to Map phase or after. *P* requests node *i* to attempt matches for every $m \in [0, M[$ such that $i = m \bmod N$. In practice it means a loop over every value $m = i, i + M, i + 2M$… while $m < M$. If a match succeeds for some *s*, the node sends *s* to *C* and exits to Termination phase. What the node does anyway, otherwise.

Because of the termination protocol, every node during this phase may also get interrupted at any time by a message from *C* requesting to go to Termination phase at once.

2.2.2.4. Termination Phase

The coordinator sends *s* found to *E.* When *C* gets *s*, it requests the calculus end at every node and its de-allocation. Details of where and how depends on the cloud used. Also, every node, that managed to terminate the match attempts unsuccessfully, wraps up its state enough to get de-allotted by the coordinator. Details of where and how depends on the cloud used.

When *E* gets *s,* it completes the recovery by XORing *s* as s_0 successively with each fragment $s_{0,i}$ and finally sends the result to *X*.

2.2.3. Discussion

It is rather easy to see that the scheme is correct and safe in the discussed senses. It is safe as for same *N*, there is at present no way to disclose s_0 faster than through the RE_{NS} match attempts. Next, no cloud intruder can discover *S* since the server does not send out s_0. The scheme is correct, since it always recovers s_0. Also, it does not miss *R* provided that every actual *T* remains the expected one. All these properties result from the reasons already discussed, especially for the hypothetical 1-node recovery and because of the details that follow.

First, it easily appears that the scheme realizes the perfect static hash partitioning over the cloud. It is very simple and potentially allows for the optimally small cloud for any given *R*. The average recovery time, say R^a is then about $R / 2$. The rationale is that every noise share at every node is equally likely to be the noised one. This results from the randomness of *m* within *I*, compound with that of the high quality one way hash. At worst, all the noises mapped to a node may be visited until the match succeeds. The average value is then clearly $M/N/2$. This leads also to the average 1-node recovery time of $R/2$.

For R and R^a, first the incidence of messaging time of the Map phase and of that of the Termination phase appear negligible with respect to the timing of Reduce phase. That is why, we may effectively have $R \cong D / N$ and $R^a = R/2$. This provided that $\alpha = 1$. The actual respect of this constraint and more generally, of $\alpha \leq 1$, to avoid exceeding R, so having also $R_a > R/2$, depends for every node n on how close B_n really is with respect to B_0. If $B_0 > B_n$ often or even $B_0 >> B_n$ for some n, many nodes overflow and C is likely to miss R, perhaps heavily. How precisely likely the event is remains subject of further work.

Vice versa, many nodes can get under-loaded and the cloud size may by far exceed the optimal one for R. Necessary condition for the optimality is the same computational power at each node. This is more likely to occur on a public cloud than on a private or hybrid one. However, even on a public cloud it may not exist. For instance, - if several virtual machines may share a node. Notice that all these comments equally apply to possible discrepancy between the owner's initial estimate of the 1-node throughput and the actual ones. A good side of a larger than the optimal cloud given node under-load, is that the average recovery time becomes somehow faster than $R/2$ as well.

One direction towards more general respect of R, i.e., less stringent on nodes throughput, is probably to calculate N, using $\alpha < 1$. The theory of (static) hash files hints towards this direction and to $\alpha = 0.8 \div 0.9$ as promising choice. The probability of an overflow is then negligible under good hash assumption. On the other hand, it is possible to generate static partitioning adapted to different B_n's. However, again as for files, this seems of doubtful utility. We leave both issues for further work.

Unlike above, the termination phase in practice may need to deal also with failures. A message may get lost or a node with the lucky number may fail before its processing. Also the prediction may heavily fail, as discussed. For all these reasons C may not get any result in a reasonable time out. It then needs some dialog with participants, for the recovery. We leave the extensions to the termination phase dealing with these issues for future analysis as well.

Finally, every hash function may generate collisions. If it happened, the calculus could pick up an incorrect noise share, as the calculus could stop at the first encountered. The basic scheme would then incorrectly decrypt the secret. The scheme does not consider this possibility. The reason is that the probability of collision with a good one-way hash, SHA 256 especially, is in general so negligible that one usually presumes it zero in practice. Nevertheless, one may feel a collision still possible. One way to amend the scheme is then to also hash S itself. The hash of the decrypted S and the original one should coincide. If not one restarts the search for the noised shares and attempt the decryption for every share matching the hint.

Example 5. On the basis of Example 4, we consider the use of the CloudLayer cloud, [C12]. The hourly pricing in Jan. 2012 guarantying the exclusivity of a node to the renter was 0.30c. This exclusivity, called in CloudLayer terminology *private* option, is crucial for expected T stability and homogeneity among the nodes. Crucial in turn to the success of the static scheme as discussed. On this basis, we extrapolate a little the pricing scheme, considering that 5m is the minimal charge per node used. Since R was 10m, C may provide the cloud bill estimate as up to about 400$ and 200$ on the average. This is in general easily affordable for valuable data for a private user. It is obviously a cheap price for a commercial one. For a larger cloud, the price would hike up accordingly. Still it could be reasonable in rush cases one can imagine.

2.2.4. Scalable Partitioning

We now suppose that every node *n* is able to *split* as we detail soon. It is also the only to know its *B* that we note B_n. The scheme goes through the same phases as the static one.

2.2.4.1. Init Phase

When *E* requests the service, *E* gets access to one cloud node. This node labels itself node 0. It constitutes the *initial* cloud with *N* = 1 node only. As previously, *E* sends data *S''* = (*t*, *M, R*), while hiding s_0 for the same reasons.

2.2.4.2. Map Phase

Node 0 determines A_0, its load as $L_0 = M / A_0$, its capacity B_0 as R / A_0 and its load factor $\alpha_0 = L_0 / B_0$. If by any chances, $\alpha \leq 1$, node 0 executes the calculus, sends the result to *E* and goes to termination phase. Otherwise, node 0 overflows and *splits*. Node 0 creates then the parameter termed *node level* and usually noted *j*, with *j* = 0 initially. Next, the node asks for the allocation of a new node, it labels node 1. Then it sets *j* to *j* = 1 and sends to node 1, the data (*B', j,* a_0), where a_0 is some (physical) address of node 0, letting node 1 to send data there. From now on, the cloud has *N* = 2 nodes. Observe that the initial cloud had in fact $N = 2^0$ nodes and now scaled up to the size of $N = 2^1$ nodes.

Next node 0 recalculates L_0 as $M / (2 A_0)$ and α_0 accordingly. The reason is that that the reduce phase if started for *N* = 2, i.e. for the cloud of these nodes only, would lead to *M* / 2 attempts per node at max. As it will appear node 1 will have the same load. Node 0 splits again if it overflows, i.e., $\alpha > 1$ still. Otherwise, it starts Reduce phase. To split, node 0 creates new node and labels it node 2. This in fact the result of the new (logical) node address computation as $0 + 2^j = 2$. Finally, node 0 performs $j := j + 1$ and sends (*B', j,* a_0) to node 2. Node 2 stores *j* as its own initial level. Hence, both nodes end up at level 2.

Node 1 acts similarly, starting with the prediction of its A_1 hence α_1. The node created, iff $\alpha > 1$, is however node 3, as $3 = 1 + 2^j$, with *j* used being *j* = 1, i.e., that set upon node 1 creation. After the creation node 3 ends up at level 2 as well. If both, node 0 and 1 split, the cloud has now 2^2

From now on, every node *i* that did not entered the Reduce phase, recursively loops over the overflow testing and splitting. The general rule they act upon, compatible with the above detailed processing at node 0 and 1, is in fact as follows.

1. Every node *n* ; *n* = 0,1… after its creation or last split, calculates its *L* as $L_n = M / (2^j A_n)$ and tests whether $\alpha_n > 1$. For this purpose, a newly created node *n*, starts with the prediction of A_n. If there is no overflow, the node goes to Reduce phase.
2. Node splits. It creates node $n' = n + 2^j$, sets *j* to *j* + 1 and sends (*B', j,* a_0) to node *n'*.

In this way, for instance, as long as node 0 overflows, it continues to split, creating successively the nodes 2^j with *j* starting at *j* = 0, i.e., nodes *n* = 1, 2, 4, 8… Node 1 will start with *j* = 1 and append nodes $1+2^j$ that are *n* = 3, 5, 9, 17… Likewise node starts with *j* = 2 and creates nodes $n = 2 + 2^j = 6$, 10… In general, every split appends one new node to the cloud. Each new node gets a unique logical address as one can easily figure out. Iff all the splitting nodes end up with the same *j*, these addresses form a contiguous space 0, 1…*N*-1 where $N = 2^j$. This happens, if they have the same *A*. Finally, observe that every split halves *L*, hence α, of the splitting node *n*. The new node *n'* starts with the same *L*. But not necessarily with the same α, as *A* may differ among the nodes.

Because of the termination protocol, every node during this phase may also get interrupted at any time by a message requesting to go to the termination phase at once.

2.2.4.3. Reduce Phase

In this step every node n matches t and h (s) for every $s \in [0, N\text{-}1]$ and such that $n = s \bmod 2^j$. Thus, for $j = 2$ for instance, node 0, loops somehow over s =0, 4, 8… Likewise, node 2 for $j = 1$ loops over 2,6,10…. If node 1 did not split even once then, hence it carries $j = 1$ and thus loops over the remaining values of [0, M[which are 1,3,5,7… If the match succeeds at any node $n > 0$, i.e., $t = h$ (s) for some s, then node $n > 0$ sends s as s_0 to node 0 and exits the step. Otherwise it must succeed at node 0. Node 0 sends s_0 finally to E. If no match succeeds, a node only exits the step.

Because of the termination protocol, every node during the above calculus may also get interrupted at any time by a message requesting to go to the termination phase at once.

Observe that every value in [0, M-1] is mapped for matching attempt to one and only one node. Normally, i.e., without any failures, the step must thus end up in a positive match at some node, as we already hinted in fact. For each value, there is also always only one match attempt, at most. This efficiency is clearly due to the conjunction of the Map and Reduce steps algorithmic.

2.2.4.4. Termination Phase

Basically, this step frees all cloud nodes used. It serves also to save computation power spent for nothing usually otherwise. This would be the case of every node among N other than the lucky one. For both purposes, node 0 broadcasts the termination message to all nodes it has created. Each of these nodes does the same. Obviously, every cloud node will get the message, regardless of cloud growth. Each node acts accordingly, regardless of the phase it was.

Optionally, as for the static scheme, this step may also include protection against the failure of receiving the successful match message by (unlucky then itself) node 0 from some node within some expected time. It lets indeed node 0 to localize any node that could not receive the map message or the successful match message from that node got lost on its way to node 0, or which would be the lucky one, but failed during the computation. We leave details for the future analysis, as for the static scheme.

Example 6. We continue with the data of Example 4. We have thus $\alpha_0 = 2^{42-29} = 2^{13}$, initially. Node 0 splits therefore 13 times till $\alpha_0 = 1$. The splits append successively nodes 1, 2, 4…4096. Then node 0 enters the Reduce phase. It attempts the matches for noise shares 0, 8192…$2^{42} - 2^{13}$. Likewise, node 1 has $\alpha = 2^{12}$ initially. It splits consequently 12 times, appending nodes 3, 5…4097. It then also enters the Reduce phase. It attempts the matches for the values 1, 8193…$2^{42} - 2^{13} + 1$. And so on, for node 2, 3…4095. The latter node has $\alpha = 2$ initially, hence it splits only once. As the result, we get $N = 2^{13}$. This is the optimal (size) cloud here, with every $\alpha = 1$. N is as small as it could be for the static scheme. Notice that such α happens for the scalable one, any time we have $\alpha_0 = 2^i$ and all T's equal.

Consider now the same initial data for node 0. It would split therefore as above. Suppose in contrast for node 1 that $T_1 = 8T_0$, e.g., it is an 8-core node, compared to 1-core. The capacity B of the node increases then eight fold as well. Then, we have initially $\alpha_1 = 2^9$ only. Node 1 would split therefore only nine times, generating nodes 3, 5…513. Nodes 1025, 2049 and 4097 would not exist. The cloud would have at least three nodes less. It would be smaller than that generated by the static scheme. Reduce phase at node 1 would process shares 1, 1025, 2049, 3073, 4097…

Finally, consider that in turn, node 0 happens to be eight times faster than node 1. The splits of the latter would generate three additional nodes. The static scheme would simply fail with respect to the goal.

2.2.5. Discussion

As for the static scheme, it is rather easy to see that one is correct and safe in the discussed senses. All this for the reasons already discussed and because of the details that follow.

First, it is easy to see that the scheme always realizes the hash partitioning. It does it regardless of the number of splits, triggered indirectly, through α values, by *T* heterogeneity. It should typically allocate more nodes than the *N* optimal for the static scheme, say $N^S = M / T$. The randomness of *D* hence of *M* choice provides that of *M* mod *R*, for any *N* and any $n = 0,1,\ldots N\text{-}1$. Hence, any $\alpha_n \in]0.5, 1]$ is equally likely. This yields α on the average $\alpha = 75\ \%$ at best[6]. Instead of fixed $\alpha = 100\ \%$ for the static scheme. The average difference is thus about 1/3*N* more nodes. The worst case α is almost 50 % at every node. If this corresponds also to the same *T* at every node, the scalable scheme generates than a cloud twice as large. Lower load factor leads in turn to faster match at the average. It is clear that it is under *R*/2. More precise determination of α and *N* in various conditions remains a future goal.

If however for some node *n*, T_n is under $T_{0,}$ then node *n* will split more times than for the static scheme, increasing *N* accordingly. The average timing is perhaps affected, not the worst case obviously. The static scheme in this case would simply fail with respect to the *R* respect, i.e., would be incorrect. In contrast, if for some *n*, $T_n > T_0$, enough to create $\alpha_n \leq 1$, then the scalable scheme may end up with even smaller *N* than for the static one. The latter could indeed comparatively uselessly split the calculus at that node, under its assumption of $T_n = T_0$. Clearly the scalable scheme is more versatile. The experiments on actual clouds are probably the only way to evaluate both schemes more accurately. The static scheme should be quite easy to try out on a Hadoop implementation. The scalable partitioning is however beyond the current capabilities of these implementations to our best knowledge. Hence, it seems to require a dedicated tool.

The scalable scheme is correct with respect to *R* if no T_n slows down during the calculation. Otherwise, the lucky node may get $\alpha_n > 1$ for time long enough to finally miss *R*. The scheme has the potential to be amended for such situations so that every node monitors *T* once the match attempts started, while $\alpha \leq 1$. If it finds *T* decreasing enough to make $\alpha > 1$, then it splits. This capability may be quite attractive. There is no such possibility for a static scheme, obviously.

Finally, choosing a scalable scheme on a public cloud may still be advantageous despite perhaps the cloud even twice as wide. The reason may be the pricing structure. On CloudLayer for instance, [C12], it should be in practice necessary to choose the so-called private option for a static partitioning, making the allocation of a node exclusive. This is the precondition for the homogeneous throughput, crucial for the static partitioning. The usual allocation mode termed public may lead to heterogeneity. Several virtual machines may share a node then. However, the public mode is three times cheaper than the private one. Choosing a public option with scalable partitioning may then anyhow save somewhere between 1/3 and 2/3 of the bill.

[6] Open question at present: why not ln2 = 0.69, known for dynamic files generated also by half to half splits?

3. Multiple Noising

This variant that we only sketch here, noises share s_0 with $F > 1$ noises. The rationale is that higher F makes the 1-node average recovery time closer to D. By the same token, it becomes less likely that the recovery happens in a small fraction of D. Likewise become the chances of the success using a cloud several times smaller than that generated for sure recovery within R limit. Otherwise, a user could perhaps consider the figures likely enough to encourage illegitimate attempts.

The *multiple noising*, we also call now F-noising, generalized the use of a single noised share above. In essence, the owner creates now F noised shares. Every noised share is an actual share of an (F+1)-share secret hiding the secret key, say S as before. The owner defines these shares in a specific way we show below, using for each a random noise in $I = [0, M[$. The owner produces also as usual share S_F and the F hints. All this enters the backup. The recovery makes at most M match attempts that suffice to discover all the noised shares. Each attempt compares the hash of a noise share to all the hints. Once all the noised shares discovered, the escrow produces S from S_F as usual.

Formally, the owner starts by choosing M and one random share, say s_0, e.g., 256b-wide for AES. S/he also defines random noises $m_0 \ldots m_{F-1}$ within I. Next, the owner calculates $f = s_0 - m_0$; $s_i = f + m_i$ for $i = 1 \ldots F\text{-}1$; $s_F = S$ XOR s_0 XOR...XOR s_{F-1} ; and $h_i = H(s_i)$ for $i = 0 \ldots F\text{-}1$. Finally, the owner forms the backup as $S'_F = (f, M, S_F, h_0 \ldots h_{F-1})$. The recovery proceeds basically as before, providing every cloud node with M and all the hints, while S_F remains at the escrow. The matches attempt possibly every value within $I' = [f, f + M[$. Each attempt matches the hash with every hint. The node sends back the successful matches. The final step recovers S as $S = S_F$ XOR s_0 XOR...XOR s_{F-1}.

The algorithm is correct. All the noises are within I, hence all noise shares are within $[f, f + M[$. The partitioning, assuming cloud use, maps every noised share to some node. Assuming no node failures, the match attempts must find each noised share, as if it was the one used for the basic schemes. The algorithm is also safe. All F noises are random, hence are the F noised shares. Also, by virtue of secret-sharing, one can determine s_F only after the collection of all F noises. No s_i can be computed through H^{-1} and no one can determine any s_i faster than above by an inverted file or some pre-computation, more generally.

The result of $F > 1$ comparisons per match attempt per m is that the probability $p_F(Z)$ of positive matches for all the F noised shares within fraction Z of M is $p_1(Z)^F$. The average 1-node recovery time, say D_a increases from $0.5D$ to $DF/(F+1)$. Similar figure holds for R for the static partitioning. As long as the time for F comparisons is negligible with respect to the noise share hash time, the cloud size, say N_F, should be the same as for the single noise, i.e., $N_F = N_1 \cong D/R$. Otherwise, the throughput decreases so initial α increases and $N_F > N_1$ nodes. The formal study of N_F is left for future work. For the scalable partitioning, if α_a is the average load factor, $\alpha_a \cong 0.75$ we recall, one has now $R_a = \alpha_a DF/(F+1)$.

For, e.g., six noises, R_a may reach therefore $R_a = (6/7)\,R = 0.85$, increasing thus by 35 % of R over the basic one. Chances to recover the secret too fast to the owner's taste decrease accordingly. An owner for whom 1 in 10 chances to recover in $0.1D$ for $F = 1$, seems an invitation to gamble, may be happier. Indeed, a cloud ten times smaller than the one normally generated for given R could then still bring the luck for a single noised share with the same probability. The counterpart of higher D_a is that the average cloud bill hikes with F, even for $N_F = N_1$. Closing on the worst case, i.e., the duration reaches R.

More generally, multiple noising brings to the arena the familiar concept of *assurance*. One can define this one for our purpose as the probability A (Z) that 1-node recovery lasts at least the time $Z\,D$. We have $A(Z) = 1 - p_1(Z)^F$. Thus, e.g., the popular 90 % assurance, corresponds to $F = 7$ for $Z = 70\%$, to $F = 11$ for $Z = 80\ \%$, but already to $F = 22$ for $Z = 90\ \%$ as well. The likelihood that a smaller cloud, e.g., even only two times smaller, may still bring the luck becomes insignificant, e.g., under 1 % already for F = 7. One can think in consequence about the concept of *resilience* to a smaller cloud, e.g., over 99% for our figures here. We leave all these aspects of RE_{NS} schemes for future studies. This is also the case of the overhead the F values imply on the cloud size.

4. Related work

The RE idea appeared apparently in [JLS10]. It was applied to the client-side encrypted data outsourcing to a cloud. The outsourced data formed an LH^*_{RE} file. As its name suggests it is an LH* based scalable distributed data structure (SDDS) [LNS96], [A+11]. The private data for LH^*_{RE}, was the client's encryption key. The backup encrypted the key as the secret whose shares were randomly distributed among the LH^*_{RE} encrypted data records. The whole file was supposed spread over many cloud nodes. To recover the key, one needed to collect all the shares. A duly authorized cloud client could do it through the LH^*_{RE} scan operation, a Map/Reduce action in the current terminology. The brute-force cloud intruder in contrast had to break into the nodes. The intrusion had to typically read almost all nodes of the file. Break-ins that wide appeared excessively difficult in practice. With respect to RE_{NS} schemes, the first difference was the necessity of a multi-node cloud for the backup storage. A larger cloud was more resilient against disclosure. Also, the scheme was designed only for the encryption keys of a file structured through an LH* based hash-partitioning. These constraints do not exist for RE_{NS} schemes. One can see then a safe outsourcing of any SDDS file together with its RE_{NS} key. This could be of interest to many applications.

The scheme termed CSCP in [JLS11] also provides the client-side encrypted cloud outsourcing. Unlike for LH^*_{RE}, CSCP keys are however shared among authorized clients. CSCP uses a variant of the DH protocol for client authentication. A private DH number constitutes the trusted ID and lets the client to safely share or backup any key. Its loss inhibits these capabilities and may lead to key loss. To offset the danger, each private ID is basically backed up as is with an Admin. This may frustrate users, as we discussed. The RE_{NS} backup appears to nicely blend with CSCP scheme. Same observation applies to the popular SharePoint, using DH similarly for group messaging. Again, such schemes seem of interest to many applications

The RE concept roots also somehow in the classical cryptography concepts of *one-way hash with trapdoor* and of *cryptograms* or *crypto puzzles*. Popular search engines and Wikipedia provide numerous references discussing these concepts, e.g., [C11] among most recent ones. RE may be seen as a one way hash where the cloud constitutes conceptually a distributed scalable trapdoor. Also, one may see RE as a scheme for cryptograms (crypto puzzles) that the owner makes complex at will through the quantity of noise injected. At the same time, the decryption algorithm makes the resolution scalable and distributed also somehow at will. This leads to perhaps other RE schemes exploring numerous ideas in these domains. Main ideas in RE opens up also perhaps as a backfire a research direction for those domains that does not seem to be initiated as yet.

The risks of key escrow (a.k.a. key archive, key backup, key recovery) we have mentioned are hardly a new issue. They were studied for decades, especially intensively in the late nineties, [A+97, DB996, W+96]. The wave was triggered by the famous Clipper chip envisioned for storing cryptographic keys by government or private agencies. The concept of recoverable encryption in this context was somehow implicit in the taxonomy in [DB96], becoming explicit in a revision of that work [DB97]. However, our recoverability is basically an optional service. The initial meaning was that it should be mandatory, for the law enforcement, [L99].

That idea was criticized for its inherent dangers, complexity, and overreach. As an alternate, Shamir proposed government key escrow using only partial keys. A government agency could recover the key by making a brute force attack on the unknown part of the key feasible. Rivest & al proposed in turn a non-parallelizable crypto-puzzles, allowing access after a guaranteed minimum amount of time only, [RSW96]. On the other hand, it was observed also that key escrow for the benefit of someone else may need to verify that the escrowed key is indeed the true key [BG97]. This does not apply to our case however, since there is no adversarial relation between the beneficiary of key recovery and the creator or owner of the data. Finally, Blaze proposed a yet different approach, through the massive secret sharing scheme distributing shares of escrowed keys to many (key) share-holders [B96]. None of these techniques was scalable or was ever intended to be.

With respect to illegal recovery attempts, it is also in principle possible to avoid legitimate cloud providers all together, by creating a botnet instead. The danger seems however also unlikely. There is now a proliferation of anti-botnet tools. Almost every computer has a high quality antivirus protection for peanuts. More and more users exercise caution on Internet. Finally, using botnets is now recognized as an undoubtful cybercrime, leading increasingly likely into jail.

5. Conclusion

Encryption key safety is the Achilles' heel of modern cryptography. To reduce the danger of key or of any related private data loss, backups are necessary. Copies increase however in turn the disclosure risk. Recoverable encryption alleviates the dilemma through the emerging cloud technology. We believe the concept of interest to many applications.

Future work should analyze RE_{NS} schemes more in depth. Experiments on public and private clouds are a necessity. They will contribute to assess the practical interest of both schemes. Next, one should address the enhancements we have sketched. Especially those coping with failures as well as those adapting the distribution to the throughput changes during Reduce phase.

Acknowledgments

The current algorithmic results in particular from helpful discussions at the initial stage with Jonathan Katz (U. of Maryland, College Park).

6. References

[A+97] H. Abelson, R. Anderson, S.M. Bellovin, J. Benaloh, M. Blaze, J. Gilmore, P.G. Neumann, R.L. Rivest, B. Schneier: The Risks of Key Recovery, Key Escrow, and Trusted Third-Party Encryption. http://www.crypto.com/papers/escrowrisks98.pdf.

[A+11] Abiteboul, S., Manolescu, I., Rigaux, Ph., Rousset, M.Ch., Senellart, P. Web Data Management. Cambridge University Press, 2011.

[B96] M. Blaze: Oblivious key escrow. Information Hiding, Springer, 1996.

[B11] Crypto++ 5.6.0 Benchmarks. http://www.cryptopp.com/benchmarks.html.

[BG97] M. Bellare and S. Goldwasser: Verifiable partial key escrow. 4th ACM CCS Conf., p. 78 – 91, 1997.

[C11] Chandrasekhar, S. CONSTRUCTION OF EFFICIENT AUTHENTICATION SCHEMES USING TRAPDOOR HASH FUNCTIONS. Ph.D Dissertation. University of Kentucky, 2011.

[C12] CloudLayer Hourly Pricing. http://www.softlayer.com/cloudlayer/computing/

[DB96] D.E. Denning and D.K. Branstad. A Taxonomy for key escrow encryption systems. Communications of the ACM, vol. 39(3), 1966.

[DB97] D.E. Denning and D.K. Branstad. A Taxonomy for Key Recovery Encryption Systems**.** http://faculty.nps.edu/dedennin/publications/TaxonomyKeyRecovery.htm

[L99] Lee, R. D. Testimony of Ronald D. Lee, Attorney General…. March 1999, http://www.cybercrime.gov/leesti.htm

[MLFR2] Miller, E. Long, D., Freeman, W. and Reed, B. Strong security for distributed file systems. In Proceedings of the Conference on File and Storage Technologies (FAST 2002), pages 1–13, January 2002.

[JLS10] Jajodia, S., Litwin, W. & Schwarz, Th. LH*$_{RE}$: A Scalable Distributed Data Structure with Recoverable Encryption. IEEE-CLOUD 2010.

[JLS11] Jajodia, S., Litwin, W. & Schwarz, Th. Privacy of Data Outsourced to a Cloud for Selected Readers through Client-Side Encryption. CCS'11 Workshop on Privacy in the Electronic Society (WPES11), Chicago.

[LNS96] Litwin, W., Neimat, M-A., Schneider, D. LH* - A Scalable Distributed Data Structure. ACM TODS. 12, 1996.

[RSW96] R.L. Rivest, A. Shamir, and D.A. Wagner: Time-lock puzzles and timed-release crypto. Technical Report, Massachusetts Institute of Technology, MIT/LCS/TR-684, 1996.

[S10] Sharepoint 2010. http://sharepoint.microsoft.com/en- us/pages/default.aspx

[W+96] S. Walker, S. Lipner, C. Ellison, and D. Balenson: Commercial key recovery. Communications of the ACM vol. 39, 3 March 1996.