ISSN 2394-3777 (Print)
ISSN 2394-3785 (Online)
Available online at www.ijartet.com

International Journal of Advanced Research Trends in Engineering and Technology (IJARTET)
Vol. 4, Special Issue 5, March 2017

# Enabling Data Integrity Protection in Regenerating-Coding-Based Cloud Storage

**S.Ayyappan.**
**AP / MCA**
**SVCET-Puliangudi**
samratayyappa@gmail.com

**Abstract -To protect outsourced data in cloud storage against corruptions, enabling integrity protection, fault tolerance, and efficient recovery for cloud storage becomes critical. Regenerating codes provide fault tolerance by striping data across multiple servers, while using less repair traffic than traditional erasure codes during failure recovery. We design and implement a practical data integrity protection (DIP) scheme for a specific regenerating code, while preserving theintrinsic properties of fault tolerance and repair traffic saving. Our DIP scheme is designed under a Byzantine adversarial model, and enables a client to feasibly verify the integrity of random subsets of outsourced data againstgeneralormaliciouscorruptions.Itworksunder the simple assumption of thin-cloud storage and allows different parameters to be fine-tuned for the performance-security trade-off. We implement and evaluate the overhead of our DIP scheme in a real cloud storage tested under different parameter choices. The project tend to style and implement a sensible information of Data responsibleness security (DRS). To tendmoreanalyzethesafetystrengthsofourDRStheme via mathematical models. To show that remote integrity checking are often presumably incorporated into make codes in sensibleoperation.**

***Index Terms*—remote data checking, secure and trusted storage systems, implementation, experimentation**

## I.INTRODUCTION

Cloud storage offers an on-demand data outsourcing service model, and is gaining popularity due to its elasticity and low maintenance cost. However, security concerns arise when data storageis outsourced to third-party cloud storage providers. It is desirable to enable cloud clients to verify the *integrity* oftheiroutsourceddatainthecloud.Onemajoruseof cloud storage is *long-term archival*, which represents a workload that is written once and rarely read. While the stored data is rarely read, it remains necessary to ensureitsintegrityfordisasterrecoveryorcompliance with legal requirements. Whole file checking in 2 process*Proofofretrievability*(POR)and*proofofdata possession* (PDP). Integrity of a large fileby spot-checking only a fraction of the file via various cryptographic primitives.

Suppose that we outsource storage to a *server*,whichcouldbeastoragesiteoracloudstorage provider. If we detect corruptions in our outsourced data (e.g., when a server crashes or is compromised).Then we should *repair* the corrupted dataandrestoretheoriginaldata.However,puttingall data in a single server is susceptible to the single-point-of-failure problem and vendor lock-ins. As suggestedinaplausiblesolutionistostripedataacross multipleservers.Thus,torepairafailedserver,wecan (i) read data from other surviving servers, (ii) reconstruct the corrupted data of the failed server,and (iii) write the reconstructed data to a new server.POR and PDP are proposed for the single-server case (e.g., Reed-Solomoncodes).

*Regenerating codes* have recently been proposed to minimize *repair traffic* (i.e., the amount ofdatabeingreadfromsurvivingservers).Inessence, theyachievethisby*not*readingandreconstructingthe wholefileduringrepairasintraditionalerasurecodes, but instead reading a set of *chunks* smaller than the original file from other surviving servers and reconstructingonlythelost(orcorrupted)datachunks. An open question is, *can we enable integrity checks atop regenerating codes, while preserving the repair traffic saving over traditional erasure codes?* A related approach is HAIL, which applies integrity protection for erasure codes. It constructs protection data on a per-file basis and distributes the protection data across different servers. To repair any lost protection data in the presence of a server failure, one needs to access the whole file, and this violates the design of regenerating codes. Thus, we need a different design of integrity checking tailored for regeneratingcodes.

The design and implementation of apractical data integrity protection (DIP) scheme for regenerating-coding-based cloud storage. We augment the implementation of the *functional minimum storage regenerating (FMSR)* code and construct *FMSR-DIP*, a code that allows clients to remotely verify the integrity of random subsets of long-term archival data under a multi-server setting.

ISSN 2394-3777 (Print)
ISSN 2394-3785 (Online)
Available online at www.ijartet.com

International Journal of Advanced Research Trends in Engineering and Technology (IJARTET)
Vol. 4, Special Issue 5, March 2017

FMSR-DIP aims to achieve several design features. First, it preserves fault tolerance and repair traffic saving as in FMSR. Second, it assumes only the thin- cloud interface, i.e., the servers only need to support the standard read/write functionalities. Third, it exportsseveraltunableparametersthatallowclientsto trade performance for security. There are used VII sections.

## II. EXISTING SYSTEM

Design and implement a practical Data Integrity Protection (DIP) scheme for regenerating coding-based cloud storage. FMSR-DIP codes preserve fault tolerance and repair traffic saving as in FMSR codes. . This adds to the portability of FMSRDIP codes and allows simple deployment in general types of storage services. By combining integrity checking and efficient recovery, FMSR-DIP codes provide a low-cost solution for maintaining data availability in cloud storage. In summary, we make the following contributions.

- These schemes can only provide the detection of corrupteddata.
- It does not recover the originaldata.
- Clientscannotgetthedatawhenthedata loss occurred in theserver.
- Time consumption for find the original data is veryhigh.

## III. PROPOSEDSYSTEM

The proposed technique is based on DRS theme uses solely the place and acquire info to work with every server. Our thin-cloud setting permits our DRS theme tobeconvenienttogeneralstylesofstorageprocedure or services, since no execution changes area unit needed on the storage backend. It differs from different “thick-”cloud-storage services wherever servers have machine capabilities and area unit capable of aggregating the proofs of multiple checks. Additionally, to cut back the native storage load, we are able to encipher all file keys with a passkey, and source the storage of the encrypted keys to the cloud. Transfer the code chunks from another server. A last various is to transfer the code chunks from all n servers.Wetendtocheckallrowsofthechunksas wellastheirAECCparities.Therowswithasetofthe bytes obvious correct is improved with FMSR codes; the rows with all bytes obvious corrupted area unit treated as erasures and can be corrected withAECC.

- It provides a low-cost solution for maintaining data present in the cloudstorage.
- Time for checking the data integrity in cloud is verylow.
- The security is analyzed by the mathematical model.
- Clients can get the data from the server in a fast manner.

## IV. RELATEDWORK

We consider the problem of checking the integrity of *static* data, which is typical in long-term archival storage systems. This problem is first consideredunderasingleserverscenariobyJulesetal and Ateniese et al, giving rise to the similar notions *proof of retrievability*(POR) and *proof of data possession*(PDP),respectively.Errorcorrectingcodes arealsoincludedinthestoredfiletoallowrecoveryof a small amount of errors within a file. The client to keep a small amount of metadata. The client can then challengetheserveragainstasetofrandomfileblocks to see if the server returns the proofs that match the metadata on the client side. The client can then challengetheserveragainstasetofrandomfileblocks to see if the server returns the proofs that match the metadata on the client side. A major limitation of the above schemes is that they are designed for a *single* server setting. If the server is fully controlled by an adversary, then the above schemes can only provide detection of corrupted data, but cannot recover the originaldata.WepointoutthatalthoughWebelievea better solution is possible by exploiting the cross-server redundancies in a multiple-server setting. Second, the storage scheme of assumes that storage servers have the encoding capabilities of generating a random linear combination of the data, while we consider a thin-cloud setting where servers only need to support standard read/write functionalities. Multi-server (or multi-cloud) storage has been proposed and implemented to protect against data loss and mitigate vendorlock-ins.

ISSN 2394-3777 (Print)
ISSN 2394-3785 (Online)
Available online at www.ijartet.com

International Journal of Advanced Research Trends in Engineering and Technology (IJARTET)
Vol. 4, Special Issue 5, March 2017

## V. PRELIMINARIES

It Provide background details. We state the threat model and the cryptographic primitives being used in our DIP scheme.

### A.FMSR Implementation

We first review the FMSR implementation. FMSR belongs to *Maximum Distance Separable (MDS)* codes. An MDS code is defined by the parameters (n, k), where $k < n$. It encodes a file F of size |F| into n pieces of size |F|/k each. An (n,k)-MDS code states that the original file can be reconstructed from any k out of n pieces (i.e., the total size of data required is |F|). An extra feature of FMSR is that a specific piece can be reconstructed from data of size less than |F|. FMSR is built on *regenerating codes*, whichminimizetherepairbandwidthwhilepreserving the MDS property based on the concept of network coding. We consider a distributed storage setting in which a file is striped over n servers using an (n, k)-MDScode.Eachservercanbeastoragesiteorevena cloud storage provider, and is independent of other servers. Suppose that one server fails. To guarantee that the MDS fault tolerance is preserved after multiple rounds of repair, NC Cloud performs two- phase checking on the new code chunks generated in the repair operation. In the case of (4,2)-FMSR the repair traffic is reduced by 25% to 0.75|F|. To access part of a file, the client needs to download anddecode theentirefile,andthisisnotsuitedtoapplicationsthat need random reads of different parts of a file. Nevertheless, FMSR is suited to long-term archival applications,wherethereadfrequencyislowandeach read operation typically restores the entire file. . We define the *repair traffic* as the amount of data being *read* from other surviving servers so as to reconstruct the lostdata.

### B. Threat Model

We adopt the adversarial model in as our threat model. We assume that an adversary is *mobile Byzantine*, meaning that the adversary compromises a subsetofserversindifferenttime*epochs*(i.e.,mobile) and exhibits arbitrary behaviors on the data stored in the compromised servers (i.e., Byzantine). To ensure meaningful file availability, we assume that the adversary can compromise and corrupt data in atmost n – k out of the n servers in any epoch, subject to the (n, k)-MDS fault tolerance requirement. At the end of each epoch, the client can ask for randomly chosen parts of remotely stored data and run a probabilistic checking protocol to verify the data integrity. Intuitively, it means that it is computationally infeasible for an adversary to break the security of a primitive without knowing its corresponding secret key. We also need a systematic *adversarial error-correcting code (AECC)* to protect against the corruptionofachunk.Inconventionalerror-correcting codes (ECC), when a large file is encoded, it is first broken down into smaller stripes to which ECC is applied independently. AECC uses a family of PRPs asabuildingblocktorandomizethestripestructureso thatitiscomputationallyinfeasibleforanadversaryto target and corrupt any particular stripe. Note that both FMSR and AECC provide fault tolerance. The difference is that FMSR applies to a file that isstriped across servers, while AECC applies to a single chunk stored within aserver.

## VI. DESIGN

We now present our design of DIP atop the FMSR code, and we call the new code *FMSR-DIP*. Our DIP scheme operates on the FMSR code chunks generated by NC Cloud [22], which is deployed as a client-side proxy that stripes data among multiple servers.

### A. Design Goals

We first state the design goals of FMSR-DIP.

***Preservation of regenerating code* properties:** We preserve the fault tolerance requirement and repair traffic saving of FMSR (with up to a small constant overhead) as compared to the conventional repair method in erasure codes

**Thin-cloud storage:** Each server (or cloud storage provider) only provides the basic interface for clients to read and write their stored files. No computation capabilities on the servers are requiredto support our DIP scheme. Cloud storage servers with encoding capabilities can be achieved by combining these two services, with the additional expense of renting the computation service. However, this approachreducesportabilityandintroducesacomplex costmodel.

***Flexibility:***Thereshouldnotbeanylimitson the number of possible challenges that the client can make, since files can be kept for long-term archival. Also, the challenge size should be *adjustable* with differentparameterchoices,andthisisusefulwhenwe want to lower the detection rate when the stored data growslessimportantovertime.Suchflexibilityshould come without any additionalpenalties.

ISSN 2394-3777 (Print)
ISSN 2394-3785 (Online)
Available online at www.ijartet.com

International Journal of Advanced Research Trends in Engineering and Technology (IJARTET)
Vol. 4, Special Issue 5, March 2017

***Cost minimization***: The cloud storage usage fee is mainly charged based on the storage space, transfer bandwidth, and number of requests. To minimizethestoragespaceandtransferbandwidth,we useAECC(whichisalsoanMDScode).Inparticular, the storage overhead of FMSR-DIP should come only from the AECC applied to each code chunk. Also, to reduce the number of requests while beingcompatible with the thin-cloud setting, we seek to reduce the numberofchallenge/responsepairsbetweentheclient and theservers.

*A .Notation*

We now define notation for FMSR-DIP, based on FMSR described in Section III-A. For an (n, k)-FMSR code, we define {_ij}1_i_n(n−k),1_j_k(n−k) as the set of encoding coefficients that encode k(n−k) native chunks {Fj}1_j_k(n−k) into n(n − k) code chunks {Pi}1_i_n(n−k). Thus, each code chunk Pi is formed by Pi = Pk(n−k) j=1 _ijFj . We define a *row* as a collection of all bytes that are at the same offset of all FMSR-DIP-encoded chunks. That is, the rth row corresponds to the bytes {P0 ir}1_i_n(n−k).

*C. Overview ofFMSR-DIP*

Our goal is to augment the basic file operations Upload, Download, and Repair of NC Cloud with the DIP feature. During Upload, FMSR-DIP expands the code chunk size by a factor of n0/k0 from the AECC. UnlikeHAIL,whichappliesDIPtothewholefile,we applyDIPtoeachFMSRcodechunkgeneratedbyNC Cloud. Thus, when NC Cloud reconstructs new code chunks during the repair of a failed server, we can directly apply DIP to the new code chunks without accessing the whole file. This preserves the property of repair traffic saving of FMSR. We describe the detailsoftheoperationsbelowtoexplainhowourDIP schemeworks.

*D. BasicOperations*

In the following discussion, we assume that FMSR-DIP operates in units of *bytes*. In Section V-C, we discuss how we relax this assumption to trade security for performance.

Upload operation. We first describe how we upload a file F to servers using FMSR-DIP.

*Step 1:* Generate the per-file secrets.
*Step* 2: Encode the file using FMSR.
*Step3*:EncodeeachcodechunkwithFMSR-DIP.
*Step* 4: Update the metadata file and upload.

***Check operation:*** In the Check operation, we verify randomly chosen rows of bytes based on the FMSR code chunks generated by NC Cloud.

*Step 1:* Check the metadata file.
*Step* 2: Sampling and row verification.
*Step 3*: Error localization.
*Step 4*: Trigger repair

***Download operation:*** We now describe how we download a file *F* from servers.

*Step 1:* Check the metadata file.

*Step 2*: Download and decode the FMSR-DIP-encoded chunks for file F.

***Repair operation:*** If some server fails then we trigger the repair operation via NCCloud as follows.

*Step 1:* Check the metadata file.

*Step 2*: Download and decode the needed chunks.

*Step 3*: Encode, update metadata, and upload.

## VII. IMPLEMENTATION

*FMSR-DIP* implementation atop NC Cloud and how we instantiate cryptographic primitives. Also,we addresshowwefine-tunevariousdesignparametersto trade security forperformance.

*A. Integration of DIP into NCCloud*

We implement a standalone DIP module and a storage interface module, and integrate them with NC Cloud .In the Upload operation, NC Cloud generates code chunks for a file based on FMSR. The code chunks will be temporarily stored in the local file system instead of being uploaded to the servers. The DIP module then reads the FMSR code chunks from the local file system, encodes them with DIP, and passes the resulting FMSR-DIP code chunks to the storage interface module, which will upload the FMSR-DIP chunks to multiple servers

ISSN 2394-3777 (Print)
ISSN 2394-3785 (Online)
Available online at www.ijartet.com

International Journal of Advanced Research Trends in Engineering and Technology (IJARTET)
Vol. 4, Special Issue 5, March 2017

In the Download operation, the DIP module checks the integrity of the chunks retrieved from the servers before relaying the chunks to NC Cloud for decoding.

*B. Instantiating CryptographicPrimitives:*

We implement all cryptographic operations using Open SSL 1.0.0g. All cryptographic primitives use128-bitsecretkeys.Werequirethatallsecretkeys be securely stored on the client side without being revealed to any server. The primitives are instantiated as describedbelow.

***Symmetric encryption:*** We use AES-128 in cipher-block chaining (CBC) mode.

***Pseudorandomfunction(PRF):***WeuseAES-128for PRF. The PRF input is first transformed to a plaintext block,whichisthenencryptedwithAES-128.Section V-C discusses how the size of the PRF output can be fine-tuned.

***Pseudorandom permutation (PRP):*** Our PRP implementation is based on AES-128, but applied ina different way as in PRF. Note that the domain size of the PRP is the number of elements to be permuted.To implement a PRP with a small and flexible domain size.

***Adversarial error-correcting codes (AECCs):*** We applythesystematicAECCadaptedfromasdescribed in Section III-C, with two main differences. First, for efficiency, Second, for mostnotably.

*C. Trade-offParameters.*

InSectionIV,FMSR-DIPoperatesinunitsof bytes. However, byte-level operations may make the implementation inefficient in practice, especially for large files. Here, we describe how FMSR-DIP can operate in units of blocks to trade security for performance.

***PRP block size*****:** Instead of permuting bytes, we can permuteblocksofatunablesize(calledthePRPblock size).AlargerPRPblocksizeincreasesefficiency,but at the same time decreases securityguarantees.

***Check block size*****:** Reading data from cloud storage is priced based on the number of GET requests. In the Check operation, downloading one byte per request will incur a huge monetary overhead. To reduce the number of GET requests, we can check a block of bytes of a tunable size (called the check block size).

***AECC parameters*****:** The AECC parameters ($n^0$,$k^0$) controltheerrortolerancewithinacodechunkandthe domain size of the PRP being used in AECC. Given the same $k^0$, a larger $n^0$ implies better protection, but introduces a higher computationaloverhead.

***Checking percentage*****:** The checking percentage $\lambda$ defines the percentage of data of a file to be checked intheCheckoperation.Alarger$\lambda$impliesmorerobust checking, at the expense of both higher monetary and performance overheads with more data to download andcheck.

## VIII. SECURITYANALYSIS

We elaborate the design choices of FMSRDIP and investigate its securityguarantees.

*A. Uses of SecurityPrimitives*

We briefly summarize the effects of various security primitives used in FMSR-DIP.

***Pseudorandom function (PRF):*** The effect of applying PRF on the data is similar to encrypting the data. It randomizes the data so that it is infeasible for the adversary to manipulate the original data and hencecorruptthedatainsuchawaythatthecorrupted bytes form consistent systems of linear equations during the Checkoperation

***Symmetric encryption*****:** We encrypt the metadata to hide the FMSR encoding coefficients. This protects against the scenario where the PRF values can be recovered with known encoding coefficients and original file content.

***Adversarial error-correcting codes (AECC)*****:** We use AECC to randomize the stripe structure, so that it is infeasiblefortheadversarytodeterministicallyrender chunks unrecoverable (see SectionIII-C).

***Message authentication codes (MAC)*****:** We include the MACs of individual chunks as metadata, and replicate them to all servers to allow integrity verification of any chunks.

ISSN 2394-3777 (Print)
ISSN 2394-3785 (Online)
Available online at www.ijartet.com

International Journal of Advanced Research Trends in Engineering and Technology (IJARTET)
Vol. 4, Special Issue 5, March 2017

*B. SecurityGuarantees*

We provide a sketch of analysis of the robustness of FMSRDIP against adversarial attacks. Recall from Section V-B that an FMSR code chunk is encoded by ($n^0$,$k^0$)-AECC. The code chunk is divided into $k^0$ fragments and b/$k^0$ stripes. Each fragment is permutedbyaPRPofsizeb/$k^0$,andtheneachstripeis encoded by an ($n^0$,$k^0$)-ECC to give a total of $n^0$ bytes each, so the code chunk is encoded by ($n^0$,$k^0$)-AECC into $n^0$ fragments Each stripe can correct up to $n^0$ –$k^0$ erasures or b($n^0$ – $k^0$)/2cerrors.

## IX. EVALUATIONS

We evaluate the practicalityof FMSR-DIP in a real storage setting by measuring the overhead of DIP in the Upload, Check, Download, and Repair operations. We empirically evaluate the running time overhead atop a local cloud storagetested.

A. Running TimeAnalysis

We first conduct tested experiments on a local cloud platform that is built on Open Stack Swift 1.4.2. We deploy our FMSR-DIP implementation in single-threaded mode on a machine equipped with Intel Xeon E5620, 16GB RAM, and 64-bit Ubuntu 11.04. The machine is connected via a Gigabit switch to an Open Stack Swift platform that is attached with 15nodes.WecreatemultiplecontainersonSwift,such that each container mimics a storageserver.

***Upload:*** We investigate the effects of four sets of parameters on the running time of the Upload operation, including (i) the input file size, (ii) the (n, k) parameters of FMSR, (iii) the ($n^0$,$k^0$) parameters of AECC, and (iv) the block sizes of PRP and PRF

***Check*:**Weevaluatetheeffectsofthecheckblocksize and the checking percentage on the Check operation. By default, we use the check block size of256KB and the checking percentage of 1%. We then vary one set of parameters each time in ourevaluations.

***Download and Repair:***

We now measure the total running times of the Download and Repair operations. Here, we only consider the effects of different file sizes, while other parameters use the same default values as in Upload. In the Repair operation, we consider the repair of a single failed server, which we simulate by setting the path of one of the Swift containers to a non-existent location.

*B. Monetary CostAnalysis*

We now describe the monetary overhead of FMSR-DIP in each of the operations compared to the original FMSR implementation in NC Cloud .

***Upload*:** The major source of the monetary overhead of our DIP scheme compared to NC Cloud is ($n^0$,$k^0$)-AECC, which expands the stored data and increases the storage cost by roughly $n^0$/$k^0$ (note that the inbound transfer cost is free for all commercial cloud providers that we consider). The cost due to the expanded file metadata is a negligible constant if the file size is large enough.

***Check:*** Since NC Cloud does not support the Check operation, we briefly discuss the sources of the Check cost. The Check cost is composed of the download bandwidthcostandtheGETrequestcost.Tominimize the download bandwidth cost, we can reduce the checkingpercentage

**Repair:** The major monetary overhead again comes from ($n^0$,$k^0$)-AECC in encoding the new FMSR code blocks. As discussed above, if there is no corrupted data in surviving servers, we preserve the network transfer cost of NC Cloud when downloading data from the surviving servers (aside from the small constant metadata traffic. Therefore, we still preserve the cost saving property of the repair operation in NC Cloud when compared to the conventional repair method (by up to 50% for RAID-6. [6]discussed about a method, Sensor network consists of low cost battery powered nodes which is limited in power. Hence power efficient methods are needed for data gathering and aggregation in order to achieve prolonged network life. However, there are several energy efficient routing protocols in the literature; quiet of them are centralized approaches, that is low energy conservation.

## X. CONCLUSION

Seeing the popularity of outsourcingarchival storage to the cloud, it is desirable to enable clients to verify the integrity of their data in the cloud. We design and implement a practical data integrity protection (DIP) scheme for functional minimum storage regenerating (FMSR) codes under a multiserver setting. Our DIP scheme preservesthe

ISSN 2394-3777 (Print)
ISSN 2394-3785 (Online)
Available online at www.ijartet.com

International Journal of Advanced Research Trends in Engineering and Technology (IJARTET)
Vol. 4, Special Issue 5, March 2017

fault tolerance and repair traffic saving properties of FMSR. FMSR-DRS. The PRFs off the FMSR-DRS
code chunks to selection the FMSR code chunks, that area unit then passed to NC Cloud for cryptography if they'renotcorrupted.DownloaditsAECCparitiesand have an effect on error modification. To understand the practicality of the integration of FMSR and DIP, we analyze its security strength, evaluate its running time overhead via tested experiments, and conduct monetary costanalysis.

## XI. REFERENCES

[1] H. Abu-Libdeh, L.Princehouse, and H. Weatherspoon. RACS: A Case for Cloud Storage Diversity. In *Proc. of ACM SoCC*, 2010.

[2] R. Ahlswede, N. Cai, S.-Y. R. Li, and R. W. Yeung. Network Information Flow. *IEEE Trans. on Information Theory*, 46(4):1204– 1216, Jul 2000.

[3] Amazon Elastic Compute Cloud. http://aws.amazon.com/ec2/.

[4] Amazon Simple Storage Service. http://aws.amazon.com/s3/.

[5] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, I. Stoica, and M. Zaharia. A view of cloud computing. *Communications of the ACM*, 53(4):50–58, 2010.

[6] Christo Ananth, S.Mathu Muhila, N.Priyadharshini, G.Sudha, P.Venkateswari, H.Vishali, "A New Energy Efficient Routing Scheme for Data Gathering ",International Journal Of Advanced Research Trends In Engineering And Technology (IJARTET), Vol. 2, Issue 10, October 2015), pp: 1-4

[7] [7] G. Ateniese, R.DiPietro, L.V.Mancini, and G.Tsudik. Scalable and Efficient Provable Data Possession. In *Proc of SecureComm*, 2008.