Designing a Secure Storage Repository for Sharing Scientific Datasets using Public Clouds

Alok Kumbhare
University of Southern California
Los Angeles
kumbhare@usc.edu

Yogesh Simmhan
University of Southern California
Los Angeles
simmhan@usc.edu

Viktor Prasanna
University of Southern California
Los Angeles
prasanna@usc.edu

ABSTRACT

As Cloud platforms gain increasing traction among scientific and business communities for outsourcing storage, computing and content delivery, there is also growing concern about the associated loss of control over private data hosted in the Cloud. In this paper, we present an architecture for a secure data repository service designed on top of a public Cloud infrastructure to support multi-disciplinary scientific communities dealing with personal and human subject data, motivated by the smart power grid domain. Our repository model allows users to securely store and share their data in the Cloud without revealing the plain text to unauthorized users, the Cloud storage provider or the repository itself. The system masks file names, user permissions and access patterns while providing auditing capabilities with provable data updates.

Categories and Subject Descriptors

D.4.6 [**Operating Systems**]: Security and Protection—*Access Control*; K.6.5 [**Management of Computing and Information Systems**]: Security and Protection

General Terms

Public Clouds, Secure Cloud Storage, Access Control, Data Repository, eEngineering

1. INTRODUCTION

Data sharing is a key tenet of scientific computing, more so as we push toward data intensive sciences and engineering. This sharing applies not just to scientific results but also to the raw and intermediate data that go into the scientific process. Several funding agencies like the US National Science Foundation even mandate this. Cloud platforms that provide the benefit of on-demand storage collocated with the computational resources required to process and analyze data are attractive to host scientific repositories. However, there are several existing and evolving issues that make this choice of hosting scientific data in *Public Clouds* difficult.

While many scientific and engineering disciplines are open to widely sharing data, there are others who are **restricted in the data they can share** and whom they can share it with. The latter occurs particularly when human subject data is concerned and regulatory requirements are put in place by Institutional Review Boards (IRB). This is not just limited to life sciences and healthcare, where HIPAA regulations are codified, but also to behavioral sciences and sociology. The data collected from *crowd sourced human sensors* is growing rapidly. People are willing to share their activity data as part of a social network or automatically report information using sensors they carry[21]. Human subject data is starting to be used in everything from traffic management to electricity demand prediction. The aggregation and sharing of this data for scientific and operational use needs careful consideration and enforcement since even de-identified data can be cross-referenced and correlated with external information to reveal private information.

The growth of pervasive (human or electronic) sensors combined with the multi-disciplinary direction in which various sciences are headed also means that the **number of data producers**, owners and consumers are increasing. The data owner, say a subject in a life sciences study, may not want to just hand over control of their data to the institution but restrict access to a specific research team with the option of expanding this permission to other researchers or even a commercial entity later, or revoking this permission altogether. Similarly, in the context of the smart power grids, an electricity consumer may wish to impose access control over fine grained power usage data collected from their smart appliances by the utility, which may use this data for demand forecasting. Being able to provide data owners verifiable control over their data while at the same time having the ability to host, manage and share a single data repository for data consumers from multiple disciplines and groups is important.

Lastly, as repositories are hosted on Cloud data centers external to a research group or institution, it **expands the potential for data leakage**[28]. Storing plain text data in the Cloud makes it available to the Cloud service provider or rogue employees. Currently, Service Level Agreements (SLAs) from major Cloud providers have limited liability provisions for loss of security and privacy and is often a best effort. In fact, the Cloud Security Alliance's 2010 guidance recognizes "Malicious Insiders" and "Data Loss or Leakage" as the third and fifth top security threats to Clouds[8]. The multi-tenancy inherent to Clouds also widens the attack surface through shared storage, network and VMs.

There are several dimensions to solving this problem and recent research into Cloud security has addressed individual aspects. Simple approaches like encrypting data that resides in the Cloud may work when the data is owned by a single institution and accessed by a small set of users with decryp-

DataCloud-SC'11, November 14, 2011, Seattle, Washington, USA.
ACM 978-1-4503-1144-1/11/11.

tion keys, but some of these techniques[18, 17] require out of band communication. CloudProof[24] provides mechanisms for key sharing and data integrity but does not address issues of file management required in a practical repository design. Cloud data sharing services like DropBox while providing an intuitive client interface expect centralized trust in them. Several Cloud providers host scientific data on their platforms to attract researchers, but these are open, unsecured data[1]. Numerous communities and federal scientific repositories provide open or coarse grained security access for their members[2]. However, a more elaborate design is necessary to support a shared, secure Cloud data repository supporting multiple data owners and consumers, hosted on an untrusted Cloud storage provider, and with no central authority privy to plain text data.

The contribution or our paper can be summarized as follows:

1. We characterize specific security and privacy requirements for eScience and eEngineering data storage and sharing, and motivate it using the smart power grid domain.

2. We present an integrated design for a shared, secure Cloud data repository, *Cryptonite*, that incorporates contemporary encryption techniques to address these issues.

3. We analyze the security offered by our design and its potential short-comings, including future work to address them.

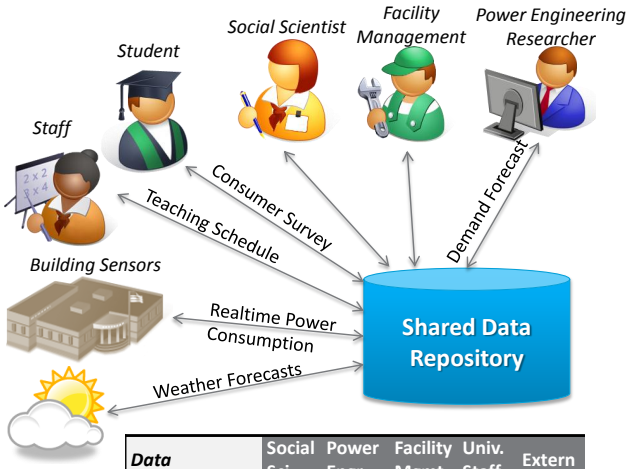
The rest of the paper is organized as follows. In Section 2, we characterize this problem further using the smart power grid domain that we are engaged in. In Section 3, we review existing security techniques and repositories that have influenced our design. In Section 4, we describe the various entities that participate in sharing and controlling the data, and in Section 5 we describe the operations that need to be supported by our design. Section 6 describes the Cryptonite architecture, the security techniques leveraged and a walk-through of certain operations. We discuss the security issues addressed by our design in Section 7, and present our conclusions in Section 8.

2. MOTIVATING APPLICATION

Energy sustainability and security is taking center-stage globally and transforming the way energy is produced and used. *Smart Power Grids* are beginning to collect and analyze realtime power usage information from consumers through smart meters and appliances to better manage electricity. Technologies like electric vehicles, intermittent renewables like wind and solar power, and smart home appliances are causing demand and supply to become more dynamic. Power systems researchers and utilities are mining consumer data, including power usage patterns, location sensors, schedules, and demographics, to build demand forecasting models, to influence electricity consumption, and to plan purchase from the energy marketplace[27]. Third party service providers

[1]Google Research Datasets, Amazon's Public Data Sets on AWS, and Microsoft Azure's Datamarket

[2]www.unidata.ucar.edu/projects/THREDDS, www.ncbi.nlm.nih.gov/genbank, www.fluxdata.org



Data	Social Sci.	Power Engr.	Facility Mgmt.	Univ. Staff	Extern
Cons. Survey	O	R*	-	-	-
Staff Schedule	R*	R*	-	O	-
Consumption	R	R	O	-	-
Weather	R	R	O	R	W
Campus Power Forecast	R	W	O	R	R

Figure 1: Data sharing in campus microrid. *R*ead, *W*rite, *O*wnership and Masking *(*)* are tabulated.

are using data analytics to assist both consumers and utilities: to schedule appliance duty cycles, provide load curtailment plans, and avoid blackouts and brownouts.

Smart grids are a prime example of *eEngineering* that is going the way of eScience in using massive amounts of data to understand and affect the behavior of cyber-physical systems. For example, the **USC campus microgrid** testbed mimics a power utility service area for the DOE-sponsored Los Angeles Smart Grid Project[29]. Researchers from social sciences, computer science and electrical engineering are collecting and studying realtime information from "power consumers" on campus facilities using building sensors, staff surveys, mobile applications, and weather services to build *demand forecasting* and *load curtailment models*[2]. A *data repository* that runs off a public Cloud will host the various sources of information for the project. Data intensive analytics algorithms collocated in the Cloud will analyze and mine them to build aggregate power demand models.

Figure 1 describes the different types of data in the campus microgrid, their producers and consumers, and their sharing and privacy requirements off the repository. The campus *Facility Management* is responsible for managing power distribution to buildings within the campus, and use smart meters and building sensors to measure power consumption in realtime and report it every few minutes. A single file of time series *consumption* data is maintained and updated for each of the 170 campus building within the data repository. *Social scientists*, who are modeling the behavior of consumers and their reaction to energy conservation programs, conduct on-campus surveys regulated by IRB. These *survey responses* from thousands of campus staff and students are hosted in the data repository. *Power engineering researchers* collect additional information from teaching staff on their class and office hour schedule to model future power demand in classrooms and office rooms. These

schedule files are regularly updated by the staff's calender applications and maintained in the repository. These engineers also aggregate publicly available *weather forecast information* from NOAA into the repository, and train the demand prediction models for the campus using compute- and data- intensive machine learning techniques[2]. Daily and 15-min interval *demand forecast plots and time series files* are generated continuously using these models by the facility management and hosted in the repository.

The table in Figure 1 lists the access restrictions that exist for each class of data for different users. The restrictions need to be enforced on *individual* users and data files, though we just show the *class* of users and files for brevity. Survey data is highly sensitive and only social scientists can own and manage them. Power engineers may read de-identified and *masked* versions of these files for their coarse grained models. Each teaching staff owns their classroom schedule information and provides masked read access to the social scientists and power engineers. Realtime building power consumption data from sensors is managed by facilities and they provide read access to these for experiments by the scientists and engineers, but not to others. Weather forecast information is maintained by the facilities and they allow external service providers to push data to them. This is readable by everyone. The predictions from the power forecast models are maintained by the facilities and shared with the *university members*, or their mobile applications, as part of the campus sustainability effort. The power engineers may update these predictions occasionally to fix outliers.

The following specific security and privacy requirements can be synthesized from this scenario for a shared repository::

<R1>: Data storage security: Data files stored in the repository should not be revealed in *plain text* to unauthorized users. These users include the Cloud provider, applications, and even the data repository provider.

<R2>: Metadata storage security: Metadata properties of the data files should not be revealed in *plain text* to unauthorized users. Even the name of the data file itself should not be revealed since file creators may unintentionally encode sensitive information in them.

<R3>: owner control of data sharing: The data owner should be able to specify access permissions to grant access to a subset of other users. These permissions should discriminate between access rights for read, write and delete. It should also be possible to revoke access at any time from specific users. These controls should not require out of bands communication between the users for key exchange, etc.

<R4>: Data integrity and Audit: The data owner should be able to verify the integrity of the the data stored in the repository. Any unauthorized changes to the file contents or its metadata should be provable to third parties or regulators.

<R5>: Masking access control list: The identities of the authorized readers and writers should not be revealed to any users, nor should the fact that several files have the same (opaque) access restrictions, to prevent unintended correlations to be drawn between the files.

<R6>: Masking access patterns: The access pattern for a particular file (*Which users are accessing the file? How many requests for a particular file are made?*) should be masked to avoid drawing attention to a given file.

<R7>: Assured file deletion: The user should be able to ensure that a file delete operation permanently deletes the file and its metadata[33].

These issues of data sharing and privacy that we discuss at the campus microgrid level are more serious[25, 22] when scaled up to the operational scenario of a *city power utility*. A smart grid ecosystem may include millions of customers, the utility's IT infrastructure, Home Area Network applications controlling consumer smart appliances[34], external vendors providing data aggregation [3], analysis and advertising, electricity regulators [4], and traders in the energy marketplace [5] who all have rights, interests or concerns in generating, sharing and consuming data.

3. RELATED WORK

There exist a number of Cloud services providing different levels of abstractions for data storage. For example, *Amazon AWS* and *Microsft Azure Storage service* can host binary files, tables or queue data stuctures. The SLAs for these systems guarantee availability, but exclude guarantees of data security. By default, these provide coarse gained authentication and access control to the entire storage account but some also allow fine-grained Access Control Lists (ACLs) and individual file policies. Despite physical security at data centers, it is possible for service providers or malevolent employees to access stored data. In addition, the access control is enforced by the Cloud service without any auditing capability, or ability to mask the users accessing data or the access patterns. Hosting an encrypted file system with ACLs in a Cloud VM, backed by the persistent Cloud storage, is also possible. However, such file systems are usually meant for protecting data from unauthorized users with physical access to the disk rather than as a means to enforce privacy across multiple users. The administrator of the OS still has access to the contents of the files.

Consumer oriented services like DropBox provide an intuitive interface but have had data breaches[6]. Systems such as Nasuni Cloud storage[7] provide a higher level of abstraction and better security guarantees through client side encryption and access control. However, they do not provide fine grained data sharing capabilities, and require key management to be handled by the client.

In the research community, NAS/DS and SNAD[11, 20] were among the first systems tackling issues of untrusted remote data storage and data sharing. They use end-to-end encryption to protect data from remote and local intruders. The storage server does not have enough information to generate the encryption key for any of the data owners – client-side en/decryption ensures this protection. They support data sharing, but use a set of permission bits to give/restrict access to each user, and trust the server to restrict access based on these bits. The SiRiUS[14] system separates file encryption and file signature keys, allowing read and write operations to be cryptographically distinguished even with an untrusted Cloud storage. It does not

[3]www.google.com/powermeter

[4]www.ferc.gov/market-oversight/mkt-electric/california.asp

[5]www.caiso.com

[6]A Breach in drop box security caused the file names to be revealed. techcrunch.com/2011/06/24/dropbox-breach-fewer-than-100-accounts-affected-but-one-person-actively-exploited-it

[7]www.nasuni.com

rely on the storage to ensure authentication. However, it has significant overhead for storage size and the number of required keys. PLUTUS[17] introduced the concept of *lockboxes* which is used to group files with similar access parameters and *key rotation* to efficiently manage data sharing and revocation. However, it requires out of band communication for key distribution. Stanton's survey[32] provides a comparative study of different approaches for secured data storage in traditional distributed systems using a set of evaluation criteria based on confidentiality, integrity, availability, and performance. While Cachin, et al.[7] discusses several concerns that can arise for data storage in Clouds and surveys cryptographic tools that can be employed to provide data integrity and consistency.

Cloud-Proof[24] and Cryptographic Cloud Storage (CCS)[18] are systems built specifically for Cloud environments. They leverage "lockboxes" similar to PLUTUS, and CloudProof further uses Broadcast Encryption[10] to encrypt the distribution of the lockbox keys which avoids the need for out of band communication. However, it has been designed at a lower system abstraction and do not address our requirements concerning file management such as ability to perform secure search over the stored data. CCS defines both consumer and enterprise data sharing scenarios, and proposes using an attribute based encryption technique for data encryption and key management. It uses searchable encryption technique to allow the end users to transparently search over the encrypted data. We leverage some of these security techniques but cover a broader set of requirements that we have motivated.

There are a number of scientific repositories that are both general purpose and bespoke for individual domains. The Globus Metadata Catalog Service (MCS)[30] is part of the popular Globus Grid computing stack, and used for discovery and access of computational and data resources. It uses the Grid Security Infrastructure to manage authentication and authorization. Catalogs such as MyLEAD[23] extend MCS for geo-spatial domains like meteorology and provide personalized catalogs. However, while such repositories often secure the network communications, securing the stored data is left to file system or database permissions. There has also been little attempt to mask data access patterns or access rights from the repository service provider. Such issues have gained recent importance with the rise in human subject data and public Cloud hosting capabilities. Our data repository design, while not full fledged in terms of client features or query capabilities, does cover a wide swathe of data privacy concerns and can serve as a replacement for the storage backend.

4. DATA ENTITIES AND ROLES

In this section, we characterize the scope of the design requirements in terms of the data entities that are supported and the different actors that will be participating in accessing and managing this data.

We model the contents of the data repository interface by keeping it similar to Cloud file storage services[8][9]. This approach is simple for users, while it also helps us cleanly map it to existing Cloud storage that will host the repository. The repository holds files, that are a sequence of bytes, along

[8]aws.amazon.com/s3

[9]www.microsoft.com/windowsazure/features/storage

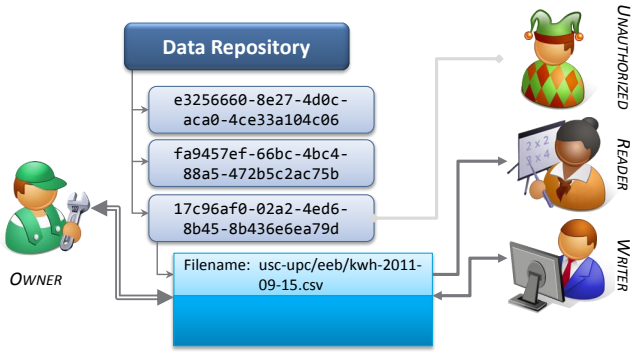


Figure 2: Data entities and roles in the repository.

with metadata in the form of <name,value> pairs of strings. In this article, we do not consider hierarchical collections of files and all files reside in a single namespace. However, file names may encapsulate hierarchies that are interpreted by the users. E.g. a filename of *usc-upc/eeb/kwh-2011-09-15.csv* may be opaque to the repository but considered by the client to be the file *kwh-2011-09-15.csv* present within the *eeb* directory, and it within the *usc-upc* directory. File names are one of the metadata fields.

Figure 2 illustrates the different data entities and user roles which are as follows:

- **User:** This refers to the end users within the community who require a secure storage repository to share data with each other. Users who create and populate data files are *"owners"* of that file. owners may provide access to other users to just read or both read and write to these file. The authorized users are *"Readers"* or *"Writers"* respectively. owners retain the exclusive right to delete the files. Users may be human beings, institutions, or their software agents. Users need to be uniquely and well identified. There may also be *unauthorized* users, who may be either benign or malicious, that attempt to access the data repository and files they do not have access to. Their restrictions must be enforced.
- **Cloud Storage Service Provider** is a passive entity which provides the required scalable and persistent storage space using file based semantics. These may be, for example, Azure Blob Storage or Amazon Simple Storage Service, that provide buckets or collections of binary files or blobs, and limited support for storing metadata properties along with the files. It provides limited access control mechanisms at the granularity of the entire storage account used to store all files by the repository and can be used to restrict access from users who have no access necessity.

We expect the Cloud provider to store and return byte content, and prevent users without an account from modifying the data. We also ensure that the end users can detect if it allows unauthorized updates. However, the extreme assumption is that the a malicious staff of the storage service provider or other Cloud users in the multi-tenant environment may be able to gain access to the storage account, and hence cannot be trusted with plain text data.

- **Secure Data Repository:** This is the shared data repository that we design and provide, and henceforth refer to as **Cryptonite**. It uses the Cloud Storage Service as the backend store and manages user accounts and enforces ac-

cess rights. The provider that runs Cryptonite itself may not be trusted with plain text data. The Cryptonite service itself is deployed in the Cloud and hence has limited trust like the Cloud provider. We trust Cryptonite service to impose appropriate restrictions by verifying the signature with each of the update requests. However, we use appropriate audit mechanisms, as described later, to detect unauthorized modifications. This enables the users to detect but not prevent the unauthorized modifications and may cause data loss.

5. SUPPORTED OPERATIONS

The data repository supports several operations that allow files to be created and accessed, as well as access permissions to be maintained. The repository is a web service that uses WebDAV-style operations. This provides intuitive semantics to the operations and maintains similarity with the REST service model used by existing Cloud service providers. In the proposed system design, we provide the following operations: **PUT**, **GET**, **DELETE**, **GRANT**, **REVOKE**, and **SEARCH**, that between them can be used to support the motivating example and help build higher level abstractions of data operations. Here, we describe each of these operations along with unique security and privacy requirements.

PUT: This operation is used to create a new file resource, or update the contents of an existing file. The user that creates the file initially is its *owners* and this operation creates a new file with "default" permissions. Authorized *writers* as well as the owner can update the contents of an existing file. Most of the motivating security requirements, R1–7 introduced in Section 2, are relevant to the PUT operation. Specifically, the need to mask the file name from the data repository means that we have to use an opaque *file handle* rather than the *file name* to refer to the file resource passed to the PUT operation. We use UUIDs as file handles.

GET: Given the file handle (UUID), this operation allows authorized *readers* to retrieve the metadata properties and the binary data for the file from the data repository. The owner and writers are implicitly readers too. The GET operation should not reveal the access patterns for a particular file, among other requirements listed earlier.

DELETE: This operation deletes the file contents, its associated metadata and the associated permissions for a given file handle from the data repository and the Cloud storage service. Only the *owner* of the file can delete the file.

GRANT: This operation grants specific access permissions to a set of users for a particular file handle, and adds to the existing access control list for the file. This can only be performed by the *owner* of the file. Among other requirements listed before, the identities of the authorized readers and writers should not be revealed to other users or the repository service.

REVOKE: This operation revokes specific access permissions from a particular set of users for the given file handle. The access control list for the file is updated with and this operation is only authorized for the owner of the file.

SEARCH: This operation is used to search the metadata associated with particular file handle (or the entire repository) by authorized readers of the files. This is commonly used to lookup the file handle for a given file name (or vice versa), or find the file handles matching a set of keywords. The search results should contain only those files over which the requesting user has read access permission.

Other operations on collections of files and modifying metadata are relevant but outside the scope of this paper. There are also operations such as "copy" and "paste" that can be implemented on the client-side, using graphical interfaces, using these primitives.

6. CRYPTONITE ARCHITECTURE

The Cryptonite Architecture integrates client-side libraries with the secure data repository service and existing Cloud storage service to support the operations defined in section 5. Figure 3 shows the components of Cryptonite.

The *Cryptonite Client Library (CCL)* is responsible for local cryptographic operations such as encrypting and preprocessing the plain text files before uploading it to the Cryptonite repository service, and decrypting it upon receipt. It may also maintain a local cache of the encrypted files and cryptographic keys to reduce latency, wherever possible without increasing the security risk. The client library runs on the user's local machine, be it on a desktop or within a Cloud VM, and forms the edge of secure operations provided by Cryptonite.

The *Cryptonite Data Repository Service* runs within a Cloud VM and has three major sub-components: the File Manager (FM), the Secure Index Manager (SIM) and the Audit Manager (AM). The *File Manager* has exclusive responsibility to interact with the Cloud storage service to store and retrieve files requested by the user. It also restricts unauthorized updates to the stored data or access permissions by verifying the digital signature on the requested operation's message.

The *Audit Manager* maintains a secure audit log for each file access performed by the File Manager, including a signature of the requested operation's message. This can be used as a transactional or provenance log to prove to a third party regulator how the current state of the repository came to be.

The *Secure Index Manager(SIM)* keeps a secure index per user for all the files that are owned by that user . The index is stored in a separate Secure Index Storage (SIS) space. The Index Manager accepts and executes user's SEARCH queries over the index and returns matching results. The index stores a map from the file's metadata, including the set of keywords, the filename and the associated attributes for a file, to the file's UUID. The index entries are encrypted using the Searchable Encryption Scheme, described later, such that only the authorized users can search over the index based on any of those parameters.

The *Cryptonite Secure Storage (CSS)* is a storage account with the public Cloud data service used to store the files and metadata. The CSS uses standard authentication mechanisms provided by Cloud data services to access this space and use it to store a flat collection of binary files. The Secure Index Storage is collocated in the same public Cloud, but can use a Cloud Table service rather than the file service to efficiently store and query the file index.

6.1 Cryptographic Techniques

Cryptonite uses several well known and emerging cryptographic and security techniques in its secure design. We introduce salient techniques here before discussing our security model.

Public Key Infrastructure(PKI)[19] provides each user with a User Identity and allows the user to associate the pub-

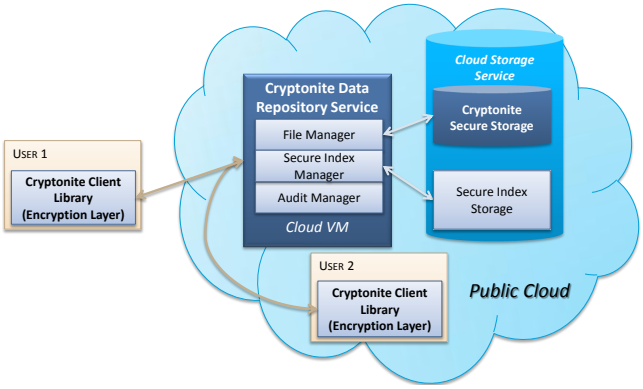


Figure 3: Architecture diagram of the Cryptonite secure data repository and client libraries.

lic/private key pair with that identity. We use PKI throughout the system to reliably associate the user's public key with his identity. For each user we denote an identity tuple $<U_i, K^{pub}_{U_i}, K^{pri}_{U_i}>$, where U_i is the identity of the user i, $K^{pub}_{U_i}$ is the public key of the user which can be retrieved from the trusted server in PKI, and $K^{pri}_{U_i}$ is the private key known only to the user U_i.

Digital Signatures[16] uses PKI for maintaining and verifying the integrity of data. The sender of a file generates its hash and signs it using their private key. The receiver who gets the file and the signed hash (i.e. signature) uses the sender's corresponding public key to decrypt the hash and verify it against the hash they generate for the received file. This also provides for non-repudiation since only the sender could have generated a valid signature using the private key matching the public key. We use digital signatures for integrity verification as well as auditing purposes. We denote a signature operation as $sign(D, K^{pri}_{U_i})$, where D is the data to be signed, and $K^{pri}_{U_i}$ is the private key of the user. The corresponding public key $K^{pub}_{U_i}$ can be used to verify this signature.

Broadcast Encryption allows a user to encrypt their data such that it can be decrypted only by a particular subset of users. The idea behind broadcast encryption is to generate a shared encryption K^{shared}_{encr} from the public keys of each of the receivers $(U_1, U_2, U_3, ...)$:
$K^{shared}_{encr} = f(K^{pub}_{U_1}, K^{pub}_{U_2}, ...)$
The data is then encrypted using this shared key and transmitted to the receivers. $F = encrypt(D, K^{shared}_{encr})$
Upon receiving this data, each of the authorized users can decrypt this using a key derived from their private keys. $D = decrypt(F, K^{pri}_{U_i})$
Broadcast Encryption was introduced in[10] and improved by[15, 12, 3]. We use broadcast encryption to distribute the file encryption and file signature keys using the *StrongBox*, described later.

Lazy Revocation[4] is a strategy in which a file, whose read access rights are removed for a user by its owner, is not re-encrypted (with high overhead) unless the file's contents change. Hence the revoked user will continue to be able to read and decrypt the file they already had access to until its contents are updated, upon which the file is encrypted and signed with a new set of keys that the revoked user does not know. *Key rotation*[26] is a related technique which allows a user to produce a sequence of keys from an initial key and a secret master key. Only the owner of the secret master key can produce the next key in the sequence, but any user knowing a key in the sequence can produce all *earlier* keys in the sequence. This gives forward secrecy. We combine lazy revocation and key rotation to minimize the overhead for re-encrypting a file when its access permissions change[17].

Searchable Encryption[31] allows a user to search within an encrypted file given an appropriate "TrapDoor" which can be opened only by the master key used to encrypt the file in first place. This allows a user to search within an encrypted file for the presence of a particular text without decrypting the entire file or revealing its contents to the searching entity. An extension of this is to create a "secure index" which maps a set of metadata properties to file handles that contain them, and allowing the user to search this encrypted index (including conjunctive queries)[13]. These early techniques, also called Symmetric Searchable Encryption (SSE), only support cases where the same user stores and searches over the secure file[1, 9]. Recently, techniques which support multi-user access scenario using Asymmetric Searchable Encryption (ASE) as well as variation of SSE have been introduced[5, 6, 9]. However, the features offered by existing searchable encryption techniques, as summarized in[18], are still insufficient to limit access to authorized readers, and highlight a gap in the state-of-the-art.

6.2 StrongBox Encrypted Data Structure

A foundational concept that we use for securing files is called a *StrongBox*. This is a data structure which is used for efficient key management and distribution for a collection of files that share the exact same access permissions (i.e. same set of readers and writers). The *StrongBox*serves a similar purpose as the "File LockBox"[17] or the "Family Key Block"[24]. All files within a *StrongBox*are encrypted using a single symmetric key, K^{sym}_{encr}, and signed/verified using a single pair of keys, $<K^{pub}_{sign}, K^{pri}_{sign}>$. All files in the *StrongBox*are owned by a single user, have one set of authorized readers and one set of authorized writers. In addition, to avoid known plain text attacks, we use a unique *Initialization Vector (IV)* to encrypt different files within the same *StrongBox*.

Each file referred to within the *StrongBox*itself contains several parts, as shown in Figure 4, and identified using a file handle or UUID. A file consists of three main parts. The encrypted data F, the security metadata M both of which are stored together in the same file; and the file metadata P, which includes the set of keywords, the file-name and various properties associated with the file. P is stored with the index entry for the file in the Secure Index and not with each of the files.

F is obtained by encrypting the plain text data, D, using the symmetric file encryption key, K^{sym}_{encr}, and an initialization vector IV. The same file encryption key is used by the writers can also be used by the readers to decrypt the file, hence the symmetry. Of the public-private signature key pair, K^{pri}_{sign} is used to sign the hash generated for the encrypted file contents, F, and generate the signature, S_F. Also, the K^{pub}_{sign} is used by Cryptonite for verifying the signature to ensure that it is coming from an authorized user.

The security metadata M contains the file handle $SUUID$ for the associated *StrongBox*, the file verification key K^{pub}_{sign}

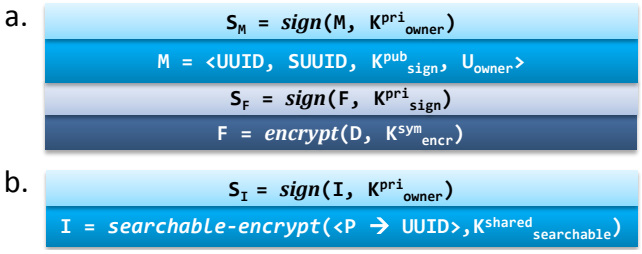


Figure 4: (a) Structure of an encrypted data file, including its security metadata(M). SUUID is the UUID of the corresponding *StrongBox*, K^pub_{sign} is the file verification key. S_M and S_F is the digital signature of security metadata and the file respectively. (b) An index entry corresponding to the file containing properties P associated with the file. S_I is the digital signature of the index entry I

and the ownerID. M, is signed by the owner's private key $K^{pri}_{U_i}$ and not by the file signature key K^{pri}_{sign}. This enables Cryptonite to restrict unauthorized modifications to the security metadata M and allows only the owner of the file to make any changes.

Figure 5 illustrates the structure of the *StrongBox*, which is a special file stored in same storage space as data files with their unique handle SUUID. It contains a list of file handles (UUIDs) for files managed by this *StrongBox*, along with the unique Initialization Vectors used for encryption. This list, L, is itself encrypted using the symmetric file encryption key , K^{sym}_{encr}, to protect the list of files sharing the same permissions from being revealed to an unauthorized user.

The security metadata of the *StrongBox*contains: (1) The symmetric file encryption key, K^{sym}_{encr}, encrypted using Broadcast Encryption such that it can be decrypted both by authorized readers and writers for the files in the list , (2) The private key used for signing and encrypted file, K^{pri}_{sign}, also encrypted using broadcast encryption such that it is only accessible to the authorized writers, (3) The Initialization Vector used to encrypt the *StrongBox*itself, and (4) The U_{owner} for this *StrongBox*. The *StrongBox*, including its encrypted list and the security metadata, is digitally signed with its owner's private key, K^{pri}_{owner}, same as the file's security metadata so that the modifications to the *StrongBox*are restricted to the owner.

6.3 Design of Operations

6.3.1 PUT Operation

The PUT operation is used to create a file or to update an existing one. Creating a file involves uploading an encrypted file from the file owner's machine, through the Cryptonite client libraries (CCL), to the Cryptonite repository service and assigning it an initial set of access permissions. An update entails rewriting the entire contents of the file without changing its access permissions, and can be performed by any authorized writer for that file. We describe the steps taken to perform this operation using the architecture components and cryptographic techniques we have introduced. Figure 6 shows this as a sequence diagram.

INPUT: A plain text file (D) to be uploaded, an initial set of access permissions $A =< U^R_{1..n}, U^W_{1..m} >$, file meta-

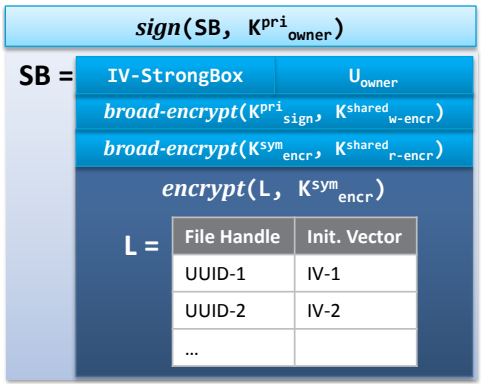


Figure 5: A *StrongBox*. L is the data stored in the *StrongBox*which is the list of file UUIDs and the corresponding IVs. $broad-encrypt(Y, X_u)$ refers to broadcast encryption of data Y with a shared key X such that the users of the set u can decrypt the data. $encrypt(D, X)$ gives the symmetric encryption of data D under key x, $sign(X, Y)$ is the digital signature of X under the private key y

data properties (P), and the identifier tuple of the owner, $<U_{owner}, K^{pub}_{U_{owner}}, K^{pri}_{owner}>$.

STEP 1: Retrieve or Create *StrongBox*

- CCL uses the access permissions A along with a unidirectional encoding function

$$SUUID = \textbf{encode}(A, K^{pri}_{owner})$$

to determine the *StrongBox*identifier and retrieve it from the repository's File Manager, FM. There are no access restrictions on the encrypted *StrongBox*file.

- If a *StrongBox*with that $SUUID$ does not exist, CCL locally creates a new *StrongBox*corresponding to Figure 5. This includes generating a new symmetric key K^{sym}_{encr} for file encryption, a new public-private key pair $<K^{pub}_{sign}, K^{pri}_{sign}>$ for file signatures, and a random initialization vector $IV - StrongBox$. It also encrypts K^{sym}_{encr} and K^{pri}_{sign} using broadcast encryption with the shared key K^{shared}_{r-encr} and K^{shared}_{w-encr} respectively, generated from the public keys of the authorized readers and writers listed in, and A and stores them in the security-metadata of the *StrongBox*.

STEP 2: Create and Send Encrypted File

- CCL retrieves the symmetric encryption key K^{sym}_{encr} and signature key K^{pri}_{sign} from the *StrongBox*.
- It then encrypts the plain text file D with a symmetric stream cipher using the encryption key K^{sym}_{encr} and a random initialization vector IV to get $F = \textbf{encrypt}(D, K^{sym}_{encr})$. It generates a hash for F and signs it using K^{pri}_{sign} to get $S_F = \textbf{sign}(F, K^{pri}_{sign})$.
- CCL generates a random $UUID$ for the file as its file handle.
- It creates the security metadata,

$$M =< UUID, SUUID, K^{pub}_{sign}, U_{owner} >$$

as shown in Figure 4. This metadata is signed using the owner's private key, $K^{pri}_{U_{owner}}$ to get

$$S_M = \textbf{sign}(M, K^{pri}_{owner})$$

- CCL then uploads the encrypted file, plain text security metadata and their signatures $<S_M, M, S_F, F>$ to the Cryptonite repository service.
- The File Manager of the Cryptonite repository verifies the signatures of the security metadata and the file using owner's public key, K^{pub}_{owner}, retrieved from the PKI using U_{owner}. It then inserts the received contents into Cloud storage service using the $UUID$ file handle passed for the file.
- The File Manager also makes an entry of the signed operation message in the Audit Manager's log. It then sends an acknowledgement of successful write back to the CCL with the original request message signed using Cryptonite's private key.
- CCL verifies the signature of the response against the public key of the Cryptonite service to validate that the requested operation has been performed.

STEP 3: Update *StrongBox*

- CCL adds an entry in the *StrongBox*list for the created file's UUID and its IV.
- It then reencrypts the *StrongBox*using the symmetric key K^{sym}_{encr} (same as the one used for encrypting the files) and a randomly generated $IV_{StrongBox}$.
- CCL then signs the *StrongBox*using the owner's private key $K^{pub}_{U_{own}}$, and uploads the signature and *StrongBox* contents to the Cryptonite repository service.
- The Cryptonite service validates the signature, uses the existing *StrongBox*security metadata (if any) to validate that the owner is making the updates, adds audit entries and returns a signed acknowledgement to CCL that verifies successful update of the *StrongBox*.

STEP 4: Update Secure Index

- CCL creates an index entry that is a mapping from the metadata properties for the file, P, to the file handle $UUID$.
- CCL then encrypts this mapping using searchable encryption with a shared key generated from the authorized readers of the index entry.
- The encrypted mapping is digitally signed with owner's private key and sent to the Cryptonite service, where the Index Manager verifies and adds this mapping to the index.

6.3.2 GET Operation

The GET operation is used to retrieve an encrypted file from the Cryptonite server. The sequence diagram for this operation is shown in Figure 7 and described below.

- Any authorized reader can retrieve a file handle by searching over the secure index, as described later in this section

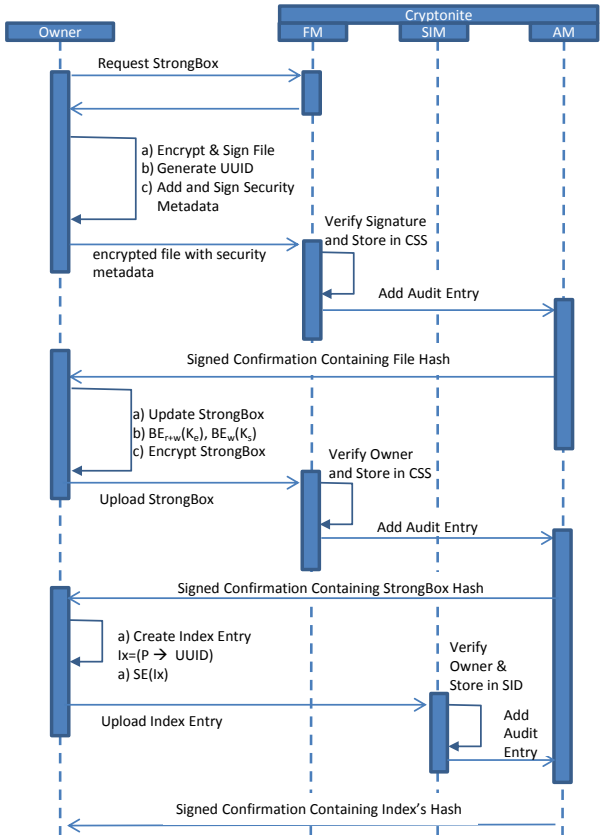


Figure 6: Sequence diagram for PUT operation. Shows interactions between the owner, via CCL, and Cryptonite service's File Manager (FM), Secure Index Manage (SIM) and Audit Manager (AM).

- The CCL retrieves the plain text security metadata of the file to extract the $SUUID$ of the *StrongBox*and downloads the *StrongBox*.
- The CCL decrypts the file, L, in the *StrongBox*which maps the file handles to their IV. Broadcast decryption can be used by any of the authorized readers to retrieve the key K^{sym}_{encr} required for this.
- The CCL downloads the encrypted file contents corresponding to $UUID$ by requesting the data repository, and uses the decrypted K^{sym}_{encr} and the IV to decrypt the contents of the file locally.

6.3.3 GRANT and REVOKE Operations

The GRANT and REVOKE operations are used to add or remove users from a file's access control list. Two scenarios exist while changing access permissions. First, is changing the access permission for a single file in a particular *StrongBox*, and second is adding or removing a user from the *StrongBox*group, thereby changing access permissions for all the files in the *StrongBox*. In the first case, the file needs to be moved to another *StrongBox*corresponding to the new access control list, potentially creating a new *StrongBox*. The entire file will have to be re-encrypted and signed using the file encryption and signature keys of the new strong box,

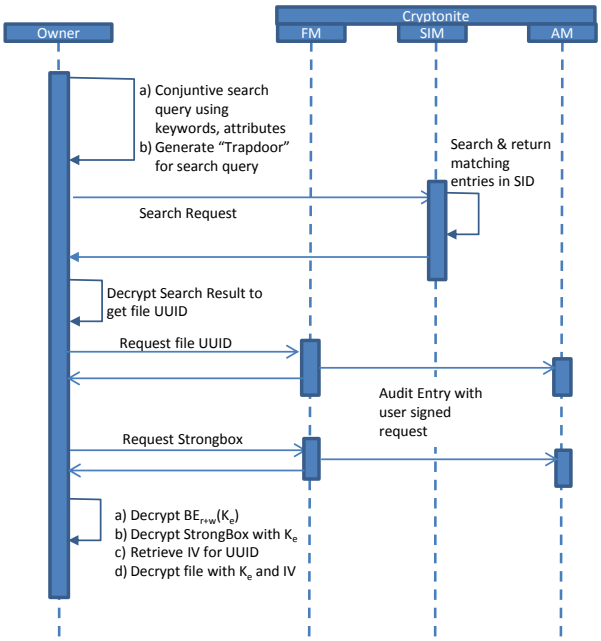


Figure 7: Sequence diagram for SEARCH followed by GET operations.

and the file's security metadata updated accordingly. In the second scenario, updating the permission will require changing the symmetric file encryption key, K^{sym}_{encr}, that causes a costly re-encryption of all the files present in that *StrongBox*. Instead, we use the "Lazy Revocation" strategy, describe earlier, wherein the data is not re-encrypted unless it is updated by an authorized user. The revoked user will have read access to the data as long as it is not updated and re-encrypted by someone else. This does not violate security since the user already had access to the files in the *StrongBox*. Once the file is updated, it is re-encrypted and signed with a new set of keys and the revoked user does not have future access to the updated file.

Newly generated keys for encryption and signature are created using key rotation describe before. These new keys are themselves re-encrypted using broadcast encryption according to the new access control list and added to the *StrongBox*. There may be some files in the *StrongBox*file list that are encrypted and signed with the new keys while others are still using older keys as a result of lazy revocation. During decryption, an authorized reader can obtain the latest encryption keys from the *StrongBox*and use key rotation mechanism to generate the previous versions of the keys.

6.3.4 SEARCH Operation

The SEARCH operation is used to locate files indexed in the secured index using a set of metadata attributes such as owner, modified date, file name, keywords, and so on. This operation uses searchable encryption described before and allows searching within an encrypted file without decrypting the entire file or revealing the contents of the file to the searching entity. This design work is in progress.

7. DISCUSSION AND FUTURE WORK

The Cryptonite service protects data confidentiality by performing client side encryption before data is stored in the repository. While we trust the Cryptonite service to perform operations on behalf of the client, this does not violate integrity requirements because, the Audit manager on the Cloud maintains a log of requested operations signed by the end user and also returns signed acknowledgments back to the end user for the operations. These acknowledgments can be used by the user to prove any unauthorized changes done by the Cloud provider or repository to a particular file, allowing detection of unauthorized modifications or even deletions *("Trust, but verify")*. For e.g., if a file is incorrectly deleted, the user will be able to produce a signed acknowledgment from the repository for the latest update confirmation while there will not be a signed message from an authorized writer requesting a delete operation.

The Cryptonite design has certain gaps with respect to the requirements. The security metadata associated with the file is stored in plain text, specifically the SSUID for the associated *StrongBox*. The untrusted Cloud provider can potentially parse through all the files stored in the repository and detect which files share the same *StrongBox*. This can reveal the correlation between the different files and their common access permissions enabling an attacker to perform focused attacks instead of exploratory ones.

In the proposed system, we have not addressed the issues of write serialization and file locking, and trust the Cryptonite service to handle contention. However we need to develop explicit locking mechanisms so that the end user has control over it. We also do not address the issue of random access, both for read as well as write, to the encrypted file. This can be a big issue for large sized files as decrypting/encrypting the entire file every time will be very inefficient.

As part of future work, we plan to address these gaps in our design and *implement the Cryptonite design* on top of an existing public Cloud storage service to support the USC Campus Microgrid project.

8. CONCLUSIONS

In this article, we have described emerging needs for increased data security and privacy for scientific and engineering datasets hosted on Cloud repositories. Our requirements have been motivated by a real world, ongoing multi-disciplinary project from the smart power grid domain. We have presented a design for the Cryptonite repository that addresses most of our desiderata and can work on top of existing public Cloud storage services without any specialized requirements. We leverage existing and proven security techniques, and combine them into this novel secure data repository architecture. We expect to put this design into practice and implement a data repository service and client libraries to support our project.

9. REFERENCES

[1] M. Abdalla, M. Bellare, D. Catalano, E. Kiltz, T. Kohno, T. Lange, J. Malone-Lee, G. Neven, P. Paillier, and H. Shi. Searchable encryption revisited: Consistency properties, relation to anonymous ibe, and extensions. In V. Shoup, editor, *Advances in Cryptology - CRYPTO 2005*, volume 3621 of *Lecture Notes in Computer Science*, pages 205–222. 2005.

[2] S. Aman, Y. Simmhan, and V. K. Prasanna. Improving energy use forecast for campus micro-grids

using indirect indicators. In *International Workshop on Domain Driven Data Mining (DDDM)*, 2011.

[3] N. Attrapadung, K. Kobara, and H. Imai. Broadcast encryption with short keys and transmissions. In *ACM Digital Rights Management Workshop*, 2003.

[4] M. Backes, C. Cachin, and A. Oprea. Lazy revocation in cryptographic file systems. *Security in Storage Workshop, International IEEE*, 0:1–11, 2005.

[5] J. Baek, R. Safavi-naini, and W. Susilo. Public key encryption with keyword search revisited. In *Computational Science and Its Applications*, 2008.

[6] F. Bao, R. H. Deng, X. Ding, and Y. Yang. Private query on encrypted data in multi-user settings. In *Information Security Practice and Experience*, 2008.

[7] C. Cachin, I. Keidar, and A. Shraer. Trusting the cloud. *SIGACT News*, 40:81–86, June 2009.

[8] Cloud Security Alliance. Security Guidance for Critical Areas of Focus in Cloud Computing V2.1. Technical report, December 2009.

[9] R. Curtmola, J. A. Garay, S. Kamara, and R. Ostrovsky. Searchable symmetric encryption: improved definitions and efficient constructions. In *Computer and Communications Security*, 2006.

[10] A. Fiat and M. Naor. Broadcast encryption. In D. Stinson, editor, *Advances in Cryptology*, volume 773 of *Lecture Notes in Computer Science*, pages 480–491. Springer Berlin / Heidelberg, 1994.

[11] W. E. Freeman and E. L. Miller. Design for a decentralized security system for network attached storage. In *IEEE Symposium on Mass Storage Systems*, 2000.

[12] J. A. Garay, J. Staddon, and A. Wool. Long-lived broadcast encryption. In *International Cryptology Conference on Advances in Cryptology*, pages 333–352, 2000.

[13] E.-J. Goh. Secure indexes. 2003.

[14] E.-J. Goh, H. Shacham, N. Modadugu, and D. Boneh. SiRiUS: Securing remote untrusted storage. In *Network and Distributed System Security Conference (NDSS)*, 2003.

[15] D. Halevy and A. Shamir. The lsd broadcast encryption scheme. In *International Crytology Conference*, pages 47–60, 2002.

[16] B. S. K. Jr. Rsa digital signature scheme. 2005.

[17] M. Kallahalla, E. Riedel, R. Swaminathan, Q. Wang, and K. Fu. Plutus: Scalable secure file sharing on untrusted storage. In *USENIX Conference on File and Storage Technologies*, 2003.

[18] S. Kamara and K. Lauter. Cryptographic cloud storage. In *Financial Cryptograpy and Data Security Conference*, 2010.

[19] U. M. Maurer. Modelling a public-key infrastructure. In *European Symposium on Research in Computer Security*, 1996.

[20] E. L. Miller, D. D. E. Long, W. E. Freeman, and B. C. Reed. Strong security for distributed file systems. In *IEEE International Performance, Computing and Communications Conference (IPCCC)*, 2001.

[21] P. Mohan, V. N. Padmanabhan, and R. Ramjee. Nericell: rich monitoring of road and traffic conditions using mobile smartphones. In *Conference On Embedded Networked Sensor Systems*, 2008.

[22] M. Newborough and P. Augood. Demand-side management opportunities for the uk domestic sector. *Generation, Transmission and Distribution, IEE Proceedings-*, 146(3):283 –293, may 1999.

[23] B. Plale, J. Alameda, B. Wilhelmson, D. Gannon, S. Hampton, A. Rossi, and K. Droegemeier. Active management of scientific data. *IEEE Internet Computing*, 9:27–34, 2005.

[24] R. A. Popa, J. R. Lorch, D. Molnar, H. J. Wang, and L. Zhuang. Enabling security in cloud storage slas with cloudproof. Technical report, Microsoft Research, May 2010.

[25] E. L. Quinn. Smart metering and privacy: Existing laws and competing policies. *SSRN eLibrary*, 2009.

[26] R. L. Rivest, K. Fu, and K. E. Fu. Group sharing and random access in cryptographic storage file systems. Technical report, MasterŠs thesis, MIT, 1999.

[27] Y. Simmhan, S. Aman, B. Cao, M. Giakkoupis, A. Kumbhare, Q. Zhou, D. Paul, C. Fern, A. Sharma, and V. Prasanna. An informatics approach to demand response optimization in smart grids. Technical report, Computer Science Dept., Univ. of Southern California, 2011.

[28] Y. Simmhan, A. Kumbhare, B. Cao, and V. K. Prasanna. An analysis of security and privacy issues in smart grid software architectures on clouds. In *International Cloud Computing Conference (CLOUD)*. IEEE, 2011.

[29] Y. Simmhan, V. Prasanna, S. Aman, S. Natarajan, W. Yin, and Q. Zhou. Towards data-driven demand-response optimization in a campus microgrid. In *ACM Workshop On Embedded Sensing Systems For Energy-Efficiency In Buildings*. ACM, 2011.

[30] G. Singh, S. Bharathi, A. Chervenak, E. Deelman, C. Kesselman, M. Manohar, S. Patil, and L. Pearlman. A metadata catalog service for data intensive applications. In *Proceedings of the 2003 ACM/IEEE conference on Supercomputing*, SC '03, pages 33–, New York, NY, USA, 2003. ACM.

[31] D. X. Song, D. Wagner, and A. Perrig. Practical techniques for searches on encrypted data. In *IEEE Symposium on Security and Privacy*, 2000.

[32] P. Stanton. Securing data in storage: A review of current research. *CoRR*, cs.OS/0409034, 2004.

[33] Y. Tang, P. P. C. Lee, J. C. S. Lui, and R. Perlman. Fade: Secure overlay cloud storage with file assured deletion. In O. Akan, P. Bellavista, J. Cao, F. Dressler, D. Ferrari, M. Gerla, H. Kobayashi, S. Palazzo, S. Sahni, X. S. Shen, M. Stan, J. Xiaohua, A. Zomaya, G. Coulson, S. Jajodia, and J. Zhou, editors, *Security and Privacy in Communication Networks*, volume 50. 2010.

[34] W. Treese. Putting it together: the home area network. *Networker*, 4:11–13, 2000.