Cryptonite: A Secure and Performant Data Repository on Public Clouds

Alok Kumbhare[†], Yogesh Simmhan[∗], Viktor Prasanna[∗†]
[∗]Department of Electrical Engineering
[†]Computer Science Department
University of Southern California, Los Angeles Ca 90089
Email: {kumbhare, simmhan, prasanna}@usc.edu

Acknowledgement

This material is based upon work supported by the Department of Energy under Award Number DE-OE0000192.

Cryptonite: A Secure and Performant Data Repository on Public Clouds

Alok Kumbhare[†], Yogesh Simmhan[*], Viktor Prasanna[*†]
[*]*Department of Electrical Engineering*
[†]*Computer Science Department*
University of Southern California, Los Angeles Ca 90089
Email: {kumbhare, simmhan, prasanna}@usc.edu

***Abstract*—Cloud storage has become immensely popular for maintaining synchronized copies of files and for sharing documents with collaborators. However, there is heightened concern about the security and privacy of Cloud-hosted data due to the shared infrastructure model and an implicit trust in the service providers. Emerging needs of secure data storage and sharing for domains like Smart Power Grids, which deal with sensitive consumer data, require the persistence and availability of Cloud storage but with client-controlled security and encryption, low key management overhead, and minimal performance costs. *Cryptonite* is a secure Cloud storage repository that addresses these requirements using a *StrongBox* model for shared key management. We describe the *Cryptonite* service and desktop client, discuss performance optimizations, and provide an empirical analysis of the improvements. Our experiments shows that *Cryptonite* clients achieve a 40% improvement in file upload bandwidth over plaintext storage using the Azure Storage Client API despite the added security benefits, while our file download performance is 5 times faster than the baseline for files greater than 100MB.**

***Keywords*-Cloud data storage; Secure data sharing;**

I. Introduction

Cloud computing has gained immense following in the business and scientific communities due to the ease of access and management, combined with a pay-as-you-go model for elastic on-demand resource usage. Of the diverse Cloud service offerings, Cloud storage is by far the more popular, whether used as an IaaS like file or BLOB store, PaaS like SQL Azure or SaaS like DropBox[1]. These storage services are used for online content distribution, synchronization across devices, and for document sharing by enterprises and scientific collaborations.

Despite its success, there is heightened concern about the security and privacy of Cloud-hosted data [1] to many eBusiness and eSciences that are considering shifting to Clouds. These arise from reasons: shared Cloud infrastructure for storing data from different organizations, co-location of multi-tenant applications with the storage services, and malicious insiders at Cloud providers getting access to data. These do not presuppose ill-intentions by Cloud providers, who make a best effort (but not guarantees) of data security, but are rather the consequence of the "shared infrastructure managed by third party providers" model of Clouds.

Existing Cloud storage services only provide basic access control mechanisms[2], and the limited research on secure, shared Cloud repositories often require extensive deployment of infrastructure services that undermines their manageability[2]. Specifically, there are two classes of current solutions. One uses simple file en/decryption on the client side, managed through shared keys, such that only encrypted files are hosted in Clouds. While this works for a small number of files and users, key sharing and management becomes unsustainable as the number of users sharing data grows. A second solution is to allow the Cloud providers or a third-party offer more advanced security services and manage user credentials such as dropbox. However, this approach still "trusts" an external service provider to secure their plaintext data.

Consider the following scenario to motivate our work. Smart Power Grids are giving utilities unprecedented access to realtime power consumption data on individual customers. Managing and analyzing this data requires large-scale compute and storage resources such as offered by Clouds[3]. As part of the Los Angeles Smart Grid Demonstration project, the University of Southern California (USC) campus is serving as a testbed to research Smart Grid software technologies and human behavior to study power usage. This multi-disciplinary research has several departments and researchers involved, and data is collected on buildings, occupants, campus events, classroom schedules, and weather, in addition to Smart Meter and sensor data. Research regulations require that the different owners of this shared dataset – including the campus power consumers, who number in the thousands – have final control on which research groups have access to their datasets [4]. A Cloud-hosted data repository to hold personally identifiable information for scientific and operational needs should enforce this single owner/multiple writers/multiple readers permission on individual data files with low management overhead, while ensuring that plaintext data is not stored in the Cloud.

This motivates the need for a *data storage service which retains the persistence and availability offered by public Cloud storage but with security and encryption that is controlled by the clients*, *a low key management overhead*,

[1]www.dropbox.com

[2]Amazon Identify and Access Management aws.amazon.com/iam

and compatibility with existing storage service interfaces while not sacrificing performance and scalability.

In this paper, we present *Cryptonite*, a secure Cloud storage respository that addresses these requirements. Our design introduced earlier [5] is validated here, to wit we present an implementation of the *Cryptonite* service and desktop client on the Microsoft Windows Azure Cloud platform. We also discuss optimizations to overcome data security overheads and provide an empirical analysis of the improvements. Specifically, our contributions are as follows:

- We implement *Cryptonite*, a secure data repository that runs within Azure and provides service APIs compatible with existing Cloud storage services,
- We present pipelined and data parallel performance optimizations to reduce the security overhead caused through encryption and key management, and
- We experimentally evaluate *Cryptonite* on the Azure Cloud, compare it with baseline client storage access, and demonstrate the efficacy of our optimizations.

The rest of the paper is organized as follows. In Section II we review the *Cryptonite* design. Section III describes the service and client implementations. We discuss performance optimizations in Section IV, and empirically analyze them in Section V. Related work is compared in Section VI. We present our conclusions in Section VII.

II. Background: *Cryptonite* Design

A detailed motivation for Cloud security in Smart Grids [4] and an initial design for *Cryptonite* [5] have appeared in literature. Here, we summarize the *Cryptonite* design as background context.

The basic tenets behind the *Cryptonite* design are to (1) allow the file owner to encrypt and sign the data at the client-side before storing it in the Cloud, (2) be able to audit operations performed in the Cloud, and (3) offer a scalable, user-friendly model for key management for efficient and secure data sharing.

For a client to upload a plaintext data file D on behalf of the owner U_{owner}, it first generates a random cryptographic symmetric key K^{sym}_{encr} and uses it encrypt the data file to create an encrypted file F. Then, it generates a random cryptographic public/private keypair, $< K^{pub}_{sign}, K^{pri}_{sign} >$, also called "file verification key" and "file signature key" respectively. F is signed using K^{pri}_{sign} to get a signature S_F. A security metadata header M is created with the file's unique $UUID$, the unique identifier for a *StrongBox* file, $SUUID$, K^{pub}_{sign} and fully qualified name of the owner U_{owner}. This header is itself signed using the private key of this owner, K^{pri}_{owner}. These steps ensure that the data file is encrypted on the client side, its content and security header can be verified using the corresponding signatures, and coarse grained access control is imposed by the *Cryptonite* service using the owner information stored in the security metadata.

The concept of a *StrongBox* file enables scalable key management by securing multiple files that share the same permissions using just the single global public-private keypair for each user. Intuitively, a *StrongBox* represents a unique combination of permission (such an access control list), with a specified owner, list of writers and list of readers. A shared symmetric key K^{sym}_{encr} is associated with a *StrongBox* and used for en/decryption of all files that have these identical permissions. This symmetric key is itself broadcast encrypted [6] using the public keys of all authorized readers (inclusive of writers and owner) $K^{pub}_{readers_{1..n}}$ and placed in the *StrongBox* . Broadcast encryption has the property that any one of private keys matching the list of public keys used for broadcast encryption, but no one else, can decrypt the contents. The *StrongBox* also contains the shared signature key K^{pri}_{sign} used for signing the updated contents for any write operations, with the signature key being broadcast encrypted using the public keys of all authorized writers (inclusive of owner) $K^{pub}_{writers_{1..m}}$. The *StrongBox* file is signed using the global private key of its owner, K^{pri}_{owner} and uploaded to the Cloud storage with unique $SUUID$. Thus, only authorized readers will be able to decrypt the shared symmetric K^{sym}_{encr} present within the *StrongBox* using their private key and use that shared key to decrypt the contents of the data file they are authorized for. Likewise, only authorized writers can decrypt the shared signature key K^{pri}_{sign} required to sign a valid encrypted data file that is being updated.

Cryptonite uses several standard and proven cryptographic techniques as its design primitives. It uses *symmetric cipher* for en/decryption of data files, and leverages the *Public Key Infrastructure(PKI)* with RSA public/private key pairs along with *digital signatures* and *checksums* for user identity and for checking the integrity of data/metadata and for non-repudiable auditing. It also uses a *Broadcast Encryption* technique for sharing a secret among a subset of the users which is used by the *StrongBox* . It allows *lazy revocation* using a form of *key rotation* [7] for re-encrypting updated files whose permissions have changed with minimal overhead.

III. Architecture and Implementation

The *Cryptonite* architecture is a validation of the *Cryptonite* design that was introduced in the previous section. It is implemented on top of the Microsoft Windows Azure Cloud platform[3] using standard Azure VMs, and Azure BLOB and table storage services. Here, we describe the various components of the *Cryptonite* architecture and the operations that it supports. These are illustrated in fig. 1.

A. Components

The *Cryptonite Data Repository* is the central service that runs in the Cloud and exposes interfaces for adding, updating and retrieving data files. It has two major components: the

[3]www.windowsazure.com

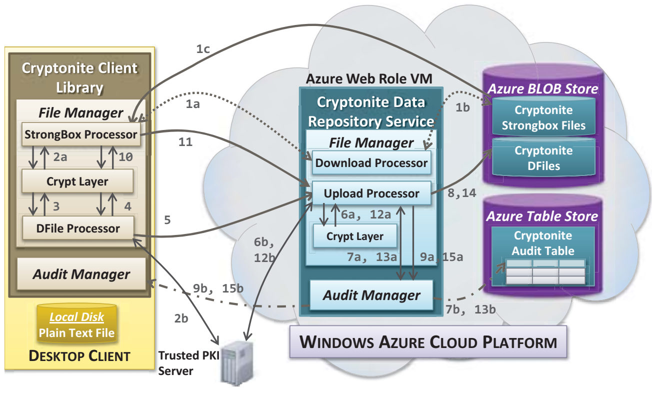


Figure 1: Architecture and interaction diagram of the *Cryptonite* data repository service and desktop client libraries.

File Manager and the Audit Manager. The *File Manager* is responsible for interacting with the Azure BLOB Storage service used for scalable and persistent storage using a single storage account. In particular, it restricts unauthorized updates to the stored data or *StrongBox* files using signature verification on requests to ensure authorised access, and also creates single-use URLs for downloading files from the BLOB storage service. The *Audit Manager* maintains a secure audit log in the Azure Table Storage for every file operation performed by the File Manager on behalf of a client. Each log entry includes a copy of the request message signed by the requesting client along with the file checksum. This is a form of provenance that can be used to prove that the current repository state were the result of valid operations.

There are two types of files that are managed in the *Cryptonite* repository: the DFile and the *StrongBox* file. The *DFile* or "Data File" consists of the original plaintext data file encrypted using an AES-256 symmetric stream cipher along with a security header as discussed in the design. The header contains the qualified name of the owner (e.g. *o=USC, ou=CENG, CN=kumbhare*), the UUID for its *StrongBox* file (e.g. 52151807-81b4-4db3-8e53-5438d73c43ac) and the file verification key pair. The encrypted contents are signed using the file signature key and the header itself is signed using the owner's private key. We use RSA-1024 bit keys for the file signature/verification key pair.

Cryptonite uses a *StrongBox* file for efficient management of access control permissions and shared keys.The *StrongBox* itself is a file that consists of a header and a body. The header contains the qualified name of the owner, the file encryption and the file signature keys – themselves encrypted using a broadcast encryption scheme, and the body contains the unique identifiers of DFiles that use this *StrongBox* along with a random seed (initialization vector, or IV) for each; the IV is used for added security during symmetric encryption and decryption of each file. Unlike the DFile, the entire *StrongBox* file is signed using the owner's private key. This allows only the owner of the *StrongBox* to update its contents, and thus add or remove DFiles associated with it.

The *Cryptonite Client Library (CCL)* is responsible for performing cryptographic operations on the client-side on behalf of the user, and interacting with the *Cryptonite* service. While the CCL is provided for convenience, users or their trusted developers can implement this library themselves using standard cryptographic algorithms and web service protocols for absolute trust when handling the plaintext files and the private keys of the users. CCL performs encryption, signing and header generation to create the DFiles from plaintext files using the private keys available for the user and shared public keys available for other users using the PKI infrastructure. It also performs decryption and signature verification of DFiles to retrieve the plaintext files. The CCL interacts with the *Cryptonite* repository and the Azure BLOB storage service using REST service interfaces. The library forms the edge of the secure *Cryptonite* operations.

Our implementation of the *Cryptonite*service and CCL is done on the .NET platform for convenience, given the native support for .NET within Windows Azure. However, our use of standard cryptographic algorithms and web service protocols for communications ensure that this service can be ported to other IaaS and PaaS Clouds, or clients in other programming languages used to access the repository. The open source *Cryptonite* package is accessible online [4]. A Java client is currently under development.

B. *File Operations*

The *Cryptonite* repository service provides a REST service interface that is similar to the one exposed by Azure's BLOB storage service. This allows existing applications that use Cloud storage to migrate to *Cryptonite*'s secure service with limited overhead. There are four key operations that we provide: POST, PUT, GET and DELETE, with the same semantics as the HTTP protocol. Here, we describe the steps for the POST operation in detail.

POST is used by the owner of a file to create and upload the secure file to the *Cryptonite* repository with an initial set of access permissions. The arrows in fig. 1 illustrates these steps. The `CreateFile()` method in the CCL takes as input the location of a plaintext file, the initial permissions for the file in the form of the owner, readers and writers, and the global private key for the owner. The client creates a normalized hashcode from the permissions by concatenating and sorting the list of authorized users and applying a unidirectional hash function (SHA-1) to get the identifier of the *StrongBox* file that reflects these permissions. The client then tries to fetch this *StrongBox* file by performing a GET operation on the *Cryptonite* service (fig. 1, step 1(a)). If present, the service returns a one-time use URL for this *StrongBox* file in the Azure BLOB storage – a facility

[4]ganges.usc.edu/wiki/Cryptonite

provided by Azure (step 1(b)), and the client uses this to download the *StrongBox* file (step 1(c)). The client then decrypts the *File Encryption key* and the *File Signature Key*, that have been broadcast encrypted into the *StrongBox* , using the private key of this user – the owner (step 2a).

If the *StrongBox* did not exist, the client generates a new one as follows. It first creates a random symmetric *File Encryption key* (using the `AESManaged` class provided by the .NET cryptography library), which is used for encrypting plaintext files, and a pair of asymmetric keys (using `RSACryptoServiceProvider`), *File Signature Key* and *File Verification Key*, for signing and verifying the integrity. The encryption key, used by readers, and the signature key, used by writers, are themselves encrypted using broadcast encryption (step 2a,2b). Given the lack of standard broadcast encryption implementation in .NET, we implemented our own version using the standard RSA algorithms [8].

The client next generates a random UUID as the BLOB name for the data file (DFile) and a random initialization vector (IV) integer. The plaintext file is then encrypted using the stream cipher(AES-256 CBC mode) using the symmetric file encryption key and the IV (step 3). The DFile is then prefixed with a security header as described earlier, and signed using the owner's private key. An audit log entry is created with the type of operation (POST), the timestamp, the UUID of the file, the file's signature and the header signature, and also signed using the owner's global private key (step 4). This DFile and the audit log are uploaded to the *Cryptonite* service using a HTTP POST operation (step 5). We introduce a special HTTP header, *x-cryptonite-audit*, to transfer the signed audit entry along with the DFile to the *Cryptonite* server. This is followed by updating the *StrongBox* with the new file's UUID and a similar audit log for it (steps 10-11).

On the service-side, when a HTTP POST operation is received, the *Cryptonite* service verifies that the file with the given resource URL (i.e. file UUID) does not exists in the BLOB storage. It then extracts the owner id from the file's security header and verifies the header signature using the owner's public key (step 6a,6b). We use a trusted third party PKI server (fig. 1) which acts as a certificate store. It returns signed X.509 public key certificates for a given fully qualified user id. Next, the service uses the file verification key present in the header to verify the signature of the encrypted file body, and the integrity of the audit entry log (step 7(a), 7(b)).

On successful verification, the entire DFile's header and body are stored in the Azure BLOB store using the provided UUID (step 8). The service creates a similar audit entry log for this operation and signs it using it's own private key, before storing both the client and service's audit logs to Azure Table store (step 9(a)). The service's audit log is returned as a receipt to the client as part of the POST response header (step 9(b)). The *StrongBox* file that has been created or updated is also received in a separate POST operation, verified, and stored in the BLOB store (steps 12-15(b)).

Since the *StrongBox* is shared by multiple files, we support optimistic consistency semantics when the identical *StrongBox* gets updated by two concurrent clients after a POST operation. We use the *ETag* header, provided by the Azure BLOB service as a version for a given BLOB file, to check for stale *StrongBox* 'es. If the *StrongBox* file's *ETag* has changed between the start and the end of POST operation, we fetch the *StrongBox* file again and redo the *StrongBox* update.

A DELETE operation is similar to the POST and can be performed only by its owner since it requires updating the *StrongBox* for removing the deleted file's entry. The request and audit verification steps are as before, except that a DFile is not present in the request body and the corresponding BLOB file is deleted from the Azure storage by the *Cryptonite* service.

A PUT operation, to update an existing file, follows a convention similar to POST , but can be performed by any authorized writer. In addition to verifying the header signature of the DFile, the *Cryptonite* service also verifies that the owner id in the updated file header has not been changed – this prevents ownership hijack by a writer. No updates to the *StrongBox* file takes place during a PUT.

The GET operation can be used by any authorized readers to download and decrypt the shared file. The user sends an HTTP GET request to the *Cryptonite* service with the UUID for the file and is returned a one-time use URL for this file in the Azure BLOB service. This URL can be used to directly download the DFile. The client then verifies the file using its signature in the header, and retrieves the *StrongBox* file for this DFile whose UUID is listed in its header. The client similarly downloads and verifies the *StrongBox* file, and uses its global private key to broadcast decrypt the symmetric file encryption key and IV present in the *StrongBox* . These can then be used to decrypt the body of the DFile and get the plaintext contents.

For all these operations, it should be reiterated that even if an unauthorized users gets access to the *StrongBox* or encrypted data files, without access to the private key of the owner, authorized readers or writers, the contents of the files cannot be read or changed. Even a hijack of the *Cryptonite* service or the Azure Cloud storage service can at worst delete the stored files – compromising persistence – but will not cause data leakage. Any attempt to change the file contents can be detected by the client using the signatures and unauthorized operations detected using the audit log receipts.

IV. Performance Optimizations

There are several cryptographic, disk and network operations that are performed in the client library and *Cryptonite*

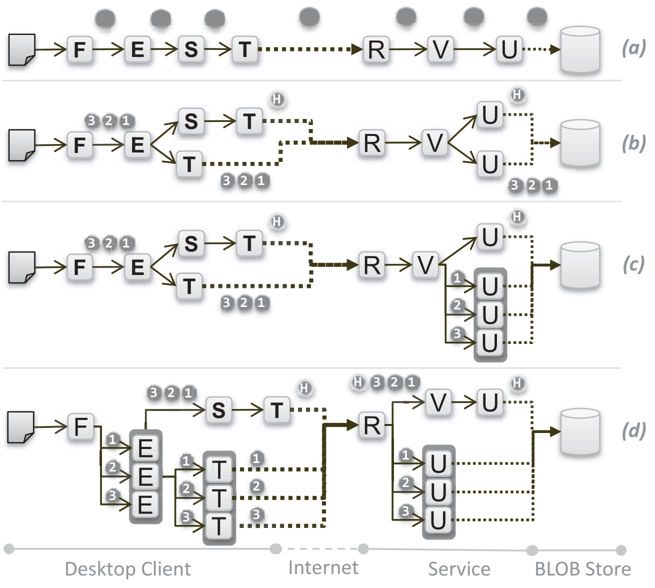


Figure 2: Performance-tuned *Cryptonite* implementations. (a) Basic, (b) Pipelined, (c) Pipelined with BLOB Parallel, and (d) Fully Pipelined and Parallel

service to provide secure storage. These can lead to significant overhead in terms of end-to-end transfer time and effective bandwidth between client and Cloud. We propose several performance optimizations to mitigate this overhead. These are discussed here using POST as the example.

Fig. 2(a) shows the primary processing stages in the basic *Cryptonite* implementation, namely, plaintext file read from disk (F), encryption using AES-256 symmetric stream cipher (E), hashing and signing of the encrypted file and header using SHA128-RSA1024 (S), followed by network transfer of the DFile header, body and audit log entry (T), on the client-side. At the service-side running in the Cloud, the DFile is received from the network (R), the signatures in the audit log and DFile header verified (V) by hashing the DFile body, following which the DFile and server-signed audit log are uploaded (U) to the Azure BLOB and table services respectively.

We leverage foundational concepts of task pipelining and data parallel execution to exploit various performance optimizations over the basic data flow in fig. 2(a). We present three increasingly sophisticated versions of this data flow: Pipelined (fig. 2(b)), Pipelined with BLOB Parallel (fig. 2(c)), and Fully Pipelined and Parallel (fig. 2(d)).

Our first optimization (fig. 2(b)) leverages an Azure BLOB service feature to upload the file as a collection of blocks, in any order, and finally commit the blocks in the desired order. This pipelines the data flow by processing one file block (e.g. of 4MB size) at a time, in order, and allows task parallel execution of individual stages of the flow concurrently. The signature task at the client and the validation task at the server, however, need to have all blocks of the file pass through them in order to calculate its hash. As a result, the DFile header cannot be computed until all blocks of the file have been accessed, and likewise for the verification (and hence the file "commit") on the service-side.

The next level of optimization (fig. 2(c)) adds data parallel execution to the block upload (U) on the service-side. The client continues to send pipelined blocks in order as before. However, since the BLOB store supports uploading the blocks in any order, we use multiple, concurrent instances of the upload task to transfer blocks from the *Cryptonite* service to the BLOB store using independent network streams. The "commit" of the BLOB ensures that the file is accessible in the correct order.

The final optimization we perform (fig. 2(d)) is in extending data parallelism to other stages in the flow, in particular, for block encryption (E) on the client side. We also decouple the validation (V) from the block upload (U) on the service-side. Both of these introduce additional complexity since the client signature (S) and service validation (V) depend on having access to the file blocks in order. We implement an in-memory thread-safe sorted queue between encryption (E) and signature (S), and between network read (R) and validation (V), that ensures decoupled yet consistent operation. We also address a security vulnerability this would expose. Encrypting each block independently with the same key and IV increases the chances of a plaintext attack. We fix this by generating and using a random byte array equal to the width of the encryption block as a prefix to the block to increase the entropy.

V. Performance Evaluation

In this section, we present a detailed empirical evaluation of the performance characteristics of the *Cryptonite* service and CCL, and the impact of different optimization strategies we have proposed. We use "effective bandwidth" as our metric, defined as the $\frac{PlaintextFileSize}{EndtoEndOperationTime} Mbits$, since this captures the tangible impact for the user. In our experimental setup, the *Cryptonite* service is deployed on a single Azure Large VM Instance (4 Cores rated at 1.6 GHz, 7 GB Memory, 400 Mbps bandwidth) and the *Cryptonite* client runs on a 4 Core, 2.5 GHz workstation with 8 GB Memory, connected to the USC campus Gigabit network. We use randomly generated plaintext test data files with sizes ranging from 100 KB to 1 GB. The results for all experiments are averaged over at least three runs; more for very small file sizes. We use the Azure .NET Storage Client API (v1.6) for Azure storage service operations.

A. Baseline Analysis

We use the baseline references to evaluate *Cryptonite* against. The first just uploads (downloads) plaintext files between the desktop client and the Azure Blob storage service with no added security. This illustrates the default performance that is achieved by a client when using the

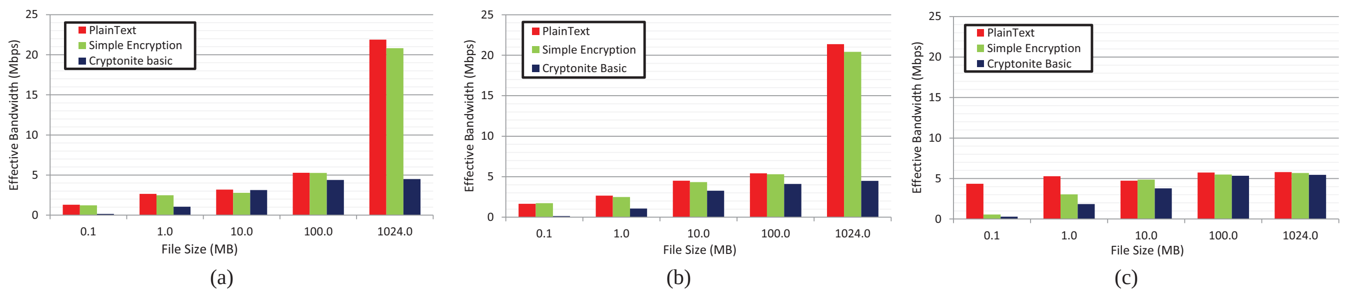


Figure 3: Comparison of Baselines and Basic *Cryptonite* Implementations for POST(a), PUT(b) and GET(c) operations.

Azure BLOB service. The second baseline adds file encryption (decryption) using a symmetric key before uploading to (after downloading from) Azure BLOB storage. This is a simple encryption approach for securing the data files from data leakage in the Cloud, and may be sufficient when only one user is involved. Both these baseline methods are compared against the basic *Cryptonite* implementation (fig. 2(a)). For *Cryptonite*, we use file access permissions with one owner, 3 readers and 3 writers; however, the actual users themselves are selected at random for each run since it impact *StrongBox* creation/selection. figs. 3a and 3b show the effective bandwidth achieved using the two baselines and the basic *Cryptonite* implementation for POST and PUT of files that range from 100 KB to 1 GB in size.

For smaller data files ($\leq$1 MB), the plaintext and simple encryption baselines offer better effective bandwidth ($\sim$2–3Mbps) than the basic *Cryptonite* implementation ($\sim$0.15–1Mbps). However, for moderate sizes (10–100 MB), *Cryptonite* approaches the baselines. This difference is due to the small, but static *Cryptonite* overhead for *StrongBox* operations such as downloading, creating/updating and uploading the *StrongBox* file. For small data files, this overhead dominates, but is a smaller fraction for medium sized files. We also see that the overhead for symmetric encryption using the second baseline is not significant compared to the plaintext baseline. When the data file size grows beyond 32 MB, the effective bandwidth of *Cryptonite* plateaus at 4Mbps while there is a sharp improvement in bandwidth for the baselines, achieving more than 20Mbps. This is due to an optimization present within the Azure .NET Storage Client API which, for files $> 32MB$, divides them into blocks of 4MB and uploads these in parallel. On the other hand, the basic CCL implementation uploads blocks sequentially to the *Cryptonite* service and also synchronously waits for it to store it into Azure BLOB storage before uploading the next block. So for larger files, the network and acknowledgement time dominates for *Cryptonite*.

The PUT operation exhibits a similar behavior to POST as expected since the key operational distinction is that POST gets and updates/creates the *StrongBox* file while PUT only retrieves an existing *StrongBox* file. The difference in time for this is negligible.

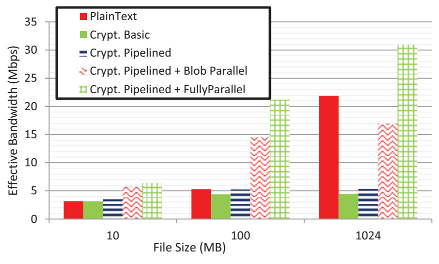
Fig. 3c shows the effective bandwidth to download a file using the baselines and basic *Cryptonite* implementation. As before, for smaller files, *Cryptonite* performs much worse than the baselines due to the static overhead to process the *StrongBox* file. Also, the Azure .NET Storage Client API does not implement any optimizations for large file downloads (unlike the upload), so there is no dramatic improvement in effective bandwidth for larger files. So the GET for the baselines are actually worse off than the PUT/POST for larger files.

B. *Analysis of Optimizations*

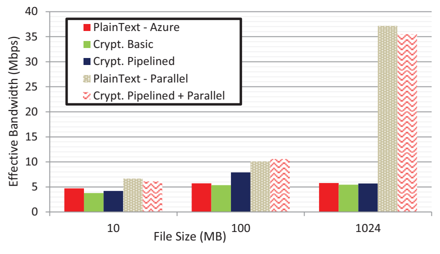
Here, we compare the baseline plaintext and basic *Cryptonite* implementations against the three optimized implementations of *Cryptonite*. Figs. 4a and 4b show the effective bandwidth achieved using these five approaches for POST and GET operations; PUT is omitted for brevity since it is similar to POST, and we also skip smaller file sizes since the benefits are negligible due to static overhead.

We can see a improvement in effective bandwidth performance for POST as we graduate from the basic implementation to the fully parallel optimization. We make several notable observations. For file sizes from 10-100MB, the pipelined optimization is better than the basic *Cryptonite* implementation and is comparable to the plaintext baseline. Introducing data parallelism takes the performance of *Cryptonite* past the baseline plaintext, making our implementation of *secure data transfer faster than an unsecured version*. Even when the parallel optimization of the Azure Client API kicks in for larger file sizes (e.g. 1 GB), we see that the fully pipelined and parallel version of *Cryptonite* offers a 40% improvement in effective transfer bandwidth over the plaintext transfer, at 31Mbps compared to 22Mbps. This remarkable improvement is achieved through fully utilizing the network bandwidth between client and the Cloud VM, made possible by the *Cryptonite* service buffering the blocks received from the client through multiple streams and asynchronously storing them on the Azure BLOB storage. Despite the asynchrony, the end to end time includes the final commit operation which ensures that entire file has been persisted before the operation completes.

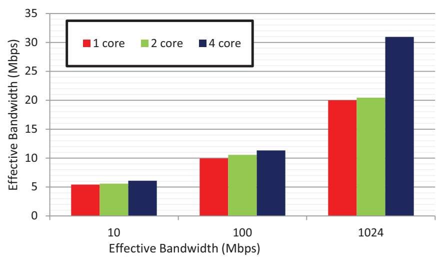
For the download scenario (fig. 4b) our optimizations offer better performance than the basic *Cryptonite* implementation. We also see dramatic improvement in effective



(a) POST



(b) GET



(c) POST with different client cores

Figure 4: Optimized *Cryptonite* Implementations.

bandwidth compared to the plaintext baseline. The main reason for this is that the Azure API does not perform parallel download operations while *Cryptonite* client API does. However we implemented a parallel download operation for plaintext data for fair comparison with the the most optimized *Cryptonite* implementation and observed that the latter performs as good as the former with only about 4% overhead, at 35.5Mbps compared to 37Mbps for a 1GB file. This can be explained by the fact that the CCL and the Azure API both perform a download directly from the Azure BLOB storage.

C. Impact of Client Capability

Since many of the cryptographic operations occur on the desktop client, we studied the effect of using different sized client machines on the performance. We run our most optimized *Cryptonite* implementation on two client VMs running within Windows Server Hyper-V, sized with 1 core/2GB RAM and 2 core/4GB VM. These are compared against the client running on the default 4 core/8GB physical machine. Fig. 4c shows the results, and we see that there is negligible performance variation between the different clients for files up to 100MB. For larger files of 1 GB, the additional CPU cores and memory available of the physical machine allows the .NET Task parallel library to create more concurrent threads; consequently, the compute becomes the bottleneck for smaller client machines.

D. Client and Server Scalability

Lastly, we study the scalability of the *Cryptonite* service as the number of concurrent clients accessing the service increases. We run the POST experiments for different file sizes on five physical machines each running between 1 to 10 clients. This allows us to compare the performance of the *Cryptonite* service for 1 to 50 concurrent users. Fig. 5 shows the *average end to end time* taken for a POST operation on a 100 MB file. It is clear that the *Cryptonite* service degrades gradually with the number of concurrent clients performing POST, taking 3 times longer with 10 concurrent clients as compared to 1. We make two further observations. As the number of clients on a single machine increases, the total POST time increases linearly. Looking at the Windows performance counters shows that this is a bottleneck on the client caused by reaching the .NET task parallelism threshold on a single machine. As we increase the total number of parallel clients, we also see the time taken for the final commit increase – after all the file blocks have been transfered asynchronously to the service and the client waits for the blocks to be flushed to Azure BLOB store by the service. This indicates that the service is the bottleneck, and we confirm that the service hits the bandwidth ceiling of 400Mbps imposed on a large VM. This motivates the need for running multiple *Cryptonite* services on-demand to meet client needs, and we plan to address this in future.

It should be noted that the scalability of the GET operation is not impacted since CCL directly downloads the file from Azure BLOB store after the *Cryptonite* service creates a one time URL for it.

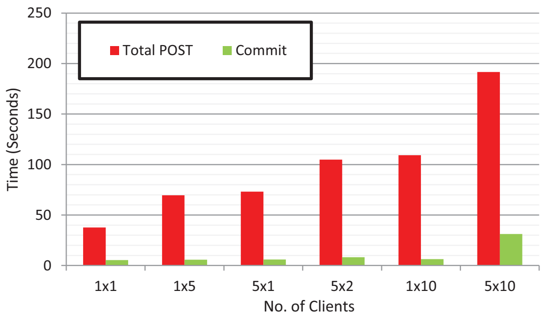


Figure 5: Client and Server Scalability.

VI. Related Work

Cloud security and privacy research have focused on identifying threats arising from using Cloud infrastructure, such as risks due to multi-tenancy, attack surfaces exposed by interactions between the user and the Cloud provider, and Cloud integrity issues due to malware injection in VMs [9]. Researchers in eHealthcare and Smart Grid domains have analyzed Cloud security requirements and raised security concerns in moving their applications and sensitive data to Clouds [4], [10]. Zhang, et. al. [10] consider the secure storage and sharing of a patient's electronic health records (EHRs) with a "patient-centric" view – a problem closely related to the scenario addressed by *Cryptonite*.

Building a trusted Cloud environment poses a number of security issues in computing, storage, and communication infrastructure. CertiCloud [11] addresses issues of detecting malware attacks on VMs, using a Trusted Platform Module (TPM) for certifying remote resources. Li et. al.[12] propose a system for securing Cloud VM execution without trusting the underlying Cloud management OS or the hypervisor by securing the CPU and memory runtime, the network interface, and local secondary storage. Such systems compliment *Cryptonite* when clients need to run within Cloud VMs rather than a secure desktop.

The issues of data security for outsourced data have been studied for distributed systems. The main distinction arises from the level of trust on the storage service provider for access not only to the plaintext data but also on the correctness of the operations performed on the data. NAS/DS and SNAD[13] were among the first systems addressing issues of untrusted remote data storage but assumed a trusted server that protected data from remote and local intruders. SiRiUS[8] and PLUTUS[7] systems extended the concept and did not rely on the storage provider for authentication. Stanton's[14] and Cachin's[15] surveys provide a comparative study of various security requirements and different approaches for a secured data storage in traditional distributed systems and in Clouds.

Recent systems such as Cloud-Proof[16] and CCS[17] built specifically for Cloud environments are closely related to *Cryptonite*. However, they have been designed at a lower system abstraction and require changes to the Cloud platform. We leverage some of their security techniques but cover a broader set of requirements that we have motivated, provide compatible service interfaces for data access, and are transparent with respect to the underlying Cloud platform.

VII. Conclusion

We have described the design and implementation of *Cryptonite*, a secure, performant data repository for Cloud platforms that offers sustainable key management features. Our empirical analysis shows that the optimizations that we propose do not have significant overhead and even perform better than the Azure .NET APIs for plaintext operations. It provides service compatibility with Azure's BLOB store and allows easy migration for Cloud data storage clients to incorporate security.

Acknowledgment

This material is based upon work supported by the United States Department of Energy under award no. DE-OE0000192 and the National Science Foundation under award no. CCF-1048311. The views and opinions of authors expressed herein do not necessarily reflect those of the US Government or any agency thereof. We also thank Microsoft Corp. for providing Azure resources under the NSF CiC grant.

References

[1] Cloud Security Alliance, "Security Guidance for Critical Areas of Focus in Cloud Computing V2.1," Tech. Rep., 2009.

[2] W. Itani, A. Kayssi, and A. Chehab, "Privacy as a service: Privacy-aware data storage and processing in cloud computing architectures," *DASC*, 2009.

[3] Y. Simmhan, M. Giakkoupis, B. Cao, and V. K. Prasanna, "On using cloud platforms in a software architecture for smart energy grids," in *CloudCom*, 2010.

[4] Y. Simmhan, A. G. Kumbhare, B. Cao, and V. Prasanna, "An analysis of security and privacy issues in smart grid software architectures on clouds," in *IEEE CLOUD*, 2011.

[5] A. G. Kumbhare, Y. Simmhan, and V. K. Prasanna, "Designing a secure storage repository for sharing scientific datasets using public clouds," in *Workshop on Data Intensive Computing in the Clouds*, 2011.

[6] A. Fiat and M. Naor, "Broadcast encryption," in *Advances in Cryptology*, ser. LNCS, D. Stinson, Ed. Springer Berlin / Heidelberg, 1994, vol. 773, pp. 480–491.

[7] M. Kallahalla, E. Riedel, R. Swaminathan, Q. Wang, and K. Fu, "Plutus: Scalable secure file sharing on untrusted storage," in *USENIX FAST*, 2003.

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

[9] M. Jensen, J. Schwenk, N. Gruschka, and L. Iacono, "On technical security issues in cloud computing," *Cloud '09*.

[10] R. Zhang and L. Liu, "Security models and requirements for healthcare application clouds," in *IEEE CLOUD*, july 2010, pp. 268 –275.

[11] B. Bertholon, S. Varrette, and P. Bouvry, "Certicloud: A novel tpm-based approach to ensure cloud iaas security," in *IEEE CLOUD*, july 2011, pp. 121 –130.

[12] C. Li, A. Raghunathan, and N. Jha, "Secure virtual machine execution under an untrusted management os," in *IEEE CLOUD*, july 2010, pp. 172 –179.

[13] E. L. Miller, D. D. E. Long, W. E. Freeman, and B. C. Reed, "Strong security for distributed file systems," in *IPCCC*, 2001.

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

[15] C. Cachin, I. Keidar, and A. Shraer, "Trusting the cloud," *SIGACT News*, 2009.

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

[17] S. Kamara and K. Lauter, "Cryptographic cloud storage," in *Financial Cryptograpy and Data Security Conference*, 2010.