Recoverable Encryption through a Noised Secret over a Large Cloud

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

[1] George Mason University, Fairfax, Virginia, USA
jajodia@gmu.edu
[2] LAMSADE, Université Paris Dauphine, Paris, France
witold.litwin@dauphine.fr
[3] Universidad Católica del Uruguay, Montevideo, Uruguay
tschwarz@ucu.edu.uy

Abstract. The safety of keys is the Achilles' heel of cryptography. A key backup at an escrow service lowers the risk of loosing the key, but increases the danger of key disclosure. We propose *Recoverable Encryption* (RE) schemes that alleviate the dilemma. RE encrypts a backup of the key in a manner that restricts practical recovery by an escrow service to one using a large cloud. For example, a cloud with ten thousand nodes could recover a key in at most 10 minutes with an average recovery time of five minutes. A recovery attempt at the escrow agency, using a small cluster, would require seventy days with an average of thirty five days. Large clouds have become available even to private persons, but their pay-for-use structure makes their use for illegal purposes too dangerous. We show the feaibility of two RE schemes and give conditions for their deployment.

Keywords: Cloud Computing, Recoverable Encryption, Key Escrow, Privacy.

1 Introduction

Data confidentiality ranks high among user needs and is usually achieved using high quality encryption. But what the user of cryptography gains in confidentiality he looses in data safety because the loss of the encryption key destroys access to the user's data. A frequent cause for key loss is some personal catastrophy that befalls the owner of the owner such as a fire that destroys the device(s) with passwords and keys. Organizations have to prevent a scenario where the sole employee with access to an important key leaves the organization or becomes incapacitated. In the past, keys were lost in natural disasters, such as when the basement of a large insurance (!) company was flooded and keys and their backups were destroyed. Many patients encrypt files with health data, but access to them becomes crucial especially if a health issue discapacitates the patient. Encrypted family data needs to be able to allow for their owner's disappearance, e.g. on a hiking trip in the Alaskan wilderness.

A. Hameurlain et al. (Eds.): TLDKS IX, LNCS 7980, pp. 42–64, 2013.
© Springer-Verlag Berlin Heidelberg 2013

A common approach to key safety is a remote backup copy with some *escrow* service [JLS10]. The escrow service can be a dedicated commercial service, an administrator at the organization, a volunteer service, ... However, if the user entrusts a key to an escrow service, the user has to be able to trust the escrow service itself. The escrow service not only needs to prevent accidental and malicious disclosure by insiders and outsiders, but most be able to convince its users that the measures taken to protect the keys against disclosure or loss are of sufficient strength. This might explain why the use of escrow service is not very popular. No wonder that some users prefer to forego encryption [MLFR02], or prefer less security by using a weak or repeated password.

The *Recoverable Encryption* (RE) scheme [JLS10] is intended to alleviate the problem. It encrypts a backup of the key so that the decryption of the backup is possible without the owner using *brute-force*. Legitimate, authorized recovery is easy, while unauthorized recovery is computationally involved and dangerous. RE [JLS10] or Clasas [SL10] was designed for client-side encrypted data stored in a large LH* file in a cloud. The key backup is subjected to secret sharing and the shares are spread randomly over many LH* buckets. To recover the key, an adversary has to intrude buckets, which are stored in different sites, and search them until she can recover all or sufficient shares of the key. Thus, unauthorized recovery of the key involves illegal activity and is at best cumbersome.

Here, we present more general schemes, which we call collectively Recoverable Encryption through a Noised Secret (RENS). In these schemes, a single computer or cloud node suffices as the storage for key backups. The client uses an encryption of the backup key that resists brute force at the site of the escrow agency, decryption is possible through brute force by distributing the workload over the many nodes of a cloud. The user can choose an encryption strenght based on the maximum time D needed to recover the key at the site of the escrow service. On average, the time to key retrieval is $D/2$. The user will choose a time D in years or at least months, depending on his trust into the additional security measures of the escrow service. The user will also specify at maximum time R for legitimate recovery, which is in the range of minutes or even seconds. The relationship between R and D is given by the number N of cloud nodes needed for legitimate recovery and is approximately given by

$$N = D/R$$

Cloud computing has brought large-scale distributed computing to the masses. The possibility to rent a large number of standardized virtual servers for short amounts of time and that allow remote access changes the possibilities of a small organization or even an individual, bringing them tremendous compute power without investing into a proper IT infrastructure. The fact that this is a paid-for service brings additional benefits, as a cloud user can be forensically connected to any services used not only by user data and login information, but also by the money trail.

Nowadays, the number N of nodes is at least in the tens of thousands. Large clouds are now available to legitimate users. Today, (2012), Google and Yahoo

claim to use clouds with more than hundred of thousands nodes and Microsoft's Azure advertizes millions of nodes. An unauthorized recovery of the key is clearly possible with these resources, but the cloud service providers are well aware of the potential of their resources for criminal activity and protect themselves against this possibility. Additionally, using legitimate clouds leaves many traces behind that can be used to trace and convict an adversary.

A legitimate recovery needs to rent the resources of such a large cloud and is somewhat costly. The amount depends of course on D, R, and the rental costs of the cloud. The user chooses R according to a trade-off between the urgency of an eventual recovery and the costs. An example that we later discuss shows that a public cloud with 8000 nodes would cost about a couple of hundred dollars. These costs are by themselves a deterrent to an escrow service who wants to "precompute their users' need". An escrow service would certainly not spend these amounts of money on recovering all keys, but when reimbursed by the user, be willing to broker the recovery using a cloud service. We can speculate that giving an economic cost to key recovery would make "key insurance" possible. The user might then protect herself against key loss by buying insurance at a nominal cost.

Technically, an RENS scheme hides the key within a *noised* secret. Like the classical shared secret scheme by Shamir [Sha79], a noised secret consits of two shares at least and the secret is the exclusive-or (xor) of the shares. At least one of the shares is *"noised"*, which means that it is hidden in a large set – the *noise space*. The size M of the noise space is a parameter set by the user who in this way determines D and R, both linearly proportional to M. With overwhelming probability only the noised share reveals the secret. The RENS recovery procedure searches for the noised share through the noise space by brute force. It recognizes the noise share because it is given a secure hash of the noised share as a *hint*. Decryption by searching for the true share within the noise space might need to inspect all shares, and will on average be successful after inspecting half of them.

If we move the search to the cloud, we can speed it up by parallelizing it. Two schemes are possible. We can use a *static* scheme where the number of nodes is selected before the search begins. A *scalable* scheme changes the number of servers if necessary in order to meet the deadline. If the through-put of each server is the same, then a static scheme will achieve the smallest cloud size N. Otherwise, a scalable scheme needs to be used.

A static scheme with a cloud of 10,000 nodes provides a speed-up that changes seconds into days, as a day has 86,400 seconds and minutes into months as a 30-day month has 43,200 minutes.

We use classical secret sharing to prevent any information leakage through the use of the cloud. Only one share of the key's backup is ever recovered by the cloud and the other share is retained by the escrow service. An adversary needs to gain access to both shares in order to obtain the key.

In the rest of this article, we analyze the feasibility of RENS schemes. We define the schemes, discuss correctness, safety, and the properties that we just

outlined. We discuss related work in Section 2. Section 3 introduces the basic RENS scheme formally. The basic scheme assumes that the capacities of the nodes are approximately identical. We present a static scheme where the escrow service knows the capabilities of the nodes in advance. For instance, the escrow service rents hardware nodes from a cloud provider for a certain time. We then present a scheme that uses scalable partitioning where the nodes autonomously adjust their number to the task at hand. We present an optimization of this scheme that uses data from one additional node in order to lower the total number of nodes involved and hence the costs to the escrow server. We discuss the performance using simulation of an inhomogeneous cloud in Section 4. Our schemes so far does not give any assurance against finishing recovery early. We provide another extension in Section 5 to our basic idea that gives tight assurance for boundaries of the recovery time. For instance, we can guarantee with three nines assurance that the actual recovery time is between 1/2 and 1 of the maximum recovery time. At the end, we conclude and discuss future work.

2 Related Work

The risks of key escrow are hardly a new issue. Key escrow mandated by government was a hotly contested issue in the nineties in the United States. Much work has been devoted to define the legal, ethical, and technical issues and to design, prototype, and standardize key recovery mechanisms. The work by Bellare and Goldwasser [BG97], the work on the Clipper proposal by US government [MA96] [Bla11], the proposal by Verheul and van Tilborg [VvT97], and the risk evaluation by Abelson and colleagues [AAB+97] on the technical side, and the ethical and legal assessments by Denning and Baugh [DBJ96] and Singhal [Sin95] among many others show this interest. The concept of recoverable encryption was implicit in Denning's taxonomy [DB96] and became more explict in a revised version [DB97]. Of course, we are considering here voluntary key escrow so that much of this work and criticism simply does not apply.

Gennaro and colleagues describe a two-phase key recovery system that allows reusing a single asymetrical cryptography operation to generate key recovery data for various sessions and give it to a recovery agent [GKM+97]. Ando *et al.* exhibit a method that replaces a human recovery agent with an automatic one [AMK+01]. Johnson and colleages patented a key recovery scheme that is interoperable with existing systems [JKKJ+00]. Gupta provides interoperability by defining a common key recovery block [Gup00], a work extended by Chandersekaran and colleagues, who patented a method for achieving interoperability between key recovery enabled and unaware systems [CG02], [CMMV05]. Andrews, Huang, and Ruan distribute information in order to simplify access to private keys in a public key infrastructure without sacrificing security [AHR+05]. D'Souza and Pandey allow data to be stored in a cloud system where the data store can release encrypted data upon receiving a threshold number of requests from third parties. The scheme is based on verifiable secret sharing [DP12]. Fan *et al.* give an overview of the state of the art [FZZ12]. Current work on key escrow in the scientific literature tries to

avoid an unintended form of key escrow, where a public key generation system can reconstruct a client key [CS11].

We published the original Recoverable Encryption (RE) idea in 2010 [JLS10], where we applied it to data that a client encrypted and entrusted to the cloud. These data form an LH*$_{\mathrm{RE}}$ file distributed over the nodes in the cloud. As its name suggests, this is a Linear Hash (LH) based Scalable Distributed Data Structure [LNS96], [AMR+11]. The encryption key was maintained by the user but also backed up in the cloud structure itself. The backup is subjected to secret sharing and to recover it, one has to collect all the shares. An authorized client of the cloud can use the LH*$_{\mathrm{RE}}$ scan operation, but an intruder would have to break into typically many cloud nodes [JLS10], [SL10]. Whereas an LH*$_{\mathrm{RE}}$ backup key is stored in the cloud itself, RENS only uses the cloud for the recovery itself.

In CSCP [LJS11], we also store files encrypted by the client in the cloud, but in contrast to LH*$_{\mathrm{RE}}$ several users share keys among authorized clients. CSCP uses a static Diffie-Helman (DH) scheme. If a client looses her Diffie-Helman number, access to keys and files are lost, but an administrator has a backup of each private Diffie-Helman key. Obviously, RENS blends nicely with CSCP.

Our current proposal replaces the dispersion of the key into shares by a recovery scheme based on a targeted amount of computation. Whereas in previous schemes, the key was dispersed into a reasonably large number of shares, here, we only use two shares and allow access to one share through a limited computational effort. This concept has been made possible by the advent of "cloud computing" that puts large-scale distributed computing at the fingertips of the masses.

The concept of RE is rooted in the cryptographical concepts of *one-way hashes with trapdoor* and *cryptograms* or *crypto-puzzles* [DN93], [Cha11], [KRS+12]. RE can be considered to be a one-way hash where the computational capacity of cloud services for a distributed brute-force attack constitutes the trapdoor. RE in this sense is similar to Rivest's and Shamir's timed release crypto [RSW96], where a certain amount of computation needs to be performed in order to obtain a secret.

3 Recoverable Encryption through a Noised Secret

Recoverable encryption through a noised secret appears to the owner as the simple entrusting of the key in processed form to the escrow server, usually accompanied with some information for what the key is used. Upon request and after authentication and payment, the owner receives the key back from the escrow service after some processing time.

3.1 Client-Side Encryption

Before entrusting the backup of a key to the escrow service, the owner X preprocesses the key. The key is a bit-string of normal length (e.g. 256b for AES) that appears to be a random number.

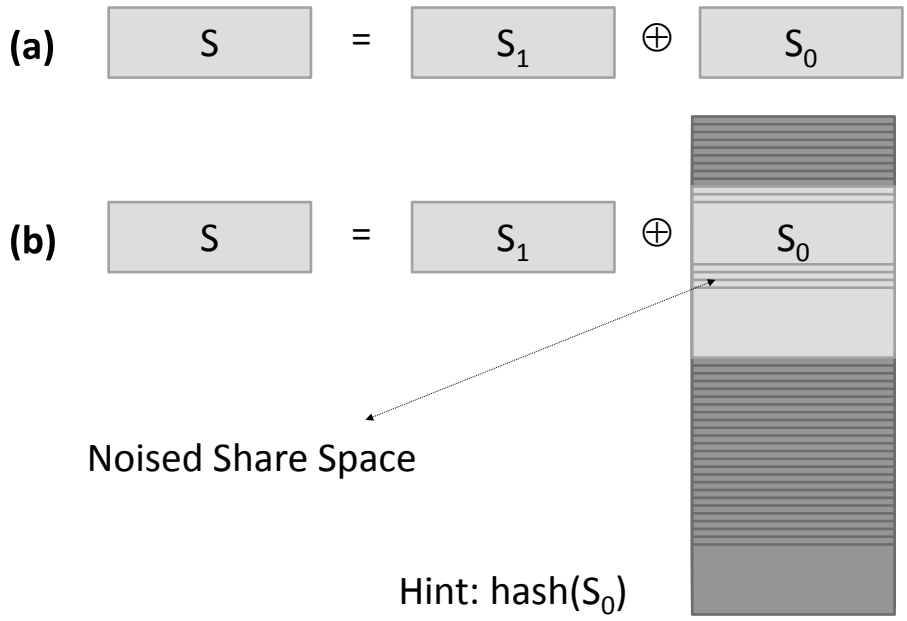


Fig. 1. Traditional secret sharing with two shares (a) and secret sharing with a noised secret (b)

```
import random

def create(S, M):
    S1 = random.getrandbits(KEYLENGTH)
    S0 = S1 xor S
    hashValue = hash(HASHALGO, S0)

    f = random.randint(0,M)
    l = int(S0) - f

    return S1, M, l, hashValue, HASHALGO

def recover(S1, M, l, hashValue, HASHALGO):
    for i in range(l, l+M):
        if hash(HASHALGO, i) == hashValue:
            S0 = i
    return S0 xor S1
```

Fig. 2. Pseudo-code for the creation of and the recovery from the noised secret

The owner uses classical secret sharing to write the key S as the exclusive or (xor) of two random strings of the same size as S:

$$S = S_1 \oplus S_0$$

The owner calculates the hash of S_0 using a standard, high-quality cryptographical hash method and stores $h(S_0)$ and a descriptor of the hash method as the *hint* $H(S)$ of the key. The owner chooses a size M of the noise space. As we will discuss, this parameter determines the average single-core recovery time D that represents the safety of the key backup. The owner creates a random number f in the interval $[0, M[$. The owner then converts S_0 from a bit string into an unsigned integer. She calculates $l = S_0 - f$. l forms the lower limit of the noise space that consists of the numbers l, $l+1$, ..., $l+M-1$. We call these numbers the *noise shares,* and refer to them collectively as the noise space. The true share S_0 is one of the noise shares and can be identified by the hint $h(S_0)$. Since we assume that the size of the hash is much larger than M, this is always possible with overwhelming probability. Figure 1 shows the procedure. The complete information given to the escrow service consists of S_1, M, l, and the hint $H(S)$.

We can still recover the original key S from this information. We iterate through the noise space starting with l and apply the hash to all noise shares. If we find one with the same hash as in the hint, we can assume that it is the true share S_0. We then recover the key as $S_1 \oplus S_0$. (Figure 2)

In order to protect against previously unknown vulnerabilities in the chosen hash method, we can choose an n-th power of a secure hash, i.e. calculate $h(S_0) = \phi^n(S_0)$ where ϕ is a NIST recommended standard hash function.

The owner uses the size M of the noise space in order to control the difficulty of the recovery operations. For this, she needs to have some reasonable estimate on the timing of the chosen hash function on a single-core processor together with a reasonable assumption on the number of cores that the escrow service or a bad employee of the escrow service might use. If she thinks that a reasonable number for the throughput of hash operations is T, then she obtains the maximum time D for recovery by the escrow using its own resources as

$$D = M/T$$

On average, an adversarial escrow service will use half that time to recover the noised share S_0 as the offset f of S_0 in the noise space was chosen randomly.

We need to be more carefull when we are using a private or public key created with one of the standard public key algorithms such as RSA, since the bits in such a key are highly redundant. It is known that an RSA key can be reconstructed from half of the bits [BM03, EJMDW05]. In case that we have a key that is not generated as a random bit string, we encrypt the key using a symmetric encryption method such as AES with a random key and then subject the latter key to our scheme. In this case, the usage information contains a description of the algorithm and the encryption of the original key.

3.2 Server Side Decryption

To recover a key, the escrow server has a share S_1, a size M of the noise space, and lower limit l of the noise space, and the hint $H(S)$, which contains the hash $h(S_0)$ of the noised share. The escrow server recovers the information using a brute-force attack, in which all elements of the noise space $l, l+1, \ldots, l+M-1$ are generated, their hash calculated, and compared with $h(S_0)$. With exceedingly high probability, there is only one share that has this hash value, namely the noised share S_0. The secret $S = S_0 \oplus S_1$ is returned.

The noise space is dimensioned so large that the server does not possibly have the means to perform this search with its own resources with any reasonable hope for success. It therefore needs to use a widely distributed computing service – a cloud service – in order to arme the recovery attempt. Brute force attacks are of course what is called "embarrassingly parallel" and can be easily partitioned into any number of sub-tasks that do not need to communicate amongst each other. If the server has Quality-of-Service (QoS) guarantees from the cloud provider, the easiest scheme is *static partitioning*, which we discuss first below. Otherwise, the server might use the principles of Scalable Distributed Data Structures (SDDS) [LNS96], (*scalable partitioning*), or a more involved interaction between a controller and participating working nodes. We describe a scalable partitioning scheme and two enhancements to deal with variations among node capacities below.

There is a (very) small chance for hash collisions, where there is more than one solution to $\text{hash}(X) = \text{hint}(= \text{hash}(S_0))$. A brute force attack will in general only returns the first solution found, which is not necessarily the true one. In this case, the escrow service will return a false key to the user. We assume that this becomes immediately obvious to the user who will complain to the escrow service. The escrow service will then repeat the search in an exhaustive manner, making sure to return all the possible solutions to $\text{hash}(X) = \text{hint}$. The probability of a collision is for a good hash close to the number of possible hashes divided by the size of the noise space. As good hashes have at least twenty bytes or one hundred and sixty bits, and as reasonable noise spaces do not have more than sixty bits, the chance for a hash collision is still in the order of 2^{-100} and probably much higher. If we want to protect against this already vanishingly small probability, we can do so at the costs of an additional hash. Since the changes necessary to switch to an exhaustive search are quite obvious, we do not consider this protection against the remote possibility of a hash collision in the following.

Example. A client wants to encrypt an AES key of length 512 bits. She wants D to be at least a month, i.e. 2^{22} seconds. She wants to be able to recover a key in minutes, leading her to set $R = 2^9$ seconds. Assume now that a node can make 2^{20} hash calculations per second. These numbers are reasonable in 2012 for a 2 GHz core processor, if we use SHA-256 as the hash. This gives us a noise space of 2^{20+22} or 2^{42} elements. Since the AES key is treated as an unsigned integer between 0 and 2^{512}, there is plenty of choice for the offset to an interval $I = [0, 2^{42}]$.

3.3 Decryption with Static Partitioning

If the server has guarantees for a minimum throughput of hash calculations at each node, the server determines the number of nodes necessary from the quality of service promise. If the maximal recovery time promised to the client is R, if a node can calculate at least T hashes per time unit, if N nodes are used, and if the size of the noise space is M then the ensemble can perform NT hashes per time unit. To evaluate a total of M hashes, it needs therefore M/NT time units, so that

$$\frac{M}{NT} \geq R$$

The minimum number of nodes needed is simply $M/(TR)$.

We can describe the algorithm using the popular map-reduce scheme. When the escrow server requests a cloud service, it deals directly only with one node, the *coordinator*. The coordinator calculates the number of *worker* nodes N. In the map-phase, the coordinator requests the N worker nodes and assigns them logical identification numbers 0, 1, ..., $N-1$. It also sends them the hint, the lower bound l of the noise space and the size M of the noise space.

In the reduce-phase, node a calculates the hash of the elements $l+a$, $l+a+N$, $l+a+2N$, ... and compares them with the hash of share S_0 contained in the hint. If it has found an element of the noise space with that hash, we assume that it has found S_0 and it sends a message with S_0 to the coordinator. If it has exhausted the search space, it sends a "terminated" message. Once the coordinator has received the result from one of the nodes, it will send a "stop" message to all nodes. A nodes that receives this message simply obeys.

In the termination phase, the coordinator sends the found string to the escrow server. This string is only one of the two shares, so the cloud itself has no information about the key. The escrow server now combines the two shares to obtain the key to return to the user.

Example Continued. Since $R = 2^9$ sec and $D = 2^{22}$ sec, $N = 2^{13} = 8192$. If we can rent a dedicated server core per hour at a cost of US$0.50 (November 2012), we would spend US$512.00 for an hour. If we can negotiate to pay for only part of the hour, the costs could sink to US$60.00 for the maximum time needed for recovery.

3.4 Recovery with Scalable Partitioning

For *scalable* or dynamic partitioning, we use the principles of Scalable Distributed Data Structures (SDDS) design to adjust the number of servers to the capabilities of the nodes. We assume that a node can reliably assess the throughput it can deliver for the time of the calculation. In order to distribute the work scalably and dynamically, *any* algorithm needs to make decisions based on the capabilities of only relatively few nodes. In this section, we present an algorithm where nodes make a decision on a split only based on their state. In the next section, Section 3.5, we provide two enhancements that use capacity information on the new node.

Our performance results (Section 4) show that they yield better performance measured in terms of the ratio of total capacity over total load. A smaller ratio means less nodes involved and hence less money paid to the cloud provider.

The scalable schemes go through the same phases as static partitioning. In the *initialization phase*, the escrow server selects a single cloud node (with index 0). The *map phase* immediately follows. Starting with the original node, each node compares its capacity with the task assigned to it and decides whether it needs to *split*, that is, request a new node from the cloud and share its workload with it. In the process of splits, each node acquires two parameters, its logical identifier and its level, that we use for the workload distribution. In this basic scheme, nodes only use local information in order to decide whether to split.

At the beginning of the mapping phase, Node 0 calculates its throughput capability B_0 given its current load. This throughput calculation is repeated at each node used in the recovery procedure. The node has a number n of hashes to calculate, a maximum time R to perform all of these calculations, and calculates a rate τ of calculations. A node then calculates its *load factor* $\alpha = \tau n/R$. If $\alpha > 1$, then the node is overloaded. If the initial node 0 has $\alpha \leq 1$, it is capable of doing the whole calculation itself, which it does and then returns the result to the escrow service. In the much more likely opposite case, Node 0 requests a new node from the cloud service provider, which becomes Node 1. The noise interval is divided into two equal halves and each half is assigned to one of the two nodes. Both new nodes acquire a new *level* $j = 1$.

Each node calculates its load factor α. If the load factor is larger than one (the node is overloaded), it *splits*. A split effectively divides the work assigned to the node between that node and a new node. Thus, each split operation requests a new node from the cloud server and incorporates it into the system. If node i with level j has split, then the node increases its level to $j + 1$, and the new node receives number $i + 2^j$ and level $j + 1$.

We recall that the noise space starts with number l. The node with identity number n and level j calculates the hashes $l + x$, where $x \equiv n \bmod 2^j$ and $0 \leq x < M$. LH* addressing [LNS96] guarantees that element in the noise space is assigned to exactly one node.

As in the static scheme, a node that finds a solution and therefore with overwhelming probability the noised share S_0 sends its find to Node 0. This constitutes the reduce phase. In the *termination phase*, Node 0 asks all other nodes to stop. It does so by sending the stop message to all nodes that split from it, i.e. to Nodes 1, 2, , 4, 8, ... Each node that receives the stop message, forwards its to all nodes that have split from it. The number of messages that a node has to send or forward is lmited by its level and therefore logarithmic in the number of nodes.

Example. We assume a very small example with nodes of largely varying capacity. Node 0 receives a workload of 15000 hashes and estimates that it can calculate 10000 hashes. Therefore, its workload factor α is 1.5 or 150%. It therefore splits. The new node has logical address $1 = 0 + 2^0$ and both nodes have level 1. Node 1 estimates that it can calculate 2000 hashes and has therefore a

load factor $\alpha = 3.75$, while the load factor at Node 0 has been halved to .75. Node 0 therefore stops splitting, but Node 1 will have do, claiming a new node with logical address $3 = 1 + 2^1$. Node 3 decides that it can handel 11000 hashes and has therefore a load factor of 0.295, whereas Node 1 has a load factor of 1.875. Therefore, Node 1 splits once more, requesting a new Node with identity number $5 = 1 + 2^2$. Its load sinks to 1625 hashes and its new load factor is .8125. If the new node 5 can handle 9000 hashes, then its load factor is 0.181, so that there are no more splits.

We now have a total of four nodes. Node 0 has level 1, Node 1 has level 3, Node 3 has level 2, and Node 5 has level 3. Assume that $l = 1000000$, so that the noise space is $[1000000, 1015000[$. Node 0 calculates the hashes of all even numbers, i.e. 1000000, 1000002, 10000004, ..., 1014998, using an increment of 2^1, since it has level 1. Node 1 has level 3, therefore an increment of 8, and calculates 1000001, 1000009, 1000017, ... Node 3 has level 2 and therefore an increment of 4, so that it calculates 1000003, 1000007, ... Node 5 has level 3, an increment of 8, and calculates 1000005, 1000013, ...

3.5 Scalable Partitioning with Limited Load Balancing

To scale well, scalable partitioning needs to minimize the interchange of information between nodes. In real life instances, the load factor of the initial node is several tens of orders of magnitude larger. For example, a scheme where the coordinator polls potential nodes for their capacity in order to use an optimal assignment is completely out of the question. In the current scalable partitioning scheme, decision on splits are made based on information only at the level of a single node. A good solution will have to balance the speed of making decisions only at the local level with the overprovisioning caused by variations in the capacity of the nodes. In the previous example, the problems stem from Node 1, which has only one fifth of the capacity of the initial load. If Node 0 and Node 3 would have been used at their full capacity, the incorporation of Node 5 would have become superfluous.

Besides allowing limited communication between nodes, we also need to change to a more flexible assignment of load. We now use a type of range partitioning to assign loads. Now nodes calculate the hashes of a contiguous range of numbers $[x_0, x_1[$ of numbers within the noise interval $[l, l + M[$. If node p with an assignment of $[x_0, x_1[$ splits, it decides on a cutoff point p_1 and assigns itself the workload $[x_0, p_1[$ and to the new node the interval $[p_1, x_1[$. During the first phase of the map-phase, p_1 will be the midpoint $\lfloor (x_0 + x_1)/2 \rfloor$. Our enhancements have the splitting node use the capacity of the new node when calculating p_1.

We only investigate here two enhancements of the scheme where during a split the new node sends the information about its capacity to the splitting node. Our first strategy has the splitting node p detect if the capacity of the new node n and its own capacity suffice to perform the work assigned currently to p. For example, if node p has a capacity of 0.8 in order to do work 1.8, it has to split. If the new node has capacity 1.2, the combined capacity of 2.0 is sufficient to do the work. However, if we distribute the work equally, p will have work of 0.9

assigned to it, and will have to split again, whereas Node n has spare capacity. In the first improvement strategy, node p will get 0.8 work and n will get 1.0 work.

Our second, additional strategy has a node decide whether the load distribution is getting close to achieve its goal. If node p has a capacity c_p and a currently assigned workload of $w < 3 \cdot p$, it will split, but assign to itself only the work that is within its capacity. The new node is likely to have to split itself, but probably (though not for sure) no more than once. Our full enhancement uses both strategies, but can be obviously expanded by interchanging information between more nodes. We have to leave the exploration of these issues to future work.

Example (Continued). If we use the full enhancement in the previous example, then Node 0 communicates with Node 1 in order to obtain its capacity. Since the combined capacity of both nodes is 12000 and the total load is 15000, the first strategy is not employed. However, since the capacity of Node 0 is within "striking distance" of the load, it assigns itself 10000 hashes (the numbers in $[1000000, 1010000[$) and the remainder (the numbers in $[1010000, 1015000[$) to Node 1. The load factor of Node 1 after this split is $5000/2000 = 2.5$ and it still has to split. Since the capacity of the new node, Node 3, is 11000, the combined capacity of Nodes 1 and 3 is sufficient. Therefore, Node 1 splits its load at a ratio of 2 : 11. It therefore assigns to itself the interval $[1010000, 1010769[$ and to Node 3 the interval $[1010769, 1015000[$. In this case, more extensive communication between Nodes 0, 1, and 3 could have let to a more balanced distribution, but not employed less nodes. The total capacity of the three nodes is 23000, so that we still overprovide. A more sophisticated scheme could have liberated Node 1, since its potential contribution is not only marginal, but also unnecessary.

4 Performance Analysis

Static partitioning always yields the best utilization of cloud nodes, but assumes that the throughputs at all nodes are perfectly even and known at the beginning of a run.

Scalable partitioning allows nodes to have different capacities, and detects these capacities whenever a new node is added. If nodes have all the same capacities, then a node will be split and its load divided by two until the load is less than 1 times the capacity of a node. If the total load is l times the node capacity, then Node 0 is split $\lfloor \log_2(l) \rfloor + 1$ times. This gives us the ratio of total capacity over total load to be

$$2^{\lfloor \log_2(l) \rfloor + 1} / l$$

This functions oscillates between 1.0 and 2.0 as Figure 3 shows. The average ratio is $\log(2) = 1.38629$ and is the price we pay for scalability.

If the capacity of the nodes is not constant but instead is subject to a non-constant probability distribution, then a different picture emerges. We assumed

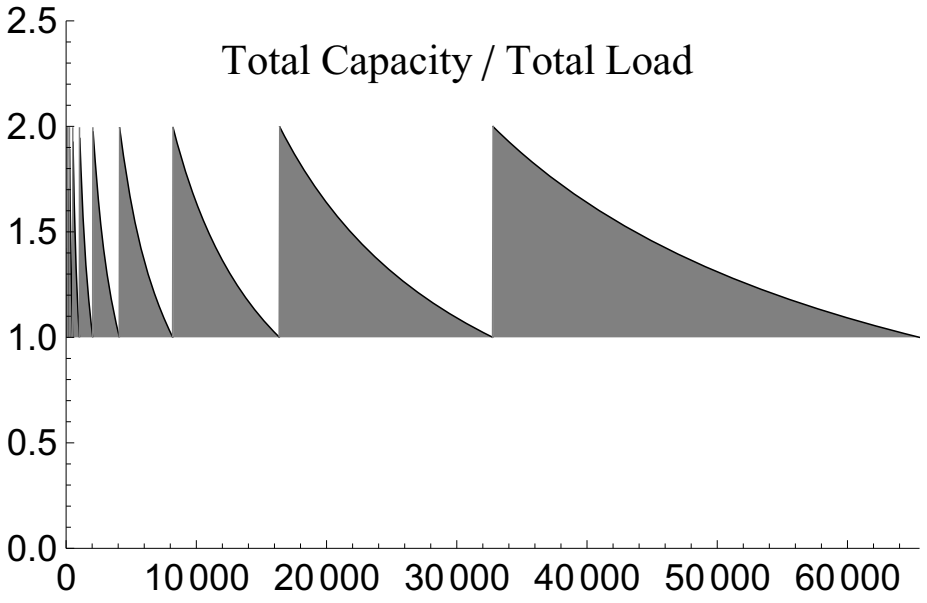


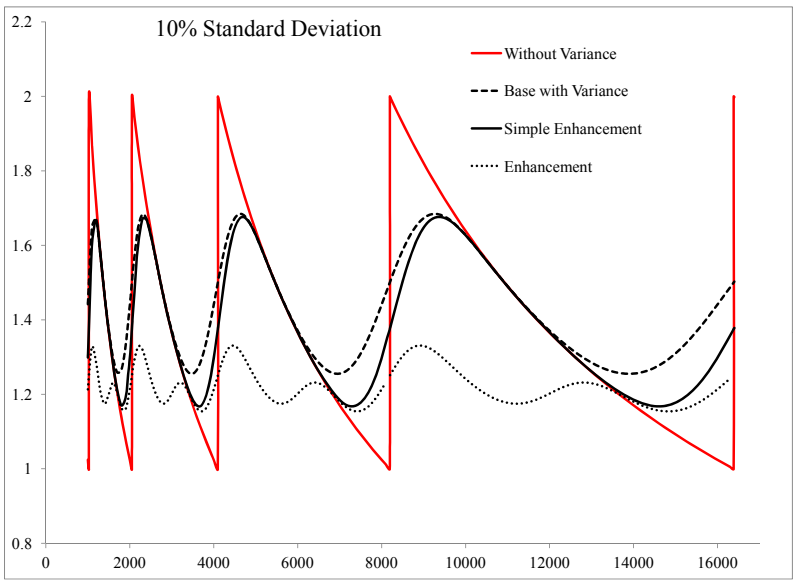
Fig. 3. Ratio of total capacity over total load with identical capacity at each node

first that the capacity of the node is normally distributed around l times the node capacity with different standard deviation and simulated the ratio. The simulation is accurate to three or four digital digits. The result of our simulation is given in Figures 4 and 5, where the standard deviation is 10%, 25%, 33%, and 50%.

The simple enhancement (as discussed previously) determines if a splitting node and the new node together have the capacity to perform the assigned task. In this case, the task is split according to capacity. Otherwise, the task is split evenly among the splitting and new node. In this case, at least one of them has to split.

The enhancement (as also discussed previously) includes the simple enhancement. If this is not the case, but if the assigned load is within three times its capacity, then the splitting node assign to itself all the load it can handle and passes the rest of the load to the new node. The assumption is that frequently, the new node will only have to split once.

We first observe that the basic variant now performs more consistently than without variation in the node capabilities. If the standard deviation is small, it exhibits overprovisioning close to the expected rate. However, the ratio of total capacity over total load for the basic scalable partitioning scheme increases with increased standard deviation. For 50% standard deviation, its ratio is consistently higher than 2. (In our simulation, we used a minimum capacity of 1/100 so that the probability distribution is strictly speaking no longer normally distributed, which would allow for negative capacities. As the standard deviation increases, the mean capacity therefore slightly increases as well.) With increasing deviation, the oscillations become much less pronounced. The simple enhancement shows visible improvements with all standard deviations, but for 10% standard deviation only in the dips of the curve. The full enhancement



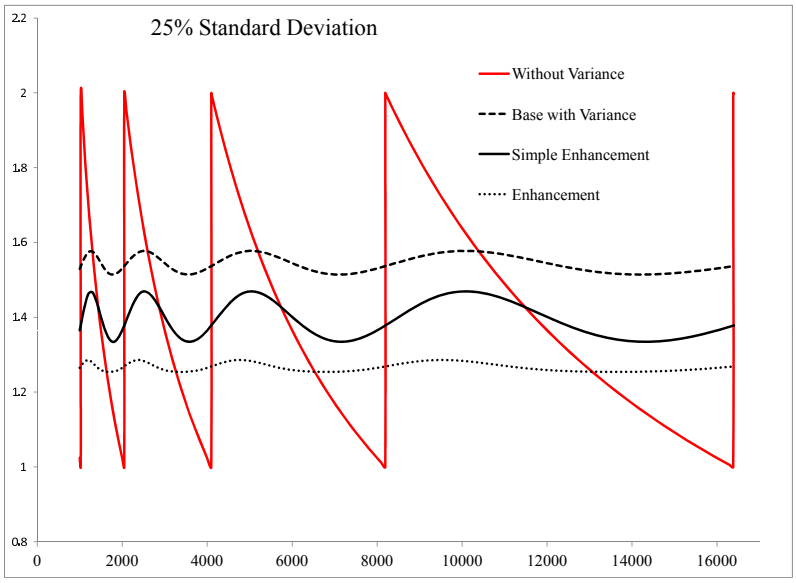


Fig. 4. Ratio of total capacity over total load depending on the load given in terms of expected node capability, using scalable partitioning without variation,scalable partitioning with normally distributed node capacity with standard deviations of 10% and 25% and with two of our extensions

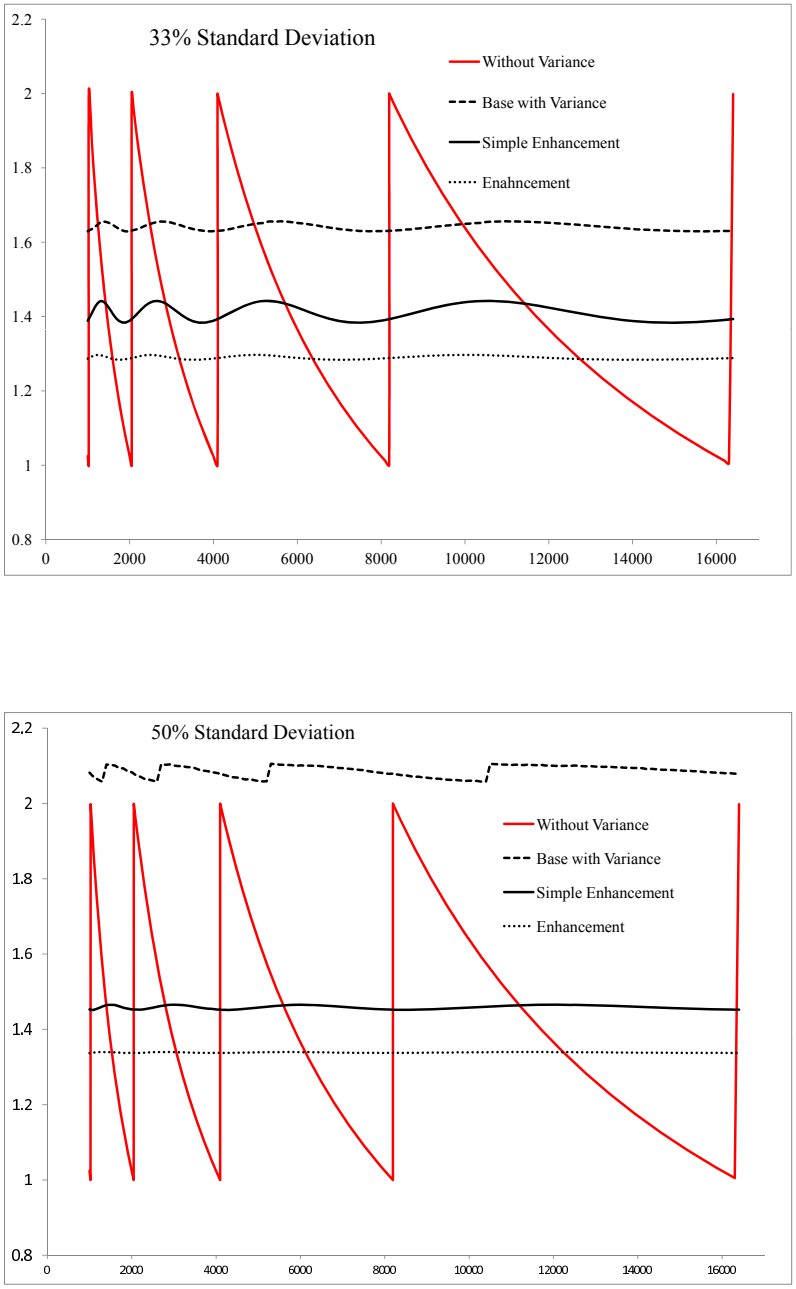


Fig. 5. Ratio of total capacity over total load depending on the load given in terms of expected node capability, using scalable partitioning without variation, scalable partitioning with normally distributed node capacity with standard deviations of 33% and 50% and with two of our extensions

Table 1. Average values of total capacity over total load ratios

Standard Deviation	Average Total Capacity over Total Load
Base with Variance	
10%	1.438
25%	1.542
33%	1.641
50%	2.086
Weibull 50%	1.856
Gamma 50%	2.170
Simple Extension	
10%	1.382
25%	1.393
33%	1.409
50%	1.458
Weibull 50%	1.478
Gamma 50%	1.598
Extension	
10%	1.219
25%	1.266
33%	1.289
50%	1.339
Weibull 50%	1.357
Gamma 50%	1.465

shows continuous improvements over the base and the simple enhancement. We notice however that the average increases slightly as is shown in Table 1.

When we simulated a scenario where the capacity of the nodes follows a different distribution, namely a gamma distribution with mean 1.0 and standard deviation of 50% and a Weibull distribution with the same mean and distribution, we found that the average ratio of total capacity over load was close to being constant, not depending on the total load. As was to be expected, the distribution is a major factor in the ratio. However, the benefits of the two extensions considered were equally obvious, though in the case of the Weibull distribution to a slightly lesser degree.

We show the effects of varying the standard deviation in Figures 6 and 7, which shows that the use of *local* capacity information when distributing small remaining load among few nodes is beneficial. The enhancements to the protocol do better in the case of the gamma distribution, since the gamma distribution (a convolution of the exponential distribution) has more small capacity nodes. We should note that our choice of probability distributions serve just as a stand-in for the unknown distribution. Much more research and measurements are necessary in this area.

The basic idea of exchanging information in the final phase of mapping does not violate the principles of scalability. In these scenarios, a node in the mapping phase enters a *final assignment* state whenever its assigned load is within c times its capacity, where c is a relatively small number. In this state, the node recruits

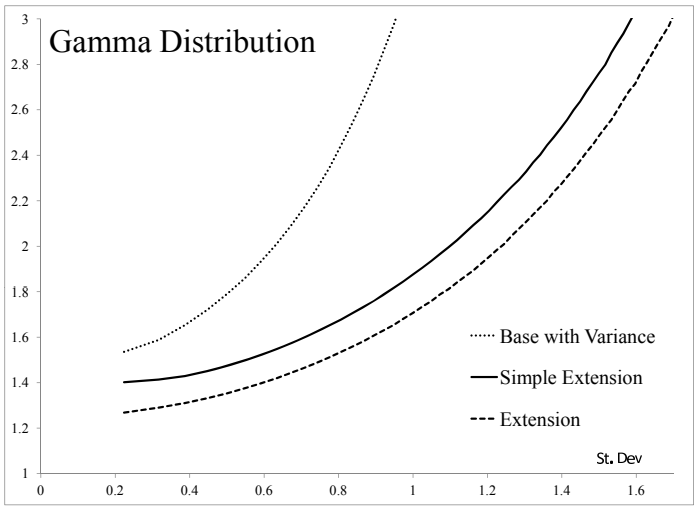


Fig. 6. Ratio of total capacity over total load in dependence on the standard deviation

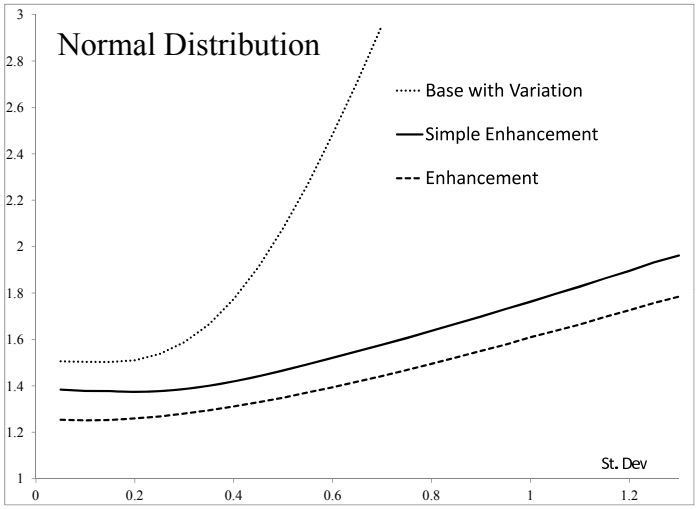


Fig. 7. Ratio of total capacity over total load in dependence on the standard deviation

new nodes one by one until there are enough nodes left to deal with the workload. In the worst case, this method leaves the last recruited node with only a marginal workload. In expectation, the number of nodes recruited would be between c and $c + 1$, so that we can estimate a reasonable upper bound on the load factor to be $1 + 1/c$.

5 Multiple Noises

In our scheme, the worst-time recovery at the esrow service is R, but the best possible time is negligible, since the very first hash calculated might yield the noised share. Some users averse to gambling might find this prospect discomforting. For this group, we present now a solution that gives guarantees against obtaining the backup of the key too quickly.

The chance to obtain the noised share within time ρR (where R is the maximum time) is equal to ρ. It is well known, that the last of n uniformly distributed tasks has a much smaller spread. In our case, this leads to *multiple noising*. There, we require the escrow service to use brute force in the cloud to invert n hashes.

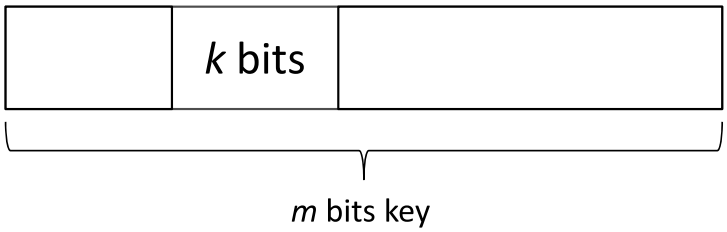


Fig. 8. Selection of noised part of key for multiple noises

Assume that we want a maximum of 2^k hashes to be calculated, that the key length is $m > k$, and that we want to invert n hashes. We recall that our scheme splits the key S into two different shares S_0 and S_1 of the same length and that share S_0 is being noised. We select k among the m bit positions in the key. Figure 8 shows a selection of k contiguous bits. The share S_0 is the concatenation of the selected bits and the $m - k$ remaining bits. We write this concatenation as $S_0 = I \sqcup R$, where I is made up of the k selected bits and R of the remaining bits. We now use classical secret sharing writing $I = I_1 \oplus I_2 \oplus \ldots I_n$, where the I-shares I_1, I_2, $\ldots$ I_n are random bit strings. We calculate the hashes $H_\nu = h(I_\nu \sqcup R)$ as the core part of n hints. (The remainder of the hints contains information about the hash selected and the length k of I and $m - k$ of R.)

For server side decryption, the escrow service uses a cloud to solve in parallel

$$h(X \sqcup R) = H_\nu \qquad \nu \in \{1, 2, \ldots, n\}$$

After it has found all solutions J_1, J_2, $\ldots$ J_n, the share S_0 is calculated as

$$S_0 = (J_1 \oplus J_2 \oplus \ldots \oplus J_n)) \sqcup R$$

This calculation terminates after the last of the n equations has been solved.

The expected time to recover the key is the expected time to recover the last of the n shares J_1, J_2, $\ldots$ J_n. We normalize the maximum recovery time to 1. Let the random variable X_i represent the time to recover share J_i. The

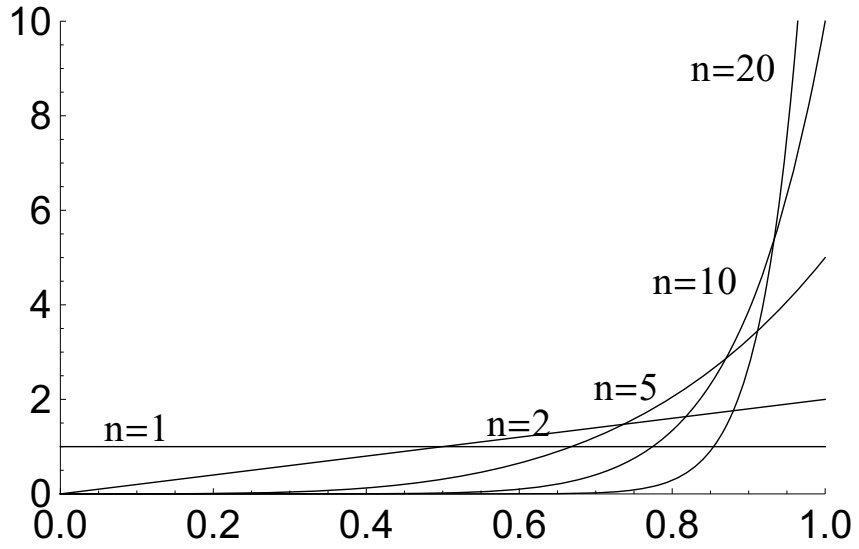


Fig. 9. Probability density for the maximum of m uniformly distributed random variables in $[0, 1]$

Table 2. Three and six nines guarantees that the last of n hashes is inverted in no less than p

n	p (three nines)	p (six nines)	Expected Value
1	0.001	10^{-6}	0.500
3	0.100	0.010	0.750
5	0.252	0.063	0.833
10	0.501	0.251	0.909
20	0.708	0.501	0.952

random variables are identically and independently distributed. The cumulative distribution function for the time to recover the last of the n shares of S_0 is then

$$F(x) = \text{Prob}(\max(X_i) < x) = \text{Prob}({X_1}^n < x) = x^n$$

The probability density function of recovering all n shares is thus given by nt^{n-1} (Figure 9). Consequentially, the probability of key recovery in an exceedingly short time is made very low. The mean time to recovery is then $n/(n + 1$.

We can also give minimum time guarantee at a certain assurance level a such as $a = 0.999$ (three nines) or $a = 0.999999$ (six nines. This is defined to be the time value x_0 such that $\text{Prob}(\max(X_i) < x_0) = a$, i.e. that with probability (at least) a, the recovery takes more than x_0 times the maximum recovery time. Table 2 gives us the guarantees. For example, if we choose a safety of six nines, then we know at this level of assurance that the last of 20 shares will be recovered in less than 0.501 or 50% of the maximum recovery time.

6 Security

The security of our scheme is measured by the inability of an attacker to recover a key entrusted to the escrow service. An attacker outside of the escrow service needs to obtain both shares S_1 and S_0 of the key. This is only possible by breaking into the escrow server and becoming effectively an insider. We can therefore restricts ourselves to an attacker who is an insider. In this case, we have to assume that the attacker can break through internal defenses and obtain S_1 and the hint $h(S_0)$ of the noised secret. The insider then has to invert the hash function in order to obtain S_0.

We can systematically list the possibilities:

It is possible that there is no need to invert the hash function. As already mentioned in Section 3.1, RSA keys can be reconstructed from about half of the bits [BM03, EJMDW05]. If the scheme would be applied to keys that cannot be assumed to be random bits, then the specification of the noise space could be sufficient to generate a single or a moderate number of candidate keys just from the knowledge of the noise space. The insider attacker can then easily recover S_0 and therefore the key.

The inversion of the hash in the noise space could be much simpler than assumed. Cryptography is full of examples of more and more powerful attacks, as the history of MD5 and WEP show. In addition, the computational power of a single machine has increased exponentially at a high rate since the beginning of computing. The introduction of more powerful Graphical Processing Units (GPU) [OHL+08, KO12], has lead to a one-time jump in the capabilities of relatively cheap servers. It is certainly feasible that GPU computing can enter the world of for-hire cloud computing. Even if this is not the case, then competition, better energy use, and server development should lower the costs of computation steadily. This is a real problem for our scheme, but shares it with much of cryptographical methods. Just as for example key length has to be steadily increased, so the size of our noise spaces needs to be increased in order to maintain the same degree of security. Only, our system has to be more finely tuned as we cannot err on the side of exaggerated security. A developer worried about computational attack on a certain cryptographical scheme can always double the key size "just to be sure" and the product will only show a slight deterioration in response time due to the more involved cryptographical calculations. In our scheme, this is not an option. On the positive side, there is no new jump in sight that would increase single machine capability as the introduction of GPU computing did, and this one came with ample warning. Second, the times of the validity of Moore's law seem to be over, as single CPU performance cannot be further increased without incurring an intolerable energy use. The new form of Moore's law will be a steady decline in the decrease of the costs of single CPU computation. Overall, the managerial challenges of decreasing computation costs seem to be quite solvable.

Finally, the insider attacker could use the recovery scheme itself, availing herself of anonimity servers and untraceable credit cards, as are sold in many countries for use as gifts. This is a known problem for law enforcement as spammers

can easily use the untraceability of credit cards in order to set up fraudulent websites. However, any commercial service that accepts this type of untraceable credit card opens itself up to charges of aiding and abetting and at least of gross negligence. When we are assessing these type of dangers, we need to be realistic. Technology such as cryptography only defines quasi-absolute security, but assuming a certain social ambience. If I want to read my boss's letters, I have certainly the technical tools to open an envelope (apparently hot water steam is sufficient), read the letter, and use a simple adhesive to close the letter. But even if I were inclined to do so, the social risk is inacceptable. In our case, an insider or an outside attacker that has penetrated the escrow service would have to undertake an additional step with a high likelihood of leaving traces. People concerned with security in organizations at high and continued risk know that adversaries usually resort to out-of-band methods. West-German ministries in the eighties were leaking secret information like sieves not because of technical faults but because of Romeo-attacks, specially trained STASI agents seducing well-placed secretaries.

7 Conclusions

We have introduced a new password recovery scheme based on an escrow service. Unlike other escrow based schemes, in our scheme the user knows that the escrow server will not peek at the data entrusted to it, as it would cost too much. Our scheme is based on a novel idea of using the scalable and affordable power of cloud computing as a back door for cryptography. Recoverable encryption could even be considered a new form of cryptography.

The relatively new paradigm of cloud computing still has to solve questions such as reliable quality of service guarantees and protection against node failures. In our setting, ignoring the issue is a reasonable strategy, since the only node that matters (*ex post facto*) is the one that will find the noised share. The expected behavior of recovery is hence the one of that single server and the quality of service of that single server is the one experienced by the end-user. However, our discussion on how to distribute an embarrassingly parallel workload in a cloud with nodes of varying capacity should apply to other problems. In this case, scalable fault-resilience does become an interesting issue. For instance, cloud virtual machines can suffer capacity fluctuations because of collocated virtual machines. We plan to investigate these issues in future work.

References

[AAB+97] Abelson, H., Anderson, R., Bellovin, S.M., Benaloh, J., Blaze, M., Diffie, W., Gilmore, J., Neumann, P.G., Rivest, R.L., Schiller, J.I., Schneier, B.: The risks of key recovery, key escrow, and trusted third-party encryption. World Wide Web Journal 2(3), 241–257 (1997)

[AHR+05] Andrews, R.F., Huang, Z., Ruan, T.Q.X., et al.: Method and system of securely escrowing private keys in a public key infrastructure. US Patent 6,931,133 (August 2005)

[AMK+01] Ando, H., Morita, I., Kuroda, Y., Torii, N., Yamazaki, M., Miyauchi, H., Sako, K., Domyo, S., Tsuchiya, H., Kanno, S., et al.: Key recovery system. US Patent 6,185,308 (February 6, 2001)

[AMR+11] Abiteboul, S., Manolescu, I., Rigaux, P., Rousset, M.C., Senellart, P.: Web data management. Cambridge University Press (2011)

[BG97] Bellare, M., Goldwasser, S.: Verifiable partial key escrow. In: Proceedings of the 4th ACM Conference on Computer and Communications Security, pp. 78–91. ACM (1997)

[Bla11] Blaze, M.: Key escrow from a safe distance: looking back at the clipper chip. In: Proceedings of the 27th Annual Computer Security Applications Conference, pp. 317–321. ACM (2011)

[BM03] Blömer, J., May, A.: New partial key exposure attacks on RSA. In: Boneh, D. (ed.) CRYPTO 2003. LNCS, vol. 2729, pp. 27–43. Springer, Heidelberg (2003)

[CG02] Chandersekaran, S., Gupta, S.: Framework-based cryptographic key recovery system. US Patent 6,335,972 (January 1, 2002)

[Cha11] Chandrasekhar, S.: Construction of Efficient Authentication Schemes Using Trapdoor Hash Functions. PhD thesis, University of Kentucky (2011)

[CMMV05] Chandersekaran, S., Malik, S., Muresan, M., Vasudevan, N.: Apparatus, method, and computer program product for achieving interoperability between cryptographic key recovery enabled and unaware systems. US Patent 6,877,092 (April 5, 2005)

[CS11] Chatterjee, S., Sarkar, P.: Avoiding key escrow. In: Identity-Based Encryption, pp. 155–161. Springer (2011)

[DB96] Denning, D.E., Branstad, D.K.: A taxonomy for key escrow encryption systems. Communications of the ACM 39(3), 35 (1996)

[DB97] Denning, D.E., Branstad, D.K.: A taxonomy for key escrow encryption systems (1997), `faculty.nps.edu/dedennin/publications/TaxonomyKeyRecovery.htm`

[DBJ96] Denning, D.E., Baugh Jr., W.E.: Key escrow encryption policies and technologies. Villanova Law Review 41, 289 (1996)

[DN93] Dwork, C., Naor, M.: Pricing via processing or combatting junk mail. In: Brickell, E.F. (ed.) CRYPTO 1992. LNCS, vol. 740, pp. 139–147. Springer, Heidelberg (1993)

[DP12] D'Souza, R.P., Pandey, O.: Cloud key escrow system. US Patent 20,120,321,086 (December 20, 2012)

[EJMDW05] Ernst, M., Jochemsz, E., May, A., de Weger, B.: Partial key exposure attacks on RSA up to full size exponents. In: Cramer, R. (ed.) EUROCRYPT 2005. LNCS, vol. 3494, pp. 371–386. Springer, Heidelberg (2005)

[FZZ12] Fan, Q., Zhang, M., Zhang, Y.: Key escrow attack risk and preventive measures. Research Journal of Applied Sciences 4 (2012)

[GKM+97] Gennaro, R., Karger, P., Matyas, S., Peyravian, M., Roginsky, A., Safford, D., Willett, M., Zunic, N.: Two-phase cryptographic key recovery system. Computers & Security 16(6), 481–506 (1997)

[Gup00] Gupta, S.: A common key recovery block format: Promoting interoperability between dissimilar key recovery mechanisms. Computers & Security 19(1), 41–47 (2000)

[JKKJ+00] Johnson, D.B., Karger, P.A., Kaufman Jr., C.W., Matyas Jr., S.M., Safford, D.R., Yung, M.M., Zunic, N.: Interoperable cryptographic key recovery system with verification by comparison. US Patent 6,052,469 (April 18, 2000)

[JLS10] Jajodia, S., Litwin, W., Schwarz, T.: LH*RE: A scalable distributed data structure with recoverable encryption. In: CLOUD 2010: Proceedings of the 2010 IEEE 3rd International Conference on Cloud Computing, pp. 354–361. IEEE Computer Society, Washington, DC (2010)

[KO12] Komura, Y., Okabe, Y.: Gpu-based single-cluster algorithm for the simulation of the ising model. Journal of Computational Physics 231(4), 1209–1215 (2012)

[KRS+12] Kuppusamy, L., Rangasamy, J., Stebila, D., Boyd, C., Nieto, J.G.: Practical client puzzles in the standard model. In: Proceedings of the 7th ACM Symposium on Information, Computer and Communications Security, ASIACCS 2012. ACM, New York (2012)

[LJS11] Litwin, W., Jajodia, S., Schwarz, T.: Privacy of data outsourced to a cloud for selected readers through client-side encryption. In: WPES 2011: Proceedings of the 10th Annual ACM Workshop on Privacy in the Electronic Society, pp. 171–176. ACM, New York (2011)

[LNS96] Litwin, W., Neimat, M.A., Schneider, D.A.: Lh* – a scalable, distributed data structure. ACM Transactions on Database Systems (TODS) 21(4), 480–525 (1996)

[MA96] McConnell, B.W., Appel, E.J.: Enabling privacy, commerce, security and public safety in the global information infrastructure. Office of Management and Budget, Interagency Working Group on Cryptography Policy, Washington, DC (1996)

[MLFR02] Miller, E.L., Long, D.D.E., Freeman, W.E., Reed, B.C.: Strong security for network-attached storage. In: Proceedings of the 1st USENIX Conference on File and Storage Technologies, p. 1. USENIX Association (2002)

[OHL+08] Owens, J.D., Houston, M., Luebke, D., Green, S., Stone, J.E., Phillips, J.C.: Gpu computing. Proceedings of the IEEE 96(5), 879–899 (2008)

[RSW96] Rivest, R.L., Shamir, A., Wagner, D.A.: Time-lock puzzles and timed-release crypto. Technical report, Massachusetts Institute of Technology, Cambridge, MA, USA (1996)

[Sha79] Shamir, A.: How to share a secret. Communications of the ACM 22(11), 612–613 (1979)

[Sin95] Singhal, A.: The piracy of privacy-a fourth amendment analysis of key escrow cryptography. Stanford Law and Policy Review 7, 189 (1995)

[SL10] Schwarz, T., Long, D.D.E.: Clasas: a key-store for the cloud. In: 2010 IEEE International Symposium on Modeling, Analysis & Simulation of Computer and Telecommunication Systems (MASCOTS), pp. 267–276. IEEE (2010)

[VvT97] Verheul, E.R., van Tilborg, H.C.A.: Binding ElGamal: A fraud-detectable alternative to key-escrow proposals. In: Fumy, W. (ed.) EUROCRYPT 1997. LNCS, vol. 1233, pp. 119–133. Springer, Heidelberg (1997)