Norton Zone: Symantec's Secure Cloud Storage System

Walter Bogorad
Symantec Inc
walter_bogorad@symantec.com

Scott Schneider
Symantec Research Labs
scottsch@alum.mit.edu

Haibin Zhang
University of North Carolina at Chapel Hill
haibin@cs.unc.edu

***Abstract*—Cloud storage services are the way of the future, if not the present, but broad adoption is limited by a stark trade-off between privacy and functionality. Many popular cloud services provide search capabilities, but make only nominal efforts to keep user data fully private. Alternatives that search private user data on an untrusted server sacrifice functionality and/or scalability.**

We describe Norton Zone, Symantec's secure and scalable public storage system based on our *valet security model*. Whereas most commercial cloud storage systems secure user data with access control and legal mechanisms, Zone's cryptographic techniques provide proven privacy guarantees. This gives users an extra layer of security without compromising functionality. Zone's performance is comparable to unencrypted cloud storage systems that support search and sharing. We report on the design of Zone and the lessons learned in developing and deploying it in commercial, distributed datacenters scalable to millions of users.

I. Introduction

Both consumers and enterprises are increasingly drawn toward public cloud storage systems, but trade-offs between privacy and functionality keep many from uploading their data to the cloud. Today's sophisticated users are often unwilling to sacrifice either one.

A *fully featured* cloud storage system [28] must provide data confidentiality and integrity, efficient data sharing and searching, and service availability and reliability. However, despite significant effort from both industry and academia, a fully functional and scalable cloud storage system is not yet practical.

Commercial, secure cloud storage systems. Most of the existing commercial "secure" cloud storage systems use encryption to protect user data. These systems can be roughly divided into two categories. One is *off-premise* security (data in the service provider's control) and the other is *on-premise* security (data in the customer's control). The most popular file hosting and sharing services, such as Dropbox [11], Box [8], Microsoft OneDrive [32], and Google Drive [22], fall into the first category. These services generate and keep users' keys. Data security and privacy rests on trust in the service and its employees rather than cryptographic guarantees. One simple way to determine if a service belongs to this category is to see if it supports password reset. This is only possible if the service has its users' keys. Off-premise security has been strongly criticized for the fact that the service can see user data and can potentially leak user keys or the keys for encrypting them ("a vicious circle"). These keys are protected only by legal and physical mechanisms.

Some less popular file hosting and sharing services, such as Mega [31], SpiderOak [40], and Wuala [44], provide on-premise security. These systems guarantee much better user privacy. They simply allow clients to encrypt their own data and then upload the ciphertext. These services have no access to user plaintext, so they cannot search user data with a strictly server-side implementation. None support encrypted search, perhaps because of its complexity or perhaps because of the computational requirements it puts on the client. This requirement is particularly prohibitive for devices such as tablets and mobile phones.

Techniques from academia. The academic world has put significant effort into building techniques for securing data in the cloud. Such systems tend to be prototypes and most address only one or two specific problems for cloud storage systems. Many challenges inhibit real-world deployment of academic techniques:

TECHNIQUE COMPATIBILITY. Techniques such as encrypted search, convergent encryption [4, 10] (to achieve secure deduplication), and secure file sharing are usually considered in isolation. Only a handful of conceptual constructions (e.g., [28]) are proposed to attain all of these goals. Though these techniques may work well by themselves, they are not necessarily compatible with each other and often complicate each other.

SEARCH FUNCTIONALITY. The search patterns enabled by the existing "practical" encrypted search schemes are limited, mainly focusing on keyword search and simple derivatives. However, customers demand the same experience as with unencrypted services, with metadata search, faceted search, disjunctive and conjunctive search, wildcard search, autocomplete, etc. Moreover, enterprise search engines are typically complex and even distributed across machines and applications. Note that fully homomorphic encryption [16] and oblivious RAM [21] are general enough to support this range of functionality, but are highly inefficient.

SPACE OVERHEAD. Though disk is becoming less expensive, space is still extremely precious in cloud computing. Two space factors inhibit the wide adoption of secure storage systems. First, search indexes have private data and must also be encrypted. Encrypted search often takes significantly more space. For instance, to encrypt SQL queries, CryptDB [36] has $4.21\times$ space overhead. Second, when files are shared, the recipients must also be able to search these documents. If the system redundantly indexes shared documents, index space

will quickly eclipse that of the indexed documents.

MOBILE ACCESSIBILITY. Modern cloud storage systems make data accessible from many platforms. Making secure data available on mobile devices is difficult, since they have limited computational ability.

SCALABILITY. Scalability is an important goal for practical cloud storage systems. Many schemes are not practical simply because they fail to scale. As a fully functional storage system is complex, theoretical analysis or even prototype implementation does not usually suffice to verify if the system is scalable. One must make sure the system can efficiently support millions of users simultaneously and work well on distributed datacenters.

This paper provides a meaningful trade-off between functionality and privacy. We advocate a new model—the valet security model, which lies in between off-premise security and on-premise security—and describes Norton Zone, a full-featured file sharing application secured by this model. Zone guarantees confidentiality, supports secure file sharing and encrypted search, and provides high availability and scalability. In particular, it is designed to handle frequent file sharing and provides all the search patterns that other commercial search engines implement. It is practical in the sense of computational and space efficiency and it adds moderate overhead compared to unencrypted, conventional file systems. It is highly scalable in the sense that it can support millions of users.

Zone's implementation included secure sharing, interfaces on Windows, Mac, web, and mobile, and many other features common to cloud storage platforms, plus enhanced security guarantees from our valet security model. We implemented and tested secure searching functionality, but we did not have time to integrate it.

Zone started production in May 2013. At the peak time Zone had about 300,000 accounts. It was terminated in August 2014 because the company decided to focus on its key strengths in security and information management and deprioritize its engagement in the file sync and share market [33]. We report on the design of Zone and the lessons learned in developing it.

II. THE VALET SECURITY MODEL

The distinctive capabilities of Zone rest on our new security model. Off-premise security, nearly universal in commercial file sharing applications, supports development of rich functionality by giving the service provider full access to the user's plaintext. Our security model supports the same functionality (with a few exceptions), but with significant additional security guarantees. Its guarantees are not as strong as those of on-premise security—guarantees preferred in academia and by the highly security conscious; however, the valet security model trades off a small amount of functionality for considerable additional security.

We compare the valet security model to on-premise and off-premise security for the following categories of attacks: 1) An attacker (whether a malicious insider or an external adversary) accesses to server storage a limited number of times. 2) An attacker has repeated or continual access to server storage. 3) An attacker accesses server memory and storage a limited number of times, but cannot alter server executables. 4) An attacker (or a service provider who breaks trust) has persistent, full access to the server.

On-premise security places no trust in the server and can be secure against all of these attacks. Off-premise security trusts the server fully; thus, any breach of server security can be extremely damaging. The valet security model trusts the server in a limited scope. This allows it to provide full security against two categories of attacks and partial protection against one more.

The core principle of the valet security model is that the server only has access to a user's keys and plaintext while the user is logged in. The user generates and stores his own master key and gives it to the server on login. The server keeps this key, and other keys and any of the user's plaintext, only in memory and uses them only as directed. All data written to disk is encrypted (with keys secured by the master key, as described in §III). On logout, the server flushes the user's keys and plaintext from memory. If the user trusts the server on these two points, he may have full confidence in his data's security while logged out.

This need for trust lacks the elegant simplicity of the on-premise security model. It seems to come down to the same assumptions as the off-premise model: that the system is designed as advertised and implemented soundly, that good security practices are in place and effective against external attackers, and that employees are trustworthy. None of these can be guaranteed. If an off-premise system encrypts user data with system or user keys, it must also store these keys. An attacker who breaches security far enough to access ciphertext must only take a few more steps to get the plaintext. Cryptography in the off-premise security model raises the bar against intrusion only quantitatively.

In one respect, the valet security model is similar. The users need to trust a system in this model in a more limited way because that keys and plaintext are never stored on disk and are flushed from memory when the user logs out. This reduces the amount of the system that is vulnerable, but that part may still be subverted by poor implementation or by attackers. This is merely a quantitative protection. It is no different from an off-premise system with encryption: the whole system's security rests on a component that stores system and user keys, yet that component is not itself protected by cryptography.

Yet the valet security model also qualitatively reduces the scope of potential vulnerabilities. If the system is compromised, the attacker cannot access logged out users' data. No preceding point made the valet security model particularly noteworthy, but protection against two and a half of the four attack categories is a significant advance. It is a unique and valuable trade-off, supporting a range of functionality (encrypted search and other, arbitrary processing of user data) and also cryptographically provable security guarantees.

We will now discuss generally how systems built in the valet security model (and not Zone specifically) resist the attack categories listed above: 1) If an attacker accesses server

storage a limited number of times, he cannot access any user plaintext. All data is encrypted by a tree of keys whose root user key is not stored on the server. Adversary can find the amount of data each user has and, depending on the system, perhaps broken down by type. 2) Repeated access to server storage does not help an attacker much. He can see when each user adds, modifies, and removes data. A system in the valet security model could conceal even this metadata by appropriately encrypting its file system, though we did not implement that. 3) A moderately sophisticated attacker might be able to access memory and disk once or a few times (e.g. by subverting the system into misusing other users' keys), but not alter executables. In this case, he can pull logged in users' keys from memory and get (or alter) all their data. He can replace a logged out user's public key(s) so that, when other users share files with this user (or the server receives user data from some external system), the files or their encryption keys are encrypted with the substituted public key. He cannot, however, read data for any user that wasn't logged in during an intrusion (except for files shared with logged in users). This considerably limits the damage from moderately sophisticated attacks. 4) Suppose an attacker (perhaps the service provider itself) has continuous access to the system or, equivalently, he can replace running binaries. Such an attacker can get virtually all user data. The system still protects the small number of users who have not logged in since the system was first breached. This only benefits users that are willing to abandon their data so that others cannot access it (e.g. users who believe their data may have been requested by a subpoena). Otherwise, the valet security model is vulnerable to highly sophisticated attacks.

In general, the valet security model's encryption requirement need not slow down processing very much. In a hybrid encryption system, most encryption is symmetric and adds limited overhead to the accompanying file system and network operations; however, the impact will be disproportionate when loading data from particular file locations if the system must decrypt an entire file. §IV describes how we solve this.

In the valet security model, Zone aims to achieve confidentiality with a fully range of search modes, integrity, secure and efficient sharing, mobile accessibility, fault tolerance, scalability, and other security requirements in a time and space efficient manner.

III. Design of Zone

Norton Zone is a fully functional, secure, and scalable cloud storage system. It distinguishes itself from other systems in three aspects: First, it securely supports industry-level search engines allowing a variety of search patterns that other encrypted search systems do not enjoy. Second, it addresses space complications incurred by scalable sharing and searching and explores meaningful trade-offs between security, efficiency, and space overhead. Third, it also supports several novel and user-friendly applications that were not explored before, including securely processing incoming data for logged out customers and secure multi-tenant search. Zone is secure in our valet security model.

A. A Toy Framework with Secure Sharing

We begin with a description of a simple but less efficient method that only supports secure sharing.Each customer i has a password pw_i for logging into the system. We adopt the conventional logging method by verifying the customer's password hash hk_i. The server can then securely and deterministically derive a customer master key mk_i from pw_i. To enable secure file sharing, the service provider additionally generates an associated public/secret key pair (pk_i, sk_i). The provider further encrypts sk_i using the customer's master key mk_i to get symmetric ciphertext c_i, which is stored on disk. The system encrypts each user file f with a separate file key k_f to facilitate efficient file sharing and enable secure file deduplication. It encrypts a user's file key with pk_i and stores the resulting c_f^i (and the encrypted file) on disk. All decrypted keys (including pw_i, mk_i, and sk_i) exist only temporarily and only in memory. They will be deleted when the customer logs out or does not perform operations for a long time. Finally, the customer i's system identifier is $\mathsf{SI}_i = (pw_i, hk_i, pk_i, c_i)$.

Each time user i logs into the system, he or she needs to provide his or her password pw_i to get the access token Token_i. This access token has the form $(\mathsf{flag}_i, mk_i, sk_i)$, where flag_i is binary flag representing whether i is granted access to its file service, mk_i is deterministically generated from its password, and sk_i is obtained by decrypting c_i using mk_i. Token_i will be used to perform subsequent file operations (e.g. sharing files, decrypting and reading the user's own encrypted files, and retrieving files shared with the user).

When an originator i shares a file f with a recipient j, the system first retrieves c_f^i and decrypts it with sk_i to get the file key k_f. It then encrypts k_f with j's public key pk_j to get a ciphertext c_f^j. When j logs into the system and obtains his access token of the form $\mathsf{Token}_j = (\mathsf{flag}_j, mk_j, sk_j)$, the system can decrypt c_f^j using j's secret key sk_j and then decrypt the encrypted file and return f. The customer login password, its commands, and results returned can be transmitted via SSL or other secure means.

The file key k_f can either be derived from the file's metadata using a PRF (so that it does not need to be stored), by randomly and uniformly sampling from a particular space (in which case it is easier to implement and especially to support secure sharing), or from the hash of the file (to enable secure file deduplication). Throughout our implementation, we use a combination of the latter two—popular or large files generating keys via their hashes and otherwise via uniformly producing the keys.

This system satisfies our valet security model. User information is fully secured against outside adversaries if the server is honest—all files are encrypted and only the intended recipients are able to get the shared files. If the server is compromised, data of the users who have not logged in will still be confidential. The service provider cannot respond to subpoenas, since it does not retain customer keys. Any request for the plaintext data must be made to the user.

However, there are at least *three* problems with the above framework. First, the public key encryption is relatively ex-

pensive. Each time a user shares files, the originator must perform a public key encryption operation. Furthermore, each time a recipient accesses a file (shared or not), the system must perform a public key decryption operation. Second, we note that for each shared file and each recipient, the system must store the file's key encrypted with the recipient's public keys. If a user shares many files with many users, this can use a large amount of space. For instance, assuming public keys are 1024 bits long (and symmetric keys are shorter), sharing 5,000 files with 2,000 users will use 1.192GB. Third, the framework does not extend to protecting and scaling search indexes. Indexes contain much of the indexed files' confidential information, so they must also be encrypted. They are typically 15%-30% of the size of indexed files, so if a shared file is indexed redundantly in many users' search indexes, the indexes' total size will quickly exceed the original files' size. All of these problems place severe practical limits on sharing files.

B. Zone Sharing

We use three novel techniques to address the problems.

Symmetric cryptography for logged in users. To minimize the performace overhead from asymmetric cryptography, we change the toy model in a few ways: First, each user has an additional symmetric key k_i' that encrypts the user's file keys. When a user accesses his own file, the system performs a symmetric key operation to get k_f. Second, when sharing a file with a recipient j that is logged in, the system can encrypt k_f with k_j'. When a recipient is logged out, we must still use public key cryptography, but our third improvement is that the first time j accesses the file, we will re-encrypt k_f with k_j' so that subsequent file operations require only symmetric cryptography. This may also be done during idle time after the recipient logs in, if latency is more important than processing time and it is sufficiently likely that the recipient will open the file.

Sharing via combination keys. To reduce the space and time cost of sharing files to the *sum* (rather than the *product*) of the number of recipients and the number of files shared, we introduce another layer of encryption that uses *combination keys*. When a user shares a file, we first identify the set (or combination) of users that will be able to access the file (including the recipients, the sharing user, and any users that could already access the file). We create a symmetric combination key ck for this set of users, if one does not already exist. We then store an encrypted copy of ck for each user. For the originator and any other users that are logged in, we encrypt ck with each user j's k_j'. For all other users, we encrypt ck with their public keys. As above, when a recipient j logs in (and accesses a shared file), we can re-encrypt ck with k_j'. If ck was previously created for this set of users, then each user already has an encrypted copy of ck. We then encrypt the keys of the shared files (the file keys—not the files themselves) with ck. We store them in a central location, so that any user with ck can access and decrypt these file keys. Now we return to our example. If a user shares 5,000 files with 2,000 users, with 128-bit symmetric cryptography, the space for storing keys (we still assume users are logged out) is 0.0025GB, instead of 1.192GB.

The second purpose of introducing the combination key is to encrypt the *index key* which is used to encrypt the index.

Space-efficient search with sharing. To make our search indexes space-efficient, Zone creates one search index (and corresponding encryption key) per combination of users that share files. Suppose that Alice shares file f_1 with Bob and shares files f_2 and f_3 with Bob and Charlie. Bob shares file f_4 with Alice. Alice and Bob also have files they share with no one. Charlie has no files of his own. The system will then have these indexes: a) Index 1 for Alice and Bob: files f_1 and f_4; b) Index 2 for Alice, Bob, and Charlie: files f_2 and f_3; c) Index 3 for Alice only: Alice's private files; and d) Index 4 for Bob only: Bob's private files.

When a user searches, we must check several search indexes. For example, when Alice searches, we must check three indexes; when Bob searches, we must check three indexes; and when Charlie searches, we must check only one index. In general, we must check search indexes for every user combination to which a user belongs where there is a shared file. In general, this can be exponential in the number of users, but cannot be more than the number of files in the system.

Undersharing and *oversharing* are two techniques for reducing search time for users that share files with many combinations of users. They trade space or security for better query time. In undersharing, we eliminate an index and redundantly index its contents in several existing, less shared indexes. The union of these indexes' user combinations equals the eliminated index's user combination. For example, we might eliminate index 1 in the example above and add f_1 and f_4 to indexes 3 and 4. When Alice or Bob queries, they must check one fewer index. In oversharing, we merge an index with another index shared by a proper superset of the eliminated index's user combination. This unfortunately means that we must check access rights when we search. For example, to overshare index 1, we merge it with index 2. As with undersharing, Alice's and Bob's queries must check one fewer index. Charlie's queries check the same indexes, but we must test whether he should be able to see each search result from index 3. This can be especially expensive when the query term is common in documents Charlie cannot access. As this example shows, oversharing has both positive and negative effects on search time.

Zone algorithms. The Zone algorithms are depicted in Figure 1 using pseudocode.

C. The Underlying Searchable Encryption

Our goal is to provide encrypted search with speed, reasonable space overhead, security, and updatability. As we will discuss in §VI, no academic encrypted search algorithm can do this in a commercially accepted manner, mainly due to their stricter security model that does not trust the server. Our intermediate security model makes this possible.

A simple but impractical approach is to encrypt the whole search index, decrypt it into memory when the user logs in, and clear it from memory when the user logs out. Searches would

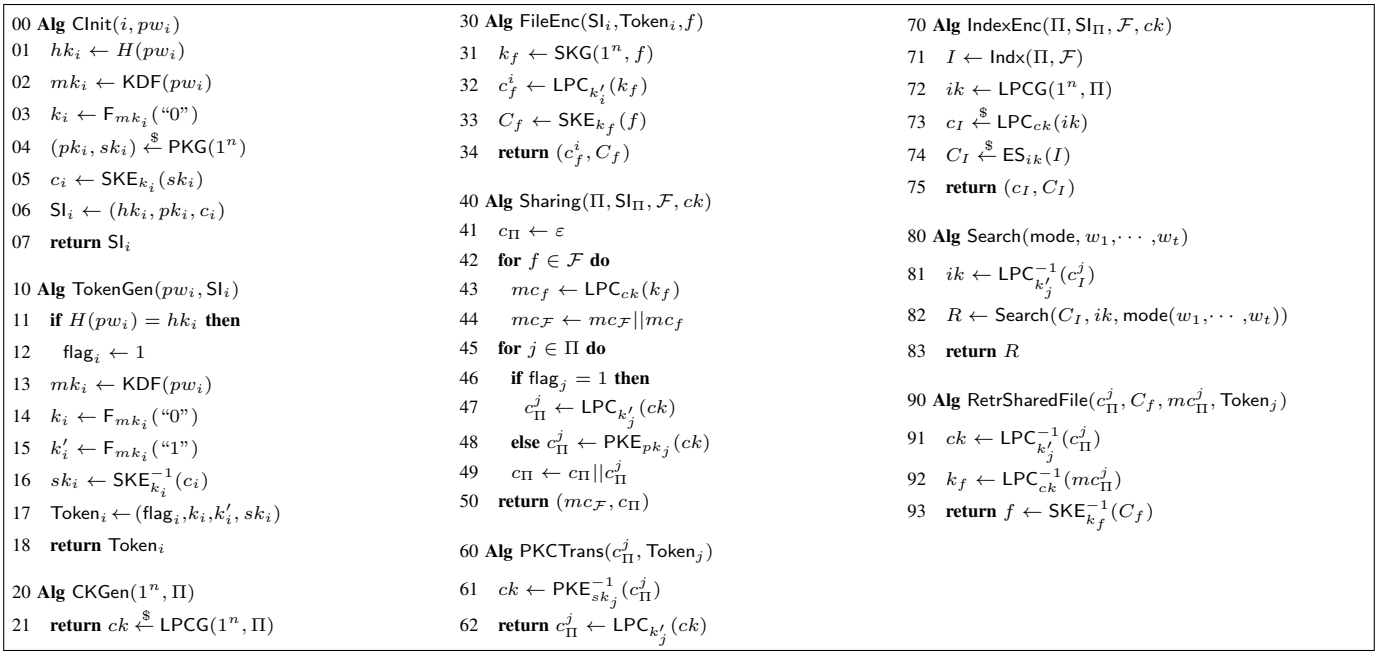


Fig. 1. **Zone algorithms.** Let $\mathsf{F}(\cdot)$ be a PRF. Let $\mathsf{KDF}(\cdot)$ be a key derivation function. Let $(\mathsf{SKG}, \mathsf{SKE}, \mathsf{SKE}^{-1})$ and $(\mathsf{LPCG}, \mathsf{LPC}, \mathsf{LPC}^{-1})$ denote a length-expanding encryption scheme and a length-preserving cipher, respectively. Let $(\mathsf{PKG}, \mathsf{PKE}, \mathsf{PKE}^{-1})$ be a public-key encryption scheme. To encrypt popular files, we employ message-locked encryption [4], where the key generation algorithm is deterministic and depends on files (see Line 31). Let Π denote customers who share files and let $\mathsf{SI}_\Pi = \{\mathsf{SI}_i\}_{i \in \Pi}$ denote their identifiers.

be fast, updates would be easy, and the index would be secure while the user is logged out. As a bonus, we could use an off-the-shelf search engine. Unfortunately, this design would create considerable latency between logging in and doing a first search.

Our approach does treat search index files like a black box, but it encrypts files in such a way that any part of the files can be independently decrypted. (Plaintext is still stored only in memory and flushed when the user logs out.) We run an off-the-shelf search engine and hook its disk reads and writes, respectively decrypting or encrypting disk contents. Using symmetric encryption, encryption takes not much longer than disk I/O. This preserves the search engine's functionality and speed without creating latency before the first search.

A search index consists of a number of files created by the core search engine. We encrypt index files with CTR$. All files in a single index share a 128-bit AES key. (We store this key in the same way as regular file keys, as described in the previous section.) Each file has its own random 128-bit IV. Although the core search engine can update its indexes, it never modifies an index file after initially writing it. Thus, our search indexes have indistinguishability from random bits.

We actually divide index files into chunks that are encrypted separately. (The reason for this is described in §IV.) Each chunk (except the last) is of a fixed, configurable size. It is encrypted with CTR$, using the file's IV plus the chunk's block offset in the ciphertext file (i.e. its byte offset divided by the block size). The probability that these counter ranges overlap is negligible, so this wrinkle does not change our system's security.

D. Confidentiality Analysis

Zone is designed to achieve "hedged" confidentiality goals: It is secure against outside adversaries, protecting everything including access tokens. Even if the server is compromised, it still provides security for users who are not logged in. See our full paper for analysis.

E. Zone's Other Functions and Features

Private and public sharing. The product supported two sharing modes. The first is "private sharing," which has been discussed in detail. It required a two-step initiation process in which a sharer sent invites to recipients and authenticated recipients had to accept or deny the invite. In case of invite acceptance, either the sharer or the recipients could upload files to the shared folder.

The other mode of sharing, "public sharing," was via web links. A sharer could create a URL to a resource to be shared and then send the URL to recipients out of band. Sharing via web links proved to be by far the most popular sharing use case, but it also was the most problematic from the security point of view. To raise the bar and adhere to the security guarantees of our in-between security model, we did the following: During the web link generation, the sharer had access to file key k_f. The system generated a one-time key (otk) and used it to encrypt k_f, i.e. $c_f = \mathsf{LPC}_{otk}(k_f)$. Then c_f and a reference to the file record (file id) were recorded into a special database table. The primary key of the record (an integer) and the 128-bit otk were parts of the URL.

Therefore, after the request was finished, the stored information (c_f, and the file id) was insufficient to decrypt the file preserving our system guarantees.

When the intended recipient sent a request to the provided URL, the system parsed the HTTP parameters, retrieved the primary key of the record and the otk, retrieved c_f and the file id from the database table and recovered $k_f = \mathsf{LPC}_{otk}(c_f)$. Using k_f, the file could be decrypted and sent to the recipients.

Another security issue with web links was logistical: they might get into wrong hands. This risk grew with time. To mitigate this, we provided UI controls so sharers could restrict the time when the link was valid and limit the maximum number of downloads allowed via the link.

Processing incoming data for logged out customers. Many applications require that the server process incoming data, even when the customer is logged out. One major scenario is backing up user content from other services, such as email and social networks. Startups may not have the best data retention practices and even Gmail has come close to losing user accounts [18]. Another scenario is processing this information for immediate use after the customer logs in. In the case of Zone, we would want index new content as it comes in so that a customer's first search does not have high latency or missing results.

Extending Zone and its security model to backup is straightforward. For each file (or unit of data that will be encrypted together), we generate a new file key, encrypt and store the file, and encrypt the file key with the customer's public key and store that. This minimizes the duration when the server has access to customer data.

We must make additional provisions for efficiently indexing a stream of content. If we simply index each incoming batch of files separately, we will wind up with a very large number of indexes, which can slow down information retrieval considerably. We cannot access the existing, main search index or any indexes for incoming files that we previously wrote. To mitigate this, we will fix a time or size threshold for incoming data. When the user searches, Zone will search all of these indexes. Furthermore, it will begin merging indexes as soon as the user logs in. Lucene already has functionality for merging many indexes while supporting queries, though it may need tuning for this use case.

Secure multi-tenant search. A multi-tenant search index stores data for multiple users. §III-B described how Zone uses multi-tenant indexes. In common practice, searches on a multi-tenant index filter out other users' results (as in our oversharing refinement) or search by the user's id as well as by keyword(s). A multi-tenant index may be more practical because it reduces space redundancy for common keywords and also because it may be faster and easier than juggling a large number of indexes, especially if these single-tenant indexes would each be small. Furthermore, the security penalty of oversharing in some applications (e.g., among an enterprise's employees) may be a lesser concern.

Consider the following scenario: an enterprise customer has a small number of divisions, each of which has its own data. Maintaining per-employee or per-division indexes for the enterprise will lead to space consumption because of redundancy in indexed keywords. Also, if an enterprise administrator wishes to search across all content, the time will be proportional to the number of the indexes. Multi-tenant search will allow more space-efficient and time-efficient search by maintaining a single index for a many users. (Note that it still has a performance penalty from filtering out results the searching user cannot access.) It is important to note that a naive implementation of multi-tenant search will compromise user privacy. Any user who can access the index can learn other users' data.

There are a number of ways to secure multi-tenant indexes, though they trade off some of the benefits listed above. We can process each user's indexed keywords with a PRF, using that user's key. Each keyword in Lucene's "dictionary" points to result information, such as the list of documents containing this keyword. We can encrypt this result information with a key based on the keyword's keyed PRF value. Lucene has hooks (different from the one Lucene Transform uses) that can support these modifications.

This allows us to effectively store multiple independent search indexes in one real index, gaining some time efficiency; however, because a keyword will have a different PRF value for each user, this scheme does not reduce redundancy in Lucene's dictionary. Furthermore, hashing keywords destroys some of Lucene's rich search functionality, such as fuzzy spelling and auto-complete. It also reintroduces sharing problems. As in §III-B, we can mitigate redundancy and performance problems with combination keys, undershared keys, and overshared keys.

F. Limitations and Discussion

This section discusses the limitations of Zone. One frequently mentioned limitation is Zone's weaker confidentiality properties, compared to an ideal cloud storage system. We have written enough about the tradeoff between functionality and privacy in our valet security model, but we have not explored similar tradeoffs. For example, a slightly different security model might keep the user's master key and other high level keys strictly on the client, giving the server only lower level keys. The client could decrypt file keys for the server as needed. This moves some computation from the server to the client and adds latency, as the client must request file key ciphertexts and send decrypted keys to the server. Alternatively, we might replace user i's k'_i with several keys for different types or folders of content. This limits the server's access to a logged in user's content, while reducing the number of round trips for key decryption (especially since users tend to work on related files, rather than a random sample of files from across the system). These alternatives make fine-grained, though still significant privacy-versus-functionality tradeoffs with Zone.

Zone does very little to guarantee the authenticity of user data. It would be easy to guarantee authenticity while the user is logged out, e.g. by storing file MACs. It would be interesting to explore how to integrate practical integrity-preserving techniques such as provable data possession [3], proof of retrievability [26], and Byzantine fault tolerance [24].

We favored conventional replication over erasure-coded replication due to bandwidth and system I/O consideration.

Another direction which we have not explored is to protect the customer anonymity from service provider in the valet security model. In our current design, the provider actually stores unencrypted user metadata (though it still cannot access encrypted content when the user is offline). Ideally, it could be enhanced by providing anonymity against the provider.

IV. Implementation

We implemented and deployed major components of Zone. Our implementation stack included Java with Tomcat, Spring, and Hibernate. The main components of our backend architecture were: 1) The application tier responsible for file upload, download, sync, and share. 2) The object store responsible for storing customer data on disk. 3) The metadata database (RDBMS) that contained user account information, a map from users to their files, and sharing information. In industry, a common technique for storing encrypted files on disk is to store encrypted keys as the first part of the encrypted file. However, since we re-encrypt file keys during sharing, we ruled out this approach and stored encrypted file keys in the metadata database. 4) The notification tier used to promptly notify clients of changes to their data. Clients maintain persistent connections to it. When a new event occurred at the application tier, notification servers alert the affected clients. This improved responsiveness of sharing and collaboration. 5) The search tier. We implemented the encrypted search tier and we will describe our implementation in detail below.

To achieve high scalability, we did not store session state at the application tier. Also, we did not use client IP address affinity at the load balancer; however, since we used HTTP 1.1 with persistent connections, the majority of sequential client requests arrived at the same application server. This implicit connection affinity helped performance by increasing the cache hit rate at the application tier. This design choice allowed us to achieve both good performance and good scalability with the penalty of some code complexity.

To achieve high availability and fault tolerance, we relied on the cross-datacenter replication capabilities of the object store and the metadata database and the high redundancy of application servers at each datacenter. The load balancer and application service tier did not persist any state, so we did not need to replicate sessions. The worst penalty of an application server failure was extra latency as the client re-established the connection. We relied on Global Server Load Balancing [23] to mitigate catastrophic failures at the datacenter level. If a datacenter did go down, the client's DNS lookup would resolve to an IP address of an active datacenter.

Encrypted search tier implementation. Our encrypted search implementation has two main challenges: to provide a commercial-grade suite of search functionality and to implement proven cryptographic security. Open source tools provide all of the former and a framework for the latter. Our modifications build proven security guarantees into the system.

The first open source tool, Lucene [1], is a very popular Java-based search engine. It provides efficient search for keywords, phrases, and wildcards, it ranks search results, and has many other features needed for commercial-grade search functionality. Lucene itself has limited support for scaling to multiple machines, but a number of tools (e.g. Solr [2] and Elasticsearch [12]) wrap this functionality around Lucene.

Most relevant to our encryption scheme is that, while Lucene indexes are dynamic (i.e. they efficiently support intermixed queries and index modifications), index files are static (i.e. never changed after they are initially written). A Lucene index consists of many segments. Each segment has an inverted index and other data structures that support efficient search. All of a segment's data files are static. A segment is also static, except that its documents may be deleted. New deletions are recorded in a new deleted documents file, so this is no exception to the rule that Lucene index files are static.

The other open source tool is Lucene Transform [30], a plug-in that hooks into Lucene's file I/O layer. It breaks every file up into chunks of configurable size and transforms each chunk, compressing or encrypting it (or both). Within a chunk, it can use a variety of modes of encryption, including CBC and CTR. It also caches decrypted chunks and provides a number of other useful services.

We made a number of changes to Lucene Transform. The first two implemented the security properties described in the design section.

- Lucene Transform did support CTR$, but each file reused the same initialization vector with all chunks. Therefore, the encryption mode for chunks (with respect to the overall file) was ECB, which is insecure. Our change modifies the IV for each chunk, ensuring that counter values used for CTR mode encryption are unique. We begin with an IV for the whole file. To compute a chunk's IV, we divide the chunk's byte offset in the ciphertext file by the cryptographic block size and add that to the file IV. We round this down, if necessary, and encrypt the chunk's first few bytes by XORing them with the last few bytes of the encrypted value of the chunk's IV.
- Lucene Transform stored an unencrypted CRC for each chunk. This was quite useful for debugging, so instead of removing it, we made it optional. The experiments in the next section do not have CRCs.
- We fixed deadlocks and a few other bugs; we reuse certain cryptographic objects that are expensive to initialize, which improved performance significantly; and we optionally use AES-NI.

Our implementations use SHA1 as the hash function for convergent encryption, PKCS#5 as the key derivation function, HMAC-SHA1 as the underlying PRF, RSA-1024 as the public key encryption algorithm, CBC with ciphertext stealing as the symmetric encryption mode to encrypt files, and CTR$ as the symmetric encryption mode to encrypt index segments.

Defending against various attacks. Zone also implemented practical approaches to defend against other threats such as access token misuse, browser session cookie stealing, and various denial of service (DoS) attacks. See our full paper.

V. Performance Evaluation

This section focuses on evaluation on share and search functionalities. Evaluation on basic operations can be found in our full paper. Our encrypted search tests use up to 400 MB of emails. This seems modest, but most test data (everything except email headers) is processed text. Although users commonly store many GB in cloud services, the vast majority of that is picture or video data; and in many file formats that carry text (e.g. PDF, Word), the actual text content is a small fraction. We believe these sizes cover the actual amounts of text data that real users upload to cloud services.

Index sharing evaluation. We ran a series of tests to explore various sharing schemes' space vs. time trade-offs. Different applications will have a wide variety of sharing patterns, so we created only a simple test setup that we hoped would yield a qualitative understanding of these trade-offs. We ran these tests (and those in the following section) on an Intel(R) Xeon(R) CPU E3-1270 V2 3.50GHz machine with 8 total cores and 16 GB of RAM. It had two hard drives: a 7,200 RPM conventional hard drive and a solid state disk. To isolate the impact of search-related disk operations, we created search indexes on the spinning drive and loaded input documents from the SSD.

Our experimental setup has four users, Alice, Bob, Charlie, and David, who share documents. Three test configurations used different total sizes of indexed documents. We used the same test machine described in §IV. The setup is detailed as follows: Each user has 1.72 MB, 8.6 MB, or 43 MB of unshared documents (for the three test configurations). The following pairs of users share 1.72 MB, 8.6 MB, or 43 MB of documents: Alice and Bob, Alice and Charlie, Bob and David, and Charlie and David. Alice, Charlie, and David share 0.84 MB, 4.2 MB, or 21 MB of documents, as do Bob, Charlie, and David. All users share 0.56 MB, 2.8 MB, or 14 MB of documents.

This adds up to 16 MB, 80 MB, or 400 MB of documents. Our experiments measure the total index space for all users and test query time for Alice. In Zone's default scheme, she must check 5 indexes (out of 11 total, for all users).

The sharing schemes we tested, from least to most sharing, are: 1) **Unshared**—Each user has his or her own index with all shared documents redundantly indexed. 2) **Undersharing**—We undershare the 3- and 4-user file sets. Each 3-user file set was duplicated into a 2-user file set and that of the remaining user. The 4-user file set was duplicated in two 2-user file sets. To query for Alice, we check 3 indexes. 3) **Zone**—Zone creates one search index per combination of users that share files. 4) **Oversharing**—We overshare the 2-user file sets. Most will go into into 3-user file sets, but files shared by Alice and Bob and Bob and David must go into the 4-user file set. In the latter case, we filter out search results for files not shared with Alice. To query for Alice, we check only 4 indexes. 5) **Single index**—A single search index serves all users. As with oversharing, this index must check access information.

Undershared and unshared indexes show the expected trade-offs with Zone: faster queries for greater space usage. Undersharing is a particularly good trade-off. Just 11-14% more space gets queries that are 21-35% faster. Moreover, unshared indexes look attractive, though with a higher sharing factor, they will often be out of the question.

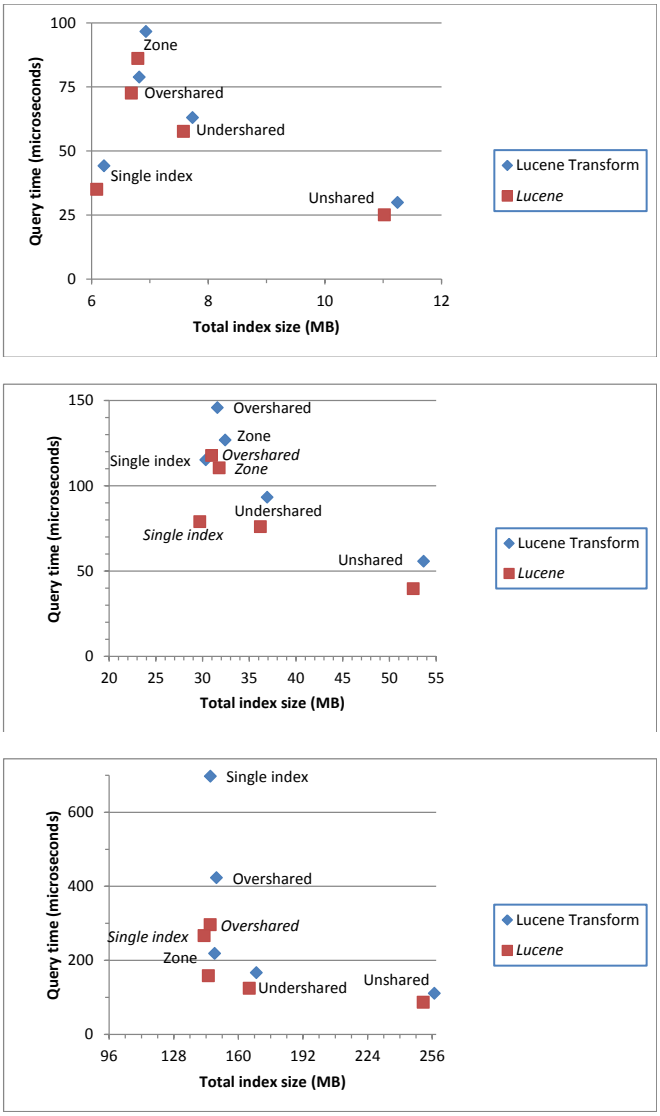


Fig. 2. Search with sharing when size varies.

Undersharing seems worthwhile when combined with a strategy for cherry-picking certain indexes to undershare. A simple strategy would be to undershare indexes that would produce the least redundant indexing. Eliminating small indexes may, however, have small time savings—if a small index is frequently used, it may be cached in memory. The best undersharing strategy may be application-specific. Overshared and single indexes show the expected speed improvement only for small indexes. With medium and large indexes, the benefit of checking fewer indexes is outweighed by the additional cost of filtering out search results the user cannot access. The worst case illustrates this: a user searches for a term that has only a few results in his documents, but many results in inaccessible

documents. In this case, we must filter out a potentially huge number of inaccessible results.

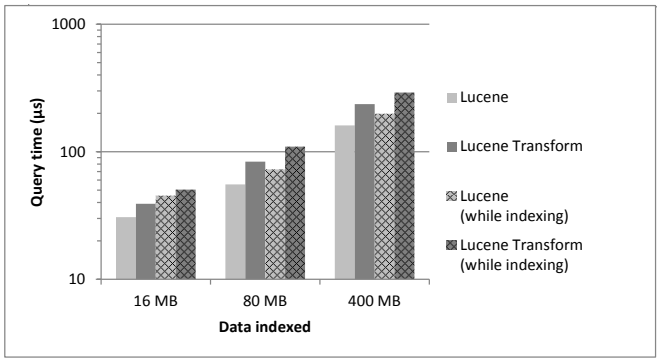
These tests show the pitfalls of oversharing, although a good oversharing strategy could conceivably be worthwhile, even on a very large system. It would need to be able to identify which indexes are small enough that the gain from checking fewer indexes would outweigh the cost of filtering. Also, other filtering strategies might be faster than ours.

Overshared and single indexes are slightly more space-efficient than Zone's indexes, despite the fact that each of these schemes indexes each document only once. This is because the size of one of Lucene's index components is proportional to the number of unique keywords. Thus, fewer bigger indexes use less space. Also, merging two large document sets gives a smaller benefit than merging two smaller document sets, because keyword storage is a smaller part of the overall index size. This explains why the space savings are smaller with medium indexes and tiny with large indexes.

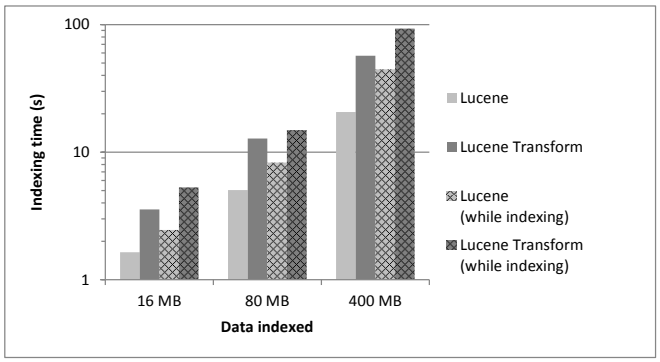
Encrypted search evaluation. We compared Zone's performance against regular, unencrypted Lucene on document sets of varying sizes, performing various operations. The document sets were 16 MB, 80 MB, and 400 MB of emails from the public Enron corpus [13]. One test timed indexing a document set using 8 threads. Another tested the number of queries the index could perform in one minute using 8 threads. The last simulated a real world scenario for a dynamic index: adding data to an existing index (using 8 threads) while simultaneously querying (using 8 threads). It added the same amount of data as was originally indexed. We ran each test 10 times. Lucene caches grow proportionately with the number of documents and keywords, on the order of a few bytes and a few bits each, respectively. We kept the Lucene Transform cache (of decrypted chunks) constant at a modest 128 kb.

Figure 3(a) and Figure 3(b) show that Lucene Transform increases query time only by roughly 50%, but increases indexing time 80% to 175%. Query overhead is quite small for 16 MB tests, probably because Lucene Transform caches a higher fraction of decrypted chunks. Indexing shows much larger overhead. The combined tests increase indexing times for Lucene and Lucene Transform by similar absolute amounts, mitigating the overhead there. Querying most likely has lower overhead than indexing as it requires many more disk seeks.

Table I shows Lucene Transform's space overhead, relative to unencrypted Lucene, to be quite modest — around 2%. It would be even lower with a larger chunk size. In ad hoc tests, AES-NI improved indexing time, but actually increased query time. Moving from Java to native code to decrypt each chunk took more time than was saved in decrypting more quickly. We were optimizing for queries over indexing, so we discarded this optimization.



(a) Query evaluation.



(b) Index evaluation.

Fig. 3. Encrypted search evaluation.

TABLE I
INDEX SIZES (IN MB) AND OVERHEAD. LT STANDS FOR LUCENE TRANSFORM.

	Indexing test			Combined test		
Data	Lucene	LT	Overhead	Lucene	LT	Overhead
16	6.0042	6.1273	2.05%	12.47	12.71	1.9%
80	29.373	30.006	2.158%	57.1	58.269	2.0%
400	141.54	144.60	2.16%	289	295	1.8%

VI. RELATED WORK

There is extensive research on secure network file systems with secure file sharing and secure file systems with encrypted search, but they are almost always considered independently of each other. Secure sharing and secure encrypted search turn out to complicate each other considerably. Only a handful of schemes (e.g., [28]) meet both of these goals, but they are just proofs of concept.

Most existing secure network file systems (e.g. [17, 20, 27]) only support limited or less scalable file sharing and they only support search on file metadata (filenames, creation dates, etc.), if at all. They are not designed for frequent file sharing; however, cloud computing, and especially social cloud storage, demand more efficient and scalable file sharing services.

The idea of sharing files at the group level is first used in Cryptfs [45] and subsequently in [27, 38]. Our idea of introducing the combination key for sharing files resembles an idea in Cepheus [15]: the "lockbox" for symmetric keys used to encrypt files. The key for a lockbox can be shared with different users. Our file sharing procedure is most similar to that of Plutus [27], where a lockbox stores encrypted file keys. The key for a lockbox is shared at the group level and file keys are used to encrypt files; but there are three differences. First, Plutus uses a lockbox to keep file keys, instead of files themselves, to prevent a potential vulnerability to known plaintext/ciphertext attacks. By contrast, we do so to

enable secure deduplication. Second, Plutus uses out-of-band communication to share the group key. Instead, we directly encrypt combination keys with the recipients' public keys or secret keys if they are logged in. Last, unlike the lockbox keys in Plutus, our combination keys also encrypt our index keys.

Existing encrypted search algorithms use a strong security model with an untrusted server. These include fully homomorphic encryption [16], oblivious RAM [21] , public key encryption with keyword search [7], searchable symmetric encryption (SSE) schemes (see [9, 41] and references therein), private information retrieval [34], etc. Briefly speaking, to keep user data private in this stricter security model, published encrypted search algorithms cannot meet both goals. At most, they support only one of the two. Despite a weaker security model, Zone is superior to practical systems such as CryptDB [36], MONOMI [43], and Mylar [37], in terms of performance, search functionality, and scalability.

VII. Conclusion

Motivated by the gap between the ideal of a scalable and fully functional cloud storage system that runs securely on untrusted servers and the present situation, we advocate a valet security model. This model lies between off-premise security and on-premise security. With this model, we designed a practical scheme, Zone, that enables efficient file sharing, supports a wide range of search patterns, minimizes space requirements, and provides easy accessability and reliability.

We have described our dedicated implementation in a modern datacenter designed to support millions of customers. As we have shown, despite our weaker security notion, designing and implementing such a system has proven extremely challenging. Security experts and cryptographers can sometimes lose sight of the big picture when they design and implement secure cloud storage schemes. We have therefore reported on the design of Zone and the lessons learned in its implementation. Zone is by no means the ultimate realization of the academic and commercial communities' goals; however, we believe that Zone is a very useful improvement on existing off-premise secure cloud storage systems, in both security and functionality. Moreover, it is our hope that our design and implementation could lead to an ideal, fully functional cloud storage system.

Acknowledgment

We thank anonymous reviewers for their comments. A number of people at Symantec helped considerably with our research. Sharada Sundaram contributed to the design of Zone. Krishna Pillai, Dinesh Ganesh, and Daniel Marino ran a number of tests for us. Much of Haibin's work was done at Symantec Research Labs. Haibin partly acknowledges NSF grant CNS 1228828, CNS 1330599, and CNS 1413996, as well as the Office of Naval Research grant N00014-13-1-0048.

References

[1] Apache Lucene. http://lucene.apache.org
[2] Apache Solr. http://lucene.apache.org/solr
[3] G. Ateniese, R. Burns, R. Curtmola, J. Herring, L. Kissner, Z. Peterson, and D. Song. Provable data possession at untrusted stores. *ACM CCS '07*.
[4] M. Bellare, S. Keelveedhi, and T. Ristenpart. Message-locked encryption and secure deduplication. *EUROCRYPT 2013*, 2013.
[5] M. Bellare, T. Ristenpart, and S. Tessaro. Multi-instance security and its application to password-based cryptography. *CRYPTO 2012*, 2012.
[6] M. Bellare and P. Rogaway. Encode-then-encipher encryption: How to exploit nonces or redundancy in plaintexts for efficient cryptography. *ASIACRYPT '00*.
[7] D. Boneh, G. Crescenzo, R. Ostrovsky, and G. Persiano. Public key encryption with keyword search. *EUROCRYPT 2004*, 2004.
[8] Box, Secure content-sharing. http://www.box.com/
[9] R. Curtmola, J. Garay, S. Kamara, and R. Ostrovsky. Searchable symmetric encryption: Improved definitions and efficient constructions. *CCS 2006*, 2006.
[10] J. Douceur, A. Adya, W. Bolosky, D. Simon, M. Theimer. Reclaiming space from duplicate files in a serverless distributed file system. *ICDCS 2002*, 2002.
[11] Dropbox, a file-storage and sharing service. http://www. dropbox.com
[12] Elasticsearch. http://www.elasticsearch.org/overview/elasticsearch
[13] Enron Email Dataset. http://www.cs.cmu.edu/∼./enron
[14] Flůd backup system. http://flud.org/wiki/Architecture
[15] K. Fu. Group sharing and random access in cryptographic storage file systems. Master thesis, MIT, June 1999.
[16] C. Gentry. Fully homomorphic encryption using ideal lattices. *STOC*, 2009.
[17] E. Geron and A. Wool. CRUST: cryptographic remote untrusted storage without public keys. *Int. J. of Info. Sec.*, vol 8, issue 5, pp. 357–377, October 2009.
[18] Gmail back soon for everyone. http://gmailblog.blogspot.com/2011/02
[19] GNUnet, a framework for secure peer-to-peer networking. https://gnunet.org
[20] E.-J. Goh, H. Shacham, N. Modadugu, and D. Boneh. SiRiUS: Securing remote untrusted storage. *NDSS 2003*.
[21] O. Goldreich and R. Ostrovsky. Software protection and simulation on oblivious RAMs. *Journal of the ACM*, 43(3): 431–473, 1996.
[22] Google Drive. http://drive.google.com
[23] GSLB (Global Server Load Balancing) Functional specification and Design Document. https://cwiki.apache.org
[24] J. Hendricks, G. Ganger, and M. Reiter. Low-overhead Byzantine fault-tolerant storage. *SOSP '07*, 2007.
[25] M. Islam, M. Kuzu, and M. Kantarcioglu. Access pattern disclosure on searchable encryption: Ramification, attack and Mitigation. *NDSS 2012*.
[26] A. Juels and B. Kaliski. PORs: Proofs of retrievability for large files. *ACM CCS '07*, 2007.
[27] M. Kallahalla, E. Riedel, R. Swaminathan, Q. Wang, and K. Fu. Plutus: scalable secure file sharing on untrusted storage. *FAST 2003*, 2003.
[28] S. Kamara and K. Lauter. Cryptographic cloud storage. *FC Workshops*, 2010.
[29] J. Li, M. Krohn, D. Mazières, and D. Shasha. Secure untrusted data repository (SUNDR). *OSDI*, 2004.
[30] Lucene transform. http://code.google.com/p/lucenetransform
[31] Mega. The privacy company. https://mega.co.nz
[32] Microsoft Onedrive. https://onedrive.live.com
[33] Norton Zone end of life FAQ. http://archive.today/KFGLq
[34] R. Ostrovsky and W. Skeith. A survey of single-database private information retrieval: Techniques and applications. *PKC 2007*, 2007.
[35] PBKDF2. PKCS #5: Password-based cryptography specification version 2.0. http://tools.ietf.org/html/rfc2898
[36] R. Popa, C. Redfield, N. Zeldovich, and H. Balakrishnan. CryptDB: Protecting Confidentiality with Encrypted Query Processing. *SOSP 2011*.
[37] R. Popa *et al*. Building web applications on top of encrypted data using Mylar. *NSDI '14*.
[38] E. Riedel, M. Kallahalla, and R. Swaminathan. A framework for evalauting storage system security. In *FAST*, 2002.
[39] P. Rogaway, M. Bellare, and J. Black. OCB: A block-cipher mode of operation for efficient authenticated encryption. In *ACM TISSEC*, 6:3, pp. 365–403, 2003.
[40] Spideroak. https://spideroak.com
[41] E. Stefanov, C. Papamanthou, and E. Shi. Practical dynamic searchable symmetric encryption with small leakage. *NDSS 2014*.
[42] TPC Benchmarks. http://www.tpc.org/information/benchmarks.asp
[43] S. Tu, M. F. Kaashoek, S. Madden, and N. Zeldovich. Processing Analytical Queries over Encrypted Data. *VLDB*, 2013.
[44] Wuala. Wuala, secure cloud storage. http://www.wuala.com
[45] E. Zadok, I. Badulescu, and A. Shender. Cryptfs: A stackable vnode level encryption file system. Tech Report CUCS-021-98. Columbia Univ. 1998.