SURGE: The Secure Cloud Storage and Collaboration Framework

By Adin R. Schmahmann

S.B., E.E.C.S. & Physics M.I.T., 2013

Submitted to the Department of Electrical Engineering and Computer Science

In Partial Fulfillment of the Requirements for the Degree of

Master of Engineering in Electrical Engineering and Computer Science

At the Massachusetts Institute of Technology

May 2014
[June 2014]

Signature redacted

Author: ______________________

Department of Electrical Engineering and Computer Science

May 27, 2014

Signature redacted

Certified by: ______________________

Nickolai Zeldovich, Associate Professor of Electrical Engineering and Computer Science

Thesis Supervisor

May 27, 2014

Signature redacted

Accepted by: ______________________

Prof. Albert R. Meyer, Chairman, Masters of Engineering Thesis Committee

SURGE

THE SECURE CLOUD STORAGE AND COLLABORATION FRAMEWORK

BY: ADIN R. SCHMAHMANN

SUBMITTED TO THE DEPARTMENT OF ELECTRICAL ENGINEERING AND COMPUTER SCIENCE

ON MAY 27, 2014 IN PARTIAL FULFILLMENT OF THE

REQUIREMENTS FOR THE DEGREE OF MASTER OF ENGINEERING IN

ELECTRICAL ENGINEERING AND COMPUTER SCIENCE

ABSTRACT

SURGE is a Secure Cloud Storage and Collaboration Framework that is designed to be easy for application developers to use. The motivation is to allow application developers to mimic existing cloud based applications, but make them cryptographically secure, in addition to allowing application developers to come up with entirely new secure cloud based applications. SURGE stores all of its data as operations and as a result can leverage techniques like Operational Transforms to allow offline usage as well as lowering network bandwidth consumption. Additionally, storing data as operations allows SURGE to develop a rich permissions system. This permission system allows basic permissions such as read-only, read-write, and administrator in addition to more advanced permissions such as write-only, group read/write, and anonymous permissions. To evaluate the usability of SURGE a C# prototype was constructed and used to create a collaborative text editor that performs well under real-world user tests.

Thesis Supervisor: Nickolai Zeldovich

Title: Associate Professor of Electrical Engineering and Computer Science

SECTION 1 INTRODUCTION

Storing and collaborating on data in "the cloud" offers many advantages to users over doing work on client devices, or operating over a peer to peer network, and allows the possibility for new and unique applications. While the last decade has seen many new applications make use of the cloud paradigm, such as Dropbox and Google Docs, it has come at the expense of requiring the user to put more trust in the application developer and cloud maintainer. For instance collaborating on a text document using Google Docs requires the user to trust that sensitive data is not being read by Google and trust that Google did not corrupt or modify the text document. However, many users would like to be able to take the same cryptographic primitives that protect data privacy, integrity, and user anonymity for standard applications and apply them to the cloud. As the cloud is a fairly simple, yet broad, paradigm approximating a single server with vast resources, a simple and broad cryptographic framework for the cloud that could be used effectively in many different situations would be extremely useful.

The current state of the art in secure collaboration via "the cloud" is SPORC (Feldman, Zeller, Freedman, & Felten, 2010). SPORC works by securely storing a log of the operations (i.e. changes) to the object being collaborated on. The security aspects of SPORC's secure logging are guaranteed using client-side encryption to protect operation confidentiality and having the clients perform signing and verification on the operations to guarantee integrity. SPORC guarantees that modifications to the data are performed only by approved users via an object Access Control List (ACL). This ACL is maintained securely in a manner similar to the object data and is consulted by user devices before applying an operation coming from "the cloud".

Unfortunately, a major shortcoming of SPORC is that it does not provide enough functionality for many real use cases. There are four major use cases detailed below that could be very useful to application developers looking to create secure cloud-based collaboration applications. Briefly these use cases are non-administrators updating their private keys, group access, granting anonymous access, and an internal messaging system.

In real systems cryptographic private keys tend to get compromised. Whether a user loses his/her smart card, a user's laptop is stolen or the user's machine is hacked into, systems must be able to deal with this reality of key compromise. Additionally, once a key is compromised it should be replaced as soon as possible in order to minimize the usability of the compromised key. As a result replacing a key should not require the user waiting for an administrator of the object to replace the key for him/her; the user should be able to replace the key by him/herself.

Additionally, in many real systems users tend to collaborate with the same group of people on multiple documents. As a result, the ability for a user to be able to share an object with the people in his/her team at work is important. Not to mention that if one of the people on the team leaves it would be great if some administrator in charge of the team could revoke all of that user's access instead of relying on the administrators of every shared object to revoke the user leaving the team (which would be an administrative nightmare). Similarly, adding a new member to the team should be a quick and painless procedure for the team administrator and not require tracking down the administrators for every object intended to be shared with the group.

In large organizations anonymity may be a real security concern. For instance if a department head is working with auditors about the productivity of his/her department he/she may not want the employees of the department to know who they are being audited by and possibly not even know that they are being audited at all. Therefore, if the department head/administrator could anonymously grant the auditor/s access to files this would help remedy this concern.

The method through which users of a cloud-based sharing system communicate tends to be an overlooked, but important part of the system as a whole. Whether it's a secure cloud storage system like Tresorium (Lam, Szebeni, & Buttyan, 2012), a non-cryptographically secure cloud storage system like Dropbox, or a secure collaboration system like SPORC, somehow users need to find out that data has been shared with them. In systems like Dropbox and Tresorit (the commercial implementation of Tresorium) the method of communication is via email, and in SPORC this method is left completely undefined. However, secure email has its own set of problems and it would be great if the system that was being designed for secure collaboration could also deal with secure communication of, at the very least, information related to the system. This secure communication could be used by anything from a tool for resolving internal system messages, to a kind of internal email system, even to a chat system, but having none of these be part of a secure collaboration system is appalling.

The contribution of this thesis is the design and implementation of the SURGE framework and library. SURGE is a secure collaboration and storage framework that can run on untrusted resources such as a commercial "cloud". Additionally, SURGE is designed to resolve the shortcomings of SPORC mentioned above and to do so in a way that is both easily usable by application developers and also easily extensible by security-minded developers.

The key idea that SURGE makes use of to implement the extra functionality it has over SPORC is that verifying operations can be more than just checking cryptographic signatures. In particular, an application developer can define an operation O and a corresponding permission for a user that is "User can perform operation O". Then when an operation is being verified the verifier checks both that the operation has not been tampered with (via cryptographic signatures) and that the user can perform the particular operation (via Boolean check). In this way the application developer can tailor the types of permissions grantable to users to fit the application model and not simply use a single type of permission for all operations. Additionally, despite the fact that this change affects only modifications/operations it is sufficient to cover all of the cases mentioned above. This is true because SURGE uses operations not just to modify the data being stored, but also to modify the ACL controlling access to the data. As a result, many kinds of new ACL permissions and operations can be created to help solve these and similar problems.

To evaluate SURGE, a prototype framework and library was built in C#, and a collaborative text editor application was built leveraging SURGE. The SURGE framework and library prototype is roughly 1000 LoC, and the application code particular to SURGE is a measly 50 LoC. The collaborative text editor performs efficiently enough that a few qualitative user tests showed no problems with application performance.

This rest of the paper is structured as follows: Section 2 will explain some of the background necessary to understanding how SURGE works, notably Operational Transformation. Section 3 will move on to explain the design of the SURGE framework. After going through the SURGE framework Section 4 will detail the SURGE API and give some examples and detail about how important SURGE concepts are used/implemented. Section 5 will delve into SURGE's Operations and Permissions Library, which contains the insights necessary to leverage the SURGE framework to achieve some of the important features (such as anonymous and group-based permissions) that are unavailable in current systems like SPORC. In Section 6 SURGE is evaluated and put into use in creating a secure collaborative text-editor application. Section 7 describes some of the work related to SURGE, before concluding in Section 8.

SECTION 2 BACKGROUND

While it is possible for a developer to use SURGE without any need for Operational Transformation (OT), for instance by requiring clients to check out objects from some server, for those truly interested in cloud-based collaboration or allowing asynchronous usage understanding OT is a must.

SECTION 2.1 OPERATIONAL TRANSFORMATION

Operational Transform (OT) is a technology designed to allow many users to work on some shared object. OT assumes that instead of storing this shared object in its entirety the object is stored as a sequence of the operations that comprise it. Given that we are storing objects as a sequence of operations OT allows users to a) work offline/disconnected and b) minimize bandwidth usage by sending only small operations to/from the clients as opposed to the entire object. OT allows users to locally commit changes and then properly synchronize state with the other users at a later time. For instance say we have a data object that is a list of characters that start off with the letters ['A', 'B', 'C'] and Alice and Bob download the object before heading out on vacation. Alice decides to remove the 'B' from the list by performing the operation *remove(2)*. Similarly Bob decides to remove 'C' from the list by performing the operation *remove(3)*. When Alice and Bob get back from vacation they send each other their operations. Realistically we would like both Alice and Bob to get ['A'] as their result after they apply each other's operations. However, while in this case Bob can naively apply Alice's *remove(2)* operation to his copy of the list (['A', 'B']) were Alice to naively apply Bob's *remove(3)* operation most programming languages would crash or throw an exception since Alice does not even have 3 elements in her copy of the list. Instead, Alice needs to realize that since she has already deleted one element from the list that she really needs to transform Bob's operation to be *remove(3-1)=remove(2)* which will remove the 'C' from Alice's ['A','C'] list and then both Alice and Bob will end up with the list ['A'].

However, while this particular example is very nice and clear cut many other examples are not, especially when the application is unaware of the user's intended semantics. For example, say Alice and Bob are sharing a String "Stop" and as before Alice and Bob go on vacation. Alice decides to perform the operation insert(5,"!") while Bob performs the operation insert(5, " Now!!!!"). When Alice and Bob send their operations to each other it is unclear whether the combined String should be "Stop! Now!!!!" or "Stop Now!!!!!". Instead really all that matters is that Alice and Bob agree on the same result and hopefully it's not too much work for them to fix the result to be what they intend. Therefore, in order to deal with these types of situations we need an arbiter to give some priority to Alice's or Bob's operation, so that in the event of a "tie" Alice and Bob know how to correctly resolve their operations. For instance if the arbiter decides that Alice's operation should get precedence then both Alice and Bob will end up with the String "Stop! Now!!!!".

SECTION 3 SYSTEM DESIGN

The major design choice was to make all changes to object data or metadata an operation as opposed to simply storing the entire updated object, as described in the OT section above. This choice was made in order to be able to allow users to grant other users permission to perform arbitrary application specific operations to both an object's data and its metadata. This section is outlined as follows: It will start by describing the fundamentals of Read and Write Permissions, give an overview of SURGE, explain the various possibilities for OT arbiters, and finally end with an overview of how an application developer might use SURGE.

SECTION 3.1 SURGE SYSTEM OVERVIEW

Fundamentally the SURGE framework is designed to make it easy for application developers to make secure cloud-based collaboration applications. To ensure the data stored via SURGE is secure, SURGE makes use of standard cryptographic primitives. Additionally, as described above SURGE makes use of OT to allow asynchronous collaboration. To package the cryptography and OT together SURGE uses the concept of a Box. A SURGE Box is the container and manager for a data object stored in the cloud as well as its associated ACL. These Boxes manage everything from sending/retrieving operations to/from the cloud to performing all of the cryptographic processes required to ensure that the operations are sent/received securely. Finally these Boxes compile the operations into the current state of the data so that application developers can easily gain access to the state of the data stored in the Box without having to deal with OT, cryptography, or any of the mechanisms internal to the SURGE framework.

The basic usage of a Box, as illustrated in Figure 1, is to be the intermediary between the application using SURGE and the location of the data stored via SURGE. Therefore, since Boxes are such an important way for application developers interface with SURGE, they should be further explained before going into the rest of the SURGE framework. In particular, the entire processes illustrated in Figures 2 and 3 take place inside of SURGE Boxes. The process in Figure 2 takes incoming secured operations (called OpCapsules) from the cloud and:

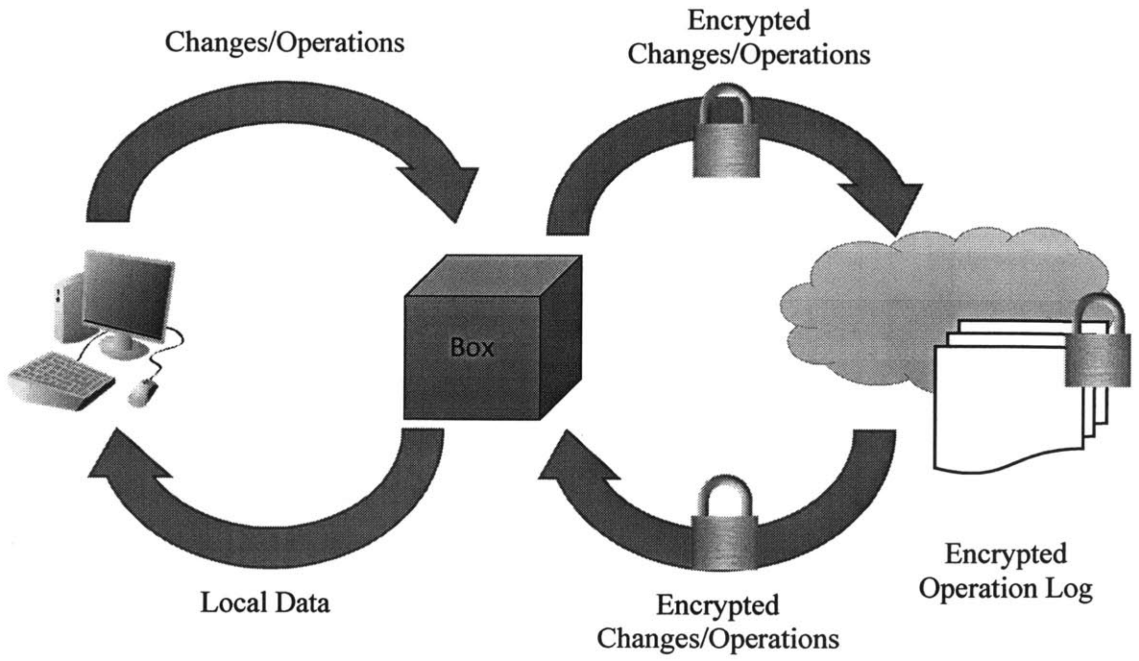

FIGURE 1 SURGE BOXES: THE INTERMEDIARIES BETWEEN CLIENTS AND THE CLOUD

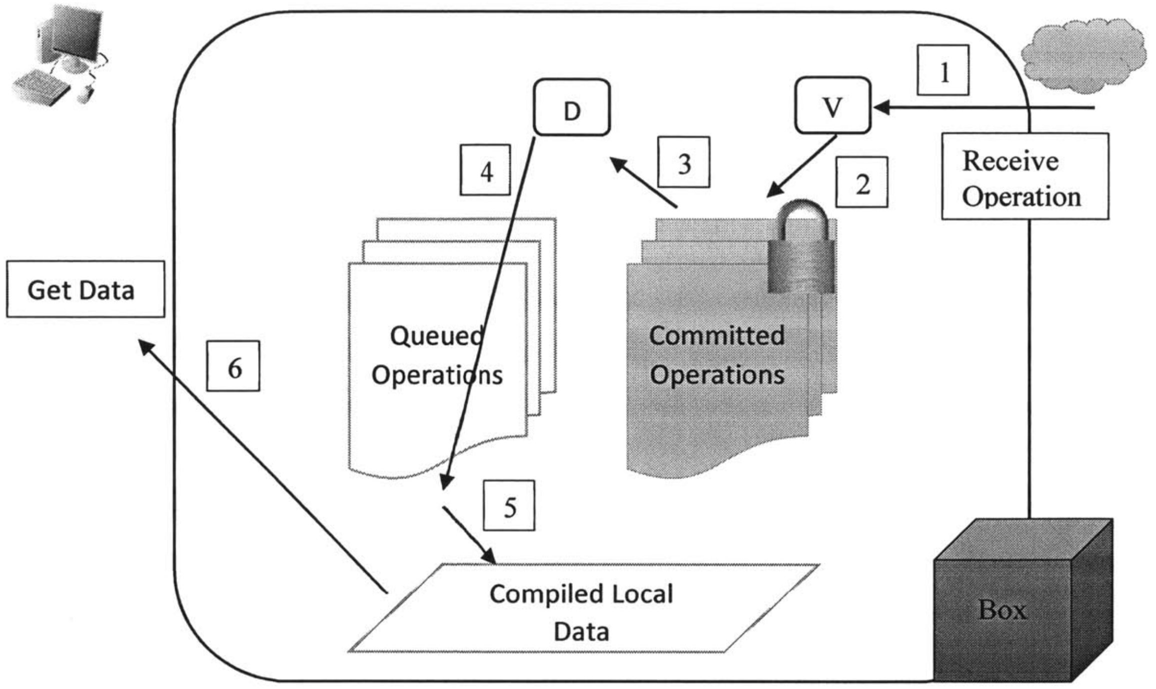

FIGURE 2 THE PROCESS AN OPERATION GOES THROUGH FROM WHEN THE CLIENT'S BOX RECEIVES AN OPERATION UNTIL IT IS READ BY THE USER. NOTE THAT D=DECRYPT, AND V=VERIFY.

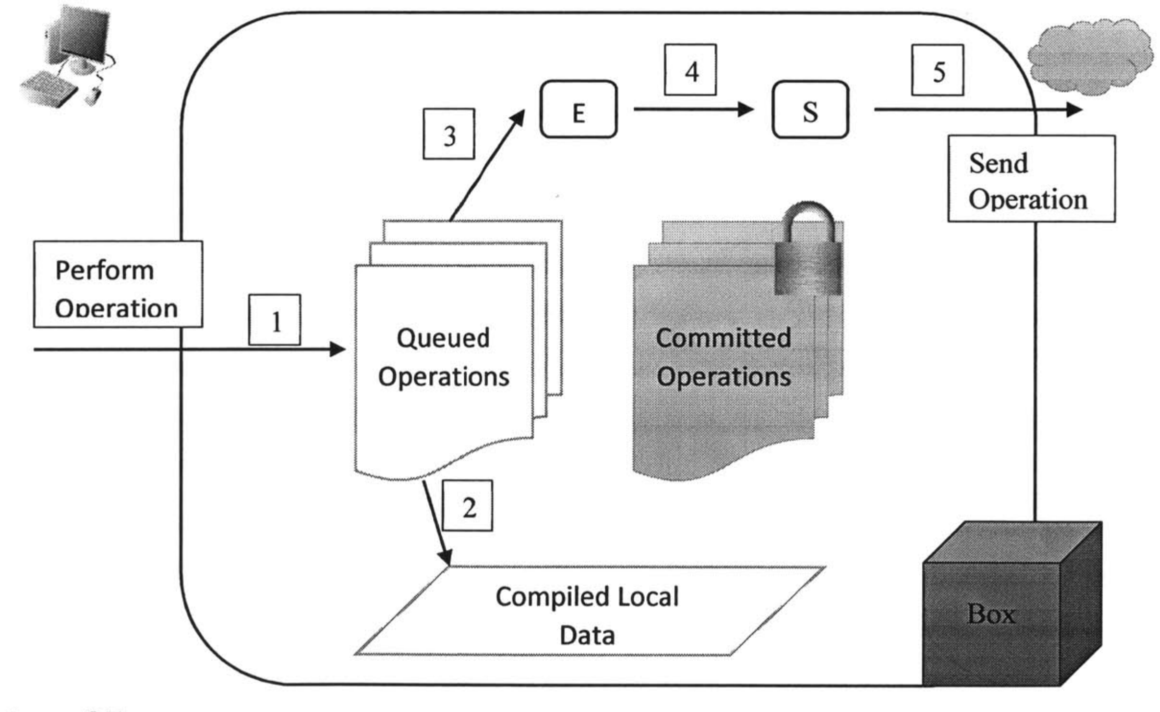

FIGURE 3 THE PROCESS AN OPERATION GOES THROUGH FROM WHEN THE CLIENT PERFORMS IT UNTIL IT REACHES THE CLOUD. NOTE THAT E=ENCRYPT, AND S=SIGN.

a) Verifies the operations are allowed to be performed by the signer of the operation

b) Appends the OpCapsule to the local list of approved committed operations

c)

    a. If the operation is encrypted then the Box will attempt to decrypt it

    b. If the operation can be retrieved (i.e. it was not encrypted, or the user making use of the Box had the decryption keys required to read the operation) it is transformed using OT across all of the operations in the Box's queue of operations that have not been committed to the cloud yet. This transformed operation is then applied to the current local state of the Box

Similarly, when (as in Figure 3) an application makes a modification to the data or ACL stored in a Box it goes through the following process:

a) The operation is added to the Box's queue of operations waiting to be sent

b) If applicable the operation is applied to the current local state of the Box. (An example where this would not be applicable was if the user making use of the Box only had permission to add content to the Box. In this case the user cannot even read the current state of the data in the Box)

c) When the operation is ready to be sent off to the cloud it is put into an appropriate OpCapsule, meaning that, if necessary, the operation is encrypted and the result is signed.

There are only two types of OpCapsules that SURGE deals with PublicOpCapsules and PrivateOpCapsules. As expected PublicOpCapsules are used to guarantee that an operation has been committed by a user permitted by the corresponding Box's ACL, but does not encrypt the contents of the operation. On the other hand PrivateOpCapsules also encrypt the contents of their encapsulated operations. While all SURGE Boxes have their ACL operations stored in PublicOpCapsules, SURGE makes use of three types of Boxes. The first is PrivateBoxes that encrypt their data, the second is PublicBoxes that do not encrypt their data, and finally SURGE even makes use of PrivateAndPublicBoxes that have two compartments, one for encrypted data and one for non-encrypted data.

SECTION 3.2 READ AND WRITE PERMISSIONS

Before continuing on with the details of how SURGE is implemented, it is important to understand Read and Write Permissions and their basic cryptographic underpinnings. To start, determining access to an object inherently involves two types of permissions, those regarding reading the content and those regarding modifying the content. These permissions are inherently tied to the cryptographic notions of confidentiality (reading the content) and integrity (modifying the content). Just as these two types of permissions are distinct, modern cryptography uses separate mechanisms to enforce them. Encryption/Decryption is used for reading, while Signing/Verification is used for modification. Read Permissions, enforced via encryption and decryption, rely solely on cryptographic mechanisms on behalf of the content owner for security, since as long as the client encrypting the data is working properly there is nothing any other client can do to violate the confidentiality implied by the encryption. On the other hand Write Permissions, enforced via signing and verification, rely on both the signer and verifier to be working correctly. For instance, an incorrect verifier could end up accepting invalid modifications and an incorrect signer could also end up allowing malicious users to make invalid modifications that a correct verifier would accept. While it is in some ways unfortunate the Write Permissions inherently require more trust than Read Permissions they also allow additional flexibility. For instance, since Write Permissions in the end are Boolean checks it is easy to define an arbitrary list of possible actions and as long as there is an ACL listing which users are allowed to perform which actions, all of these arbitrary actions are doable. On the other hand having an arbitrary list of possible Read Permissions is much harder since the developer must work with the cryptography by either creating a new cryptosystem for the desired set of permissions or having many different encryptions of the same data under different permissions. As a result SURGE allows granting arbitrary Write

Permissions on an object or its ACL, but only allows a standard Read Permission for entire SURGE objects.

SECTION 3.3 VERIFYING OPERATIONS

Assuming an unchanging ACL, when a user performs an operation on an object he applies it locally immediately and then puts it in a queue to be sent to an untrusted managing server later (this may be a long time if the user is disconnected/in offline mode). The operation sent to the server contains a signed OpCapsule. For the purpose of illustration say the operation being committed is in a PrivateOpCapsule, as described in Figure 4.

The PrivateOpCapsule contains:

a) The operation information encrypted with an asymmetric public key (e.g. an RSA key) belonging to the object

b) The identity of the client and the number operation this is for that client (i.e. it is Client ABC's 5th operation on this object)

c) The global number corresponding to the latest operation on this object received from the server (as discussed in the OT section above OT needs some sort of arbiter and SURGE can

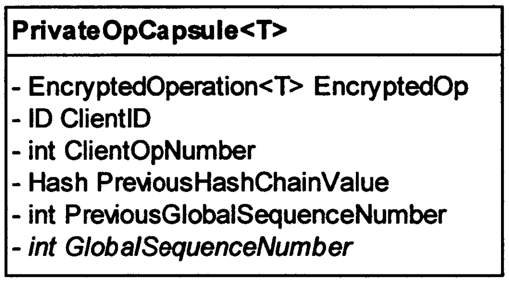
PrivateOpCapsule<T>
- EncryptedOperation<T> EncryptedOp
- ID ClientID
- int ClientOpNumber
- Hash PreviousHashChainValue
- int PreviousGlobalSequenceNumber
- *int GlobalSequenceNumber*

FIGURE 4 A CAPSULE CONTAINING THE OPERATION AND INFORMATION REQUIRED FOR SECURITY CONCERNS. NOTE THAT WHEN THE PRIVATEOPCAPSULE IS SIGNED THE SIGNATURE DOES NOT COVER THE GLOBALSEQUENCENUMBER WHICH COMES FROM THE UNTRUSTED SERVER.

have an untrusted server be an arbiter that simply labels the incoming operations with increasing numbers[1])

d) The global number of this operation. This number will be set by the server once the operation is received, and as a result is not included in the signature of the PrivateOpCapsule.

e) The hash chain value of the latest operation seen from the server (i.e. the hash chain of all of the operations from the server up until the latest)

After the operation has been committed to the server when clients request the operation they check that it is valid (by checking if it is really the n+1st operation committed by the client specified, really is in the correct place in the hash chain, has a valid signature, the signature belongs to someone specified in the ACL as being allowed to perform the operation, etc.). If it is valid they transform the operation across all of the operations committed locally but yet to be sent to the server, they then perform the transformed operation.

SECTION 3.4 MODIFYING THE ACL

The above discusses SURGE with an unchanging ACL, but changing ACLs is a very important part of the system. An ACL, as used in SURGE, has the properties described in Figure 6.

[1] Note that the untrusted server cannot really do anything malicious with this labeling ability without the clients detecting the malicious activity. This scheme is explained in detail in the SPORC paper.

```
BoxKeys
PublicReadPermission PublicBoxReadKey
EncryptedContainer<List<ReadPermission>> EncPreviousKeys
```

FIGURE 5 SURGE BOX KEYS

```
ACL
- List<PublicWritePermission> Administrators
- List<PublicWritePermission> Writers
- List<WrappedPermissions> Readers
- BoxKeys BoxKeyInfo
```

FIGURE 6 A SURGE ACL

To break the ACL down into more understandable terms we will look at it field by field.

a) The Administrators field is the list controlling which users can modify the ACL, and how they can do so. More concretely the "Administrators" list can be understood as a list of public keys and their corresponding permissions. For instance one of the items in the Administrators list might be Alice's public key and a corresponding indicator that Alice has permission to make arbitrary modifications to the "Writers" field of the ACL.

b) The Writers field is the list controlling which users can modify the object data and how they can do so. Like the "Administrators" list the "Writers" list can be understood as a list of public keys and their corresponding permissions. For instance, one of the items in the Writers list might be Bob's public key and a corresponding indicator that Bob can perform arbitrary operations on the object data.

c) The BoxKeyInfo field has the structure described in Figure 5, and contains 1) the current public encryption key for the box and 2) all of the previous private keys for the box encrypted under the current box's public encryption key

d) The Readers field is effectively SURGE's key management system. It stores the current box private decryption key encrypted under the public keys of all of the users who are given

"Read Permission" to the box. In this way the "Readers" field simply contains the keys that allow the appropriate users to read the box contents.

Modifying the ACL is as simple as performing an operation on it, much like the data operations described above. The only difference is that instead of using a PrivateOpCapsule with an encrypted operation, instead ACL modification operations use PublicOpCapsules in which the operations are not encrypted. As a result, adding new users to the ACL is simple, to grant read access simply encrypt the object's asymmetric private key with the new user's public key and the administrator can then add that key to the object's Readers field in the ACL. To grant the other permissions write/administrator the administrator simply needs to perform a modification to the object's ACL indicating that the new user has the intended privileges. All of these operations are public and it is easy to verify whether an ACL operation is valid since it uses the same mechanism as for regular operations (signatures and checking the object's current ACL).

On the other hand removing users from the ACL is a little trickier, and is outlined in Figure 7. Two notations that will be used from here on out are the notation *A{B}* which means that the key B is

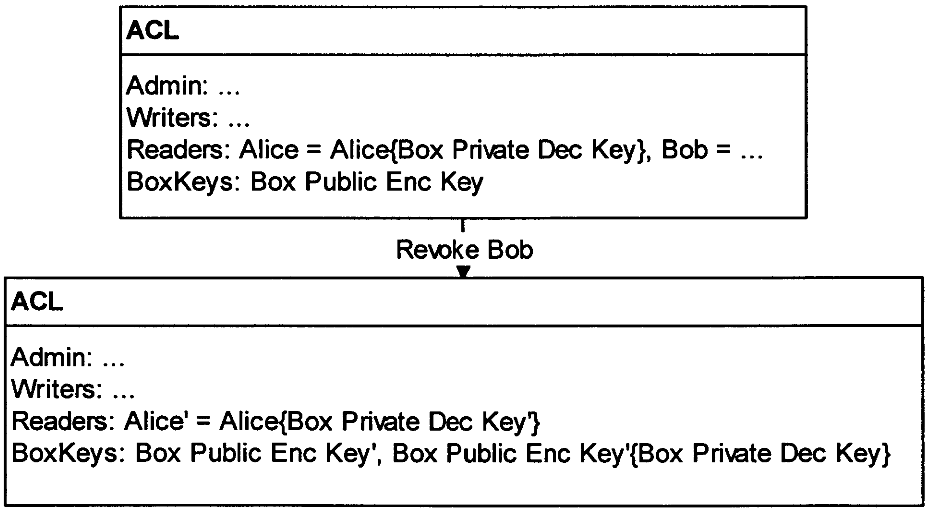
ACL
Admin: ...
Writers: ...
Readers: Alice = Alice{Box Private Dec Key}, Bob = ...
BoxKeys: Box Public Enc Key
Revoke Bob
ACL
Admin: ...
Writers: ...
Readers: Alice' = Alice{Box Private Dec Key'}
BoxKeys: Box Public Enc Key', Box Public Enc Key'{Box Private Dec Key}

FIGURE 7 THE ACL OF AN OBJECT BEFORE AND AFTER BOB'S READ ACCESS TO THE OBJECT IS REVOKED. THE NOTATION A{B} MEANS THAT THE KEY B IS ENCRYPTED UNDER THE KEY A. ADDITIONALLY, THE NOTATION X' MEANS THAT X HAS CHANGED. FINALLY THE NOTATION ALICE' = ALICE{BOX PRIVATE DEC KEY'} MEANS THAT ALICE'S NEW READ PERMISSION IS DEFINED BY HER PUBLIC KEY WRAPPING THE NEW BOX PRIVATE KEY.

encrypted under the key A, and the notation *A'* which means that a user A has changed keys to A'. In particular revoking read access means that the object's asymmetric key must be updated to a new one, but the old operations must still be readable. As a result, SURGE has the administrator replace all of the old asymmetric keys encrypted under users' public keys with the new asymmetric key encrypted under their public keys. Additionally, the old key, and all previous keys, are encrypted under the new key to allow new users to have easy access to decrypting older operations. A final important detail about revoking access to the ACL regards concurrency. If two administrators, Alice and Bob, simultaneously tried to revoke access to two different users (say Andrew and Brenda respectively) then, while this might not inherently seem problematic it turns out that since each of these operations involves encrypting the old keys under a new one, there isn't really a correct one to use. In particular, using either of these two keys results in either Andrew or Brenda still having access to the data going forward. In order to prevent this SURGE uses SPORC's concept of a "Barrier Operation" that basically reintroduces synchronization to the system by requiring that any operations that come after the Barrier Operation *b* be reexamined and recommitted after taking into account (via some procedure like OT) operation *b*.

SECTION 3.5 ROOT OF TRUST/VERIFYING THE FIRST OPERATION

One issue that we have thus far glossed over is the "root of trust" of an object's ACL. In particular, incoming SURGE operations can be verified only given that the previous state of the object (data and ACL) are also verified. Then how is the first operation creating the object verified? SURGE directly exposes this issue by requiring that the public key of the first operation on the object be passed into the function that retrieves SURGE objects from the "cloud". While in many frameworks keeping track of the "root of trust" for every shared object might be problematic, SURGE's internal messaging system (discussed in Section 5.4) ensures that a user Alice can have an "inbox" and that when other users share objects with her they send her a notification that includes the "root of trust" for the object.

In this way Alice is guaranteed that her "root of trust" for the object will not change over time, as long as her inbox is not tampered with (discussed in the internal messaging section).

SECTION 3.6 CHOICES FOR THE ARBITER

While SURGE can use a server as the OT arbiter as described above, SURGE also allows for different arbiters. Recall from the description of Operational Transformation above (Section 2.1) the Arbiter has a fairly simple role. It simply needs to take two operations that could have occurred simultaneously (for instance Alice and Bob in the examples above could have performed their operations at the same time, but it is clear that the version of the object that Alice and Bob had before they went on vacation is an earlier version then what they had after they came back from vacation) and give all the parties the same (arbitrary) information to help them resolve the transformation of operations when there are multiple possible options.

One solution for an arbiter is to have an untrusted server giving incrementing numbers to operations that are inbound (this is the solution described in the design section above). This allows all clients (Alice, Bob, etc.) to put a definitive priority ordering on the operations, and to therefore to achieve the same result after performing the operational transformations. It is possible to not even need to care about a server maliciously giving numbers to incoming operations due to check-ability using fork*-consistency. Additionally, the server could give preference to Alice's operations over Bob's when they both could have come in at the same time (as in our examples above), but that it is irrelevant since one of the points of properly designed operational transforms is not to make it hard to fix an "incorrect" transform, if performing an "incorrect" transform is even doable.

Therefore, since the server is allowed to arbitrarily favor users it is possible to eliminate the arbiter role from the server entirely. For instance, operations can be put in a priority ordering based on the unique identifier of the clients committing operations, and the hash chain value corresponding to the operations that came before them. As an example, in order to arbitrate between operations committed by Alice and Bob it could be agreed upon that since lexigraphically Alice comes before Bob that Alice's operations will be given "priority". This manages to remove any need for a server and therefore allows the system to make use of a simple cloud storage medium instead of a server. The only real downside to this approach is that there must be some system in place for properly giving identifiers to all users. While in the case of Alice and Bob this may not be problematic, if Alice is using multiple devices to pose as herself there must be a way of identifying "Alice1", "Alice2", "Alice3", etc. so that the multiple Alice clients can have a sorted ordering. The simplest way to do this is to just give all of Alice's devices large random numbers upon creation, then the probability that two IDs are identical is practically zero. Alternatively, Alice could register all new devices and get new unique IDs for them such as incremental IDs "Alice1", "Alice2", "Alice3".

SECTION 3.7 SURGE FOR APPLICATION DEVELOPERS

To make use of SURGE for a new application a developer has three main tasks. Firstly, if the application requires collaboration on a new type of data, such as some custom struct, then the developer must create this new data object complete with serialization/deserialization routines. For the majority of tasks this step will be simple, but it is nonetheless an important step in making SURGE work for the new application

Secondly and most importantly the developer must create the operations that modify the target data object. Operations on data are fairly simple and simply express a transformation from a *prevState* to a *newState*. For example, an operation on a String could be a simple as the "Append Zero" operation where $newState \leftarrow prevState||'0'$. On the other hand operations can be generic such as an "Insert" operation on a String where the operation is constructed using a String *insertionString* and a location *insertionIndex* to create the operation:

$$newState \leftarrow Insert(prevState,\ insertionData,\ insertionIndex).$$

Perhaps the most difficult part about creating new data operations is figuring out how to perform OT on the new operations (for more information about implementing Operations see the Operations and Permissions Library section).

Thirdly, the application developer may desire custom permissions to go with the custom operations for the application. In this case, the application developer will likely only need to make custom PublicWritePermissions, which are defined and detailed in section 4.1.1 below. If so then the work will mainly be in defining what types of operations are allowed by the particular type of PublicWritePermission granted to users.

Finally, as with all libraries the developer must deal with the "glue code" that gets SURGE to talk to the rest of the application. Thankfully, with SURGE this is relatively easy with most of the work involved in getting any sort of User Interface to properly express all of the different possible permissions that the developer might want the user to be allowed to grant/revoke. Once all this is done the application developer can bundle up his new data object and operations with the rest of his application and ship it out to users. Note that if the developer chooses to use a server based arbiter that the server does not need to be aware of the data objects and operations specific to the application and it can instead just be a generic SURGE compatible server.

SECTION 4 SURGE API

The API is broken up into a number of components: Object Containers, Object Updaters, Permissions, and Operations.

SECTION 4.1 PERMISSIONS

The control over the permissions system is what gives SURGE so much more functionality over other similar systems. To do so it has four simple types of permissions:

Section 4.1.1 PUBLIC WRITE PERMISSION

Type Signature:

bool verifyData(byte[] signature, byte[] data)

bool allows<T>(T operation)

Public Write Permissions are essentially public verification keys, but that do not just check that cryptographic signatures are valid, they also check whether a particular operation is allowed. Therefore, there are two broad categories of PublicWritePermissions, those that implement cryptographic primitives (i.e. RSA Public Keys) and those that leverage existing cryptographic primitives but implement new "allows" functions. The only instance of a cryptographically based PublicWritePermission in the SURGE prototype is the RSAPublicVerifyKeyPermission, which is essentially just an RSA Public Key that allows no operations. On the other hand the framework has a number of Public Write Permissions that do not bother with new cryptographic measures, but instead focus on the allows functions. These include the AllPermission, TypeBasedPermission, ReplaceSelfWritePermission. Pseudo-code for these permissions follows:

```
class AllPermission : PublicWritePermission
    PublicWritePermission fWritePerm
    AllPermission(PublicWritePermission writePerm)
        fWritePerm = writePerm

    bool allows<T>(T op)
        return true

    bool verifyData(byte[] signature, byte[] data)
        return fWritePerm.verifyData(signature, data)


class TypePermission : PublicWritePermission
    PublicWritePermission fWritePerm;
    Type fType
    TypePermission(PublicWritePermission writePerm, Type t)
        fWritePerm = writePerm
        fType = t

    bool allows<T>(T op)
        return op is fType
    ...

class ReplaceSelfWritePermission : PublicWritePermission
    PublicWritePermission fWritePerm;
    TypePermission(PublicWritePermission writePerm)
        fWritePerm = writePerm

    bool allows<T>(T op)
        if (op is ReplaceWritePermissionOp)
            return ((ReplaceWritePermissionOp) op).OldKey.Equals(fWritePerm)
        return false

    ...
```

Public Write Permissions are a key component which allow users to determine whether an operation was done by a valid user. For most all intents and purposes they may be treated as Public Keys with some metadata tag indicating the intent of the permission. Developers should AVOID making new Public Write Permissions that use cryptography unless they are confident in their abilities since some mistakes could be catastrophic (e.g. including private information in the permission). However, developers should feel free to either use inheritance or more commonly the composite pattern to be able to implement new versions of the "allows" function.

It is worth pointing out that the ease of defining new PublicWritePermissions using the composite pattern really gives SURGE a lot of flexibility. For instance to allow users to perform any operation of a particular type (say an Insert into String Operation) a TypeBasedPermission is a great tool. However, the developer can have even more control over the kinds of operations a user can perform, such as only being allowed to perform ReplaceWritePermissionOps when the permission being replaced belongs to a particular user. In fact, it turns out that the ReplaceSelfWritePermission is extremely useful for having users be able to replace their own compromised credentials (see Section 5.1 below).

Furthermore, notice that a user can be granted a TypeBasedPermission for an Insert Operation that inserts data into the object without necessarily being granted read permission for the object. This is thanks to the object having an asymmetric box key to preserve confidentiality even in the case of write-only permissions.

Section 4.1.2 Write Permission

Type Signature:

byte[] signData(byte[] objectData)

PublicWritePermission toPublicWritePermission()

Write Permissions are simply the necessary counterpart to their Public friends. If Public Write Permission are basically Public Keys that allow verifying input, then Write Permissions also allow signing that input. As a result the SURGE prototype currently has a single WritePermission, the RSAPrivateSigningKeyPermission, which is essentially an RSA private key. For this reason developers should not need to create new Write Permissions unless they are making use of cryptographic primitives in which case caution should be utilized as described for PublicWritePermissions above.

Section 4.1.3 Public Read Permission

Type Signature:

byte[] encryptData (byte[] objectData)

The SURGE prototype currently has two Public Read Permissions: RSAPublicEncryptionKeyPermission and WrappedPermission. RSAPublicEncryptionKeyPermission is, as expected, a simple RSA Public Encryption Key. On the other hand WrappedPermission is a bit more complex. A WrappedPermission is essentially a PublicReadPermission encrypting/wrapping a number of nested ReadPermissions. For instance, the most commonly used WrappedPermission is a user's RSAPublicEncryptionKeyPermission wrapping an AESKeyPermission that in turn wraps a Box's RSAPrivateDecryptionKeyPermission. A WrappedPermission can be unwrapped into an

UnwrappedPermission by using the ReadPermission corresponding to the PublicReadPermission that corresponds to the highest level of encryption for the WrappedPermission.

While Public Write Permissions are necessary because they allow the user to verify their object, Public Read Permissions are more like place markers. These place markers ensure that in the event of changes in Read Access to the object the object's keys are made available to the correct users. Additionally, Public Read Permissions allow for the existence of Write-only permissions that are still confidential. Public Read Permissions can for the most part be treated simply as Public Encryption Keys, developers should not need to implement new ones unless they plan on making use of cryptographic primitives.

Section 4.1.4 READ PERMISSION

Type Signature:

byte[] decryptData(byte[] encryptedObjectData)

PublicReadPermission toPublicReadPermission()

The SURGE prototype currently has three ReadPermissions: RSAPrivateDecryptionKeyPermission, AESKeyPermission, UnwrappedPermission. RSAPrivateDecryptionKeyPermission and AESKeyPermission are, as expected, simply RSA Private Decryption Keys and AES Keys respectively. UnwrappedPermissions are, as described above, simply the ReadPermissions that result from unwrapping a WrappedPermission with the correct key. UnwrappedPermissions are used within the framework to make use of WrappedPermissions, but should be of no concern to the average SURGE developer.

SECTION 4.2 OPERATIONS

SURGE Operations are roughly the same as the operations in any operational transform based system.

SURGE Operations have the following signature:

```
T applyTo(T previousState)
```

```
Operation<T> transform(Operation<T> op)
```

The "applyTo" function simply performs the operation on some old state and returns a new one. The "transform" function deals with operational transformation and returns a modified operation op' that should be applied instead of op to a state where the current operation has already been applied. Most of the time application developers spend working the SURGE will likely be creating new operations. As a result, the SURGE prototype and collaborative text editor application have many operations. Some of the examples important to the SURGE Operations and Permissions library will be detailed in that section.

In general, developers will almost certainly need to develop new operations specific to their application. While the "applyTo" functions are generally extremely obvious to implement (they are programmed the same way as any regular function), the "transform" functions take a little more thought and some experience with operation transformation in order to properly develop them. Developers are encouraged to spend some time reading up on implementing operational transformation beyond this paper.

SECTION 4.3 BOXES

As described in the Section 3.1 Boxes are designed to make it easy for application developers to apply operations to their data and to retrieve the data without having to worry about the inner workings of SURGE. To that effect below is the signature for PrivateBox<T>, a Box that manages data of type T and encrypts all operations on the data before sending them to the cloud.

Signature for PrivateBox<T>:

```
applyDataOperation(Operation<T> op, WritePermission perm)
```

applyACLOperation(Operation<ACL> op, WritePermission perm)

T GetData()

ACL GetACL()

GetUpdates()

SendUpdates()

Additionally there are Box Factory methods for Creating and Retrieving Boxes with the signatures:

CreateBox<T>(IAddress address, WritePermission initialWritePermission, ReadPermission userReadPerm, PublicReadPermission boxPermission)

GetBox<T>(IAddress address, WritePermission initialWritePermission, ReadPermission userReadPerm)

While many of the above operations may be self-explanatory a few (notably the factory methods) require a little more explanation. The major methods of PrivateBox revolve around getting and modifying the data or ACL. However, as Boxes also deal with sending/receiving operations to/from they somehow need to deal with all the particulars of interfacing with a particular cloud provider, etc. As a result, this is taken care of in the constructor to the Box Factory which takes in an Updater interface for dealing with cloud providers. Finally, there are Box constructors made available through the Box Factory. Both take the address of the Box (to be used by the internal Updater), the "root of trust" for the object, and the user's ReadPermission needed for decrypting incoming operations. However, the "CreateBox" function also requires the starting PublicReadPermission (i.e. public encryption key) for the Box.

SECTION 5 OPERATIONS AND PERMISSIONS LIBRARY

While SURGE offers a tremendous amount of flexibility in terms of designing new permissions, there are a number of common use cases that developers would probably rather not implement themselves. To that extent SURGE offers a standard library of Operations and associated Permissions.

SECTION 5.1 REPLACING COMPROMISED CREDENTIALS

In a tradition ACL model, as well as those proposed by alternative such as SPORC, there is a set of administrators that is in charge of managing keys/permissions for access to the object content. However, if the administrator Alice grants Bob read permission to the file and Bob's private key becomes compromised or expires then Bob has to ask Alice to replace his old read permission with a new one. However, using our framework this problem is easily resolvable by giving Bob a verifiable permission to replace his key with a new key. If the scenario is, as in Figure 8, that Bob needs permission to update his Read Permission then Bob can be granted the TypeBasedPermission to allow the

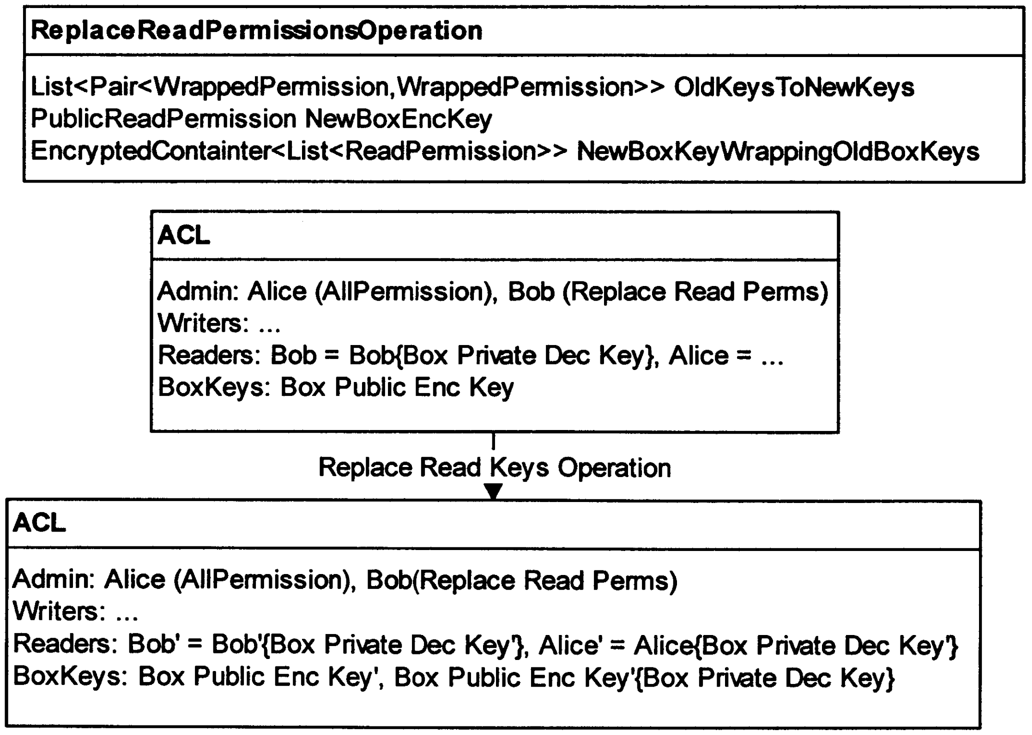
ReplaceReadPermissionsOperation
List<Pair<WrappedPermission,WrappedPermission>> OldKeysToNewKeys
PublicReadPermission NewBoxEncKey
EncryptedContainter<List<ReadPermission>> NewBoxKeyWrappingOldBoxKeys
ACL
Admin: Alice (AllPermission), Bob (Replace Read Perms)
Writers: ...
Readers: Bob = Bob{Box Private Dec Key}, Alice = ...
BoxKeys: Box Public Enc Key
Replace Read Keys Operation
ACL
Admin: Alice (AllPermission), Bob(Replace Read Perms)
Writers: ...
Readers: Bob' = Bob'{Box Private Dec Key'}, Alice' = Alice{Box Private Dec Key'}
BoxKeys: Box Public Enc Key', Box Public Enc Key'{Box Private Dec Key}

FIGURE 8 THE WAY THE ACL OF AN OBJECT SHARED BY ALICE AND BOB CHANGES AFTER BOB USES HIS PERMISSION TO REPLACE HIS READ KEY.

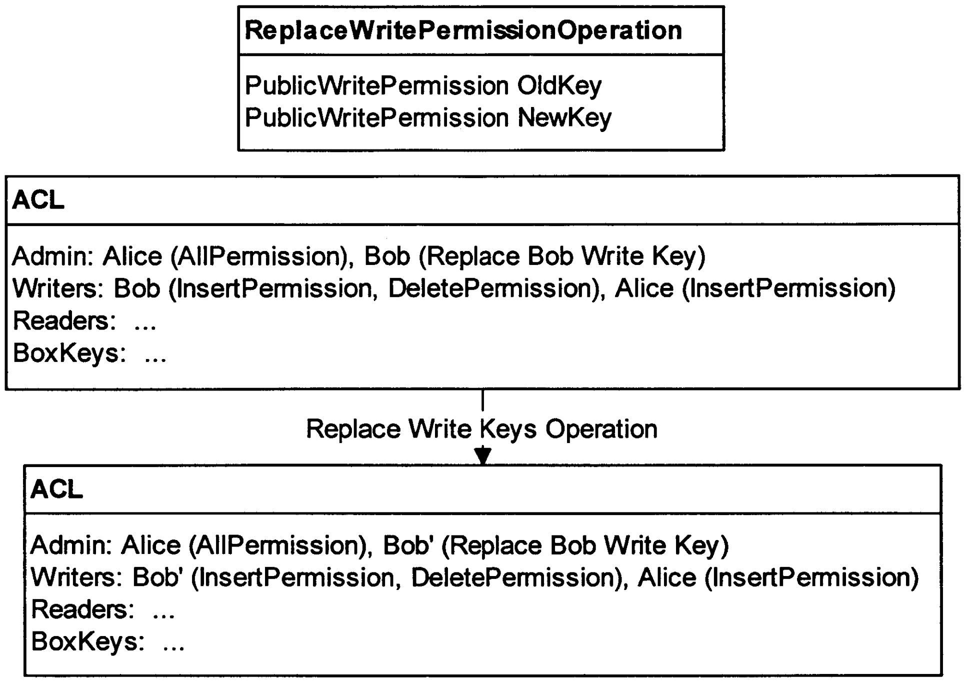
ReplaceWritePermissionOperation
PublicWritePermission OldKey
PublicWritePermission NewKey
ACL
Admin: Alice (AllPermission), Bob (Replace Bob Write Key)
Writers: Bob (InsertPermission, DeletePermission), Alice (InsertPermission)
Readers: ...
BoxKeys: ...
Replace Write Keys Operation
ACL
Admin: Alice (AllPermission), Bob' (Replace Bob Write Key)
Writers: Bob' (InsertPermission, DeletePermission), Alice (InsertPermission)
Readers: ...
BoxKeys: ...

FIGURE 9 THE WAY THE ACL OF AN OBJECT SHARED BY ALICE AND BOB CHANGES AFTER BOB USES HIS PERMISSION TO REPLACE HIS WRITE KEY.

ReplaceReadPermissionsOperation. In this way Bob is allowed to generate new read keys for the Box, and generate new WrappedPermissions for all of the users with read permission to the Box. It is worth noting that Bob can in fact commit incorrect WrappedPermissions for other users to deny them access. However, this particular abuse is easily detectable (the user is denied permission) and easily recoverable (either by rolling back to before the bad operation, or by having a user with a correct WrappedPermission properly refresh keys). On the other hand, the scenario described in Figure 9, where Bob is given access to update his Write Permissions, does not have any such downsides. To allow Bob to update his Write Permissions he can be granted a ReplaceSelfWritePermission. Then when the ReplaceSelfWritePermission is executed only Bob's Write Permissions are affected and they are simply replaced with cloned permissions that use the *verifyData* function from the *NewKey*.

SECTION 5.2 GROUPS

The ability to replace compromised/expired credentials directly leads to the ability to have user groups. The idea behind a user group is that Alice would like to be able to share her documents with all of her co-workers (which there may be many of) without having to separately encrypt and store the Object Key for each co-worker. Additionally, it is a nice usability to be able to share objects with groups of people without having to manually give permission to the same set of users for every object planning on being shared.

Groups can be implemented by creating a Group data object which is essentially just two asymmetric key pairs for signing and encrypting, and a list of names/identifiers of the objects shared with the group. These key pairs should have the public key portion publicly visible, as a simple example it could even be the name/identifier of the object. Users who have read permission to the Group have access to the group's private asymmetric keys and can therefore pose as members of the group. For instance, as shown in Figure 10, if Alice gives a Group G read/write permission to an object and Bob has read permission to G then Bob can read/write to the object using G's private key. If G's group administrator, Adam, later decides he wants to revoke Bob's group membership then Adam will change G's asymmetric keys. Additionally, since the group object contains the objects shared with the group Adam can go to every object and request that the group key be replaced with the new group key. It is worth noting that depending on how particularly groups are set up the individual users in the group may be empowered to change which key is used to keywrap the Object Keys of objects shared with the group. This is important for legitimate reasons, such as if one of the group members' keys is compromised, but could also be used for more malicious purposes, such as replacing the group key used by an object with a user's own key. However, such a change would be obvious to group users (i.e. easily

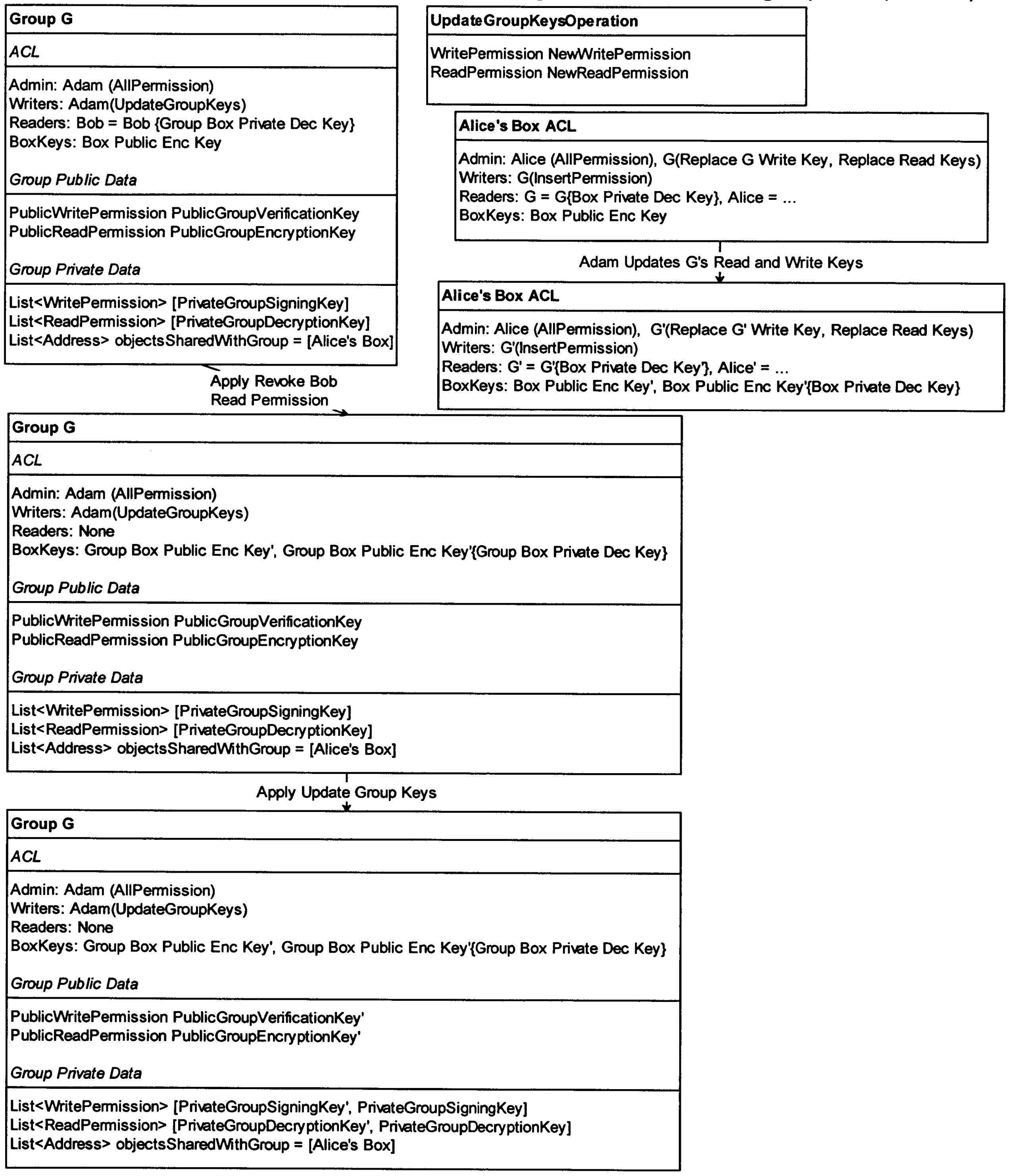
Group G
ACL
Admin: Adam (AllPermission)
Writers: Adam(UpdateGroupKeys)
Readers: Bob = Bob {Group Box Private Dec Key}
BoxKeys: Box Public Enc Key
Group Public Data
PublicWritePermission PublicGroupVerificationKey
PublicReadPermission PublicGroupEncryptionKey
Group Private Data
List<WritePermission> [PrivateGroupSigningKey]
List<ReadPermission> [PrivateGroupDecryptionKey]
List<Address> objectsSharedWithGroup = [Alice's Box]
UpdateGroupKeysOperation
WritePermission NewWritePermission
ReadPermission NewReadPermission
Alice's Box ACL
Admin: Alice (AllPermission), G(Replace G Write Key, Replace Read Keys)
Writers: G(InsertPermission)
Readers: G = G{Box Private Dec Key}, Alice = ...
BoxKeys: Box Public Enc Key
Adam Updates G's Read and Write Keys
Alice's Box ACL
Admin: Alice (AllPermission), G'(Replace G' Write Key, Replace Read Keys)
Writers: G'(InsertPermission)
Readers: G' = G'{Box Private Dec Key'}, Alice' = ...
BoxKeys: Box Public Enc Key', Box Public Enc Key'{Box Private Dec Key}
Apply Revoke Bob
Read Permission
Group G
ACL
Admin: Adam (AllPermission)
Writers: Adam(UpdateGroupKeys)
Readers: None
BoxKeys: Group Box Public Enc Key', Group Box Public Enc Key'{Group Box Private Dec Key}
Group Public Data
PublicWritePermission PublicGroupVerificationKey
PublicReadPermission PublicGroupEncryptionKey
Group Private Data
List<WritePermission> [PrivateGroupSigningKey]
List<ReadPermission> [PrivateGroupDecryptionKey]
List<Address> objectsSharedWithGroup = [Alice's Box]
Apply Update Group Keys
Group G
ACL
Admin: Adam (AllPermission)
Writers: Adam(UpdateGroupKeys)
Readers: None
BoxKeys: Group Box Public Enc Key', Group Box Public Enc Key'{Group Box Private Dec Key}
Group Public Data
PublicWritePermission PublicGroupVerificationKey'
PublicReadPermission PublicGroupEncryptionKey'
Group Private Data
List<WritePermission> [PrivateGroupSigningKey', PrivateGroupSigningKey]
List<ReadPermission> [PrivateGroupDecryptionKey', PrivateGroupDecryptionKey]
List<Address> objectsSharedWithGroup = [Alice's Box]

FIGURE 10 THE MODIFICATION OF GROUP G AND ALICE'S BOX WHICH IS SHARED WITH G, AS BOB IS REMOVED FROM G.

detectable) and it would be clear which user performed the malicious change since the objects are composed of operations which essentially results in a built in logging system.

Groups as put forth in this section act as a single entity. As a result if a particular member of a group performs some malicious edit the group as a whole can be blamed, but not any particular user. In some cases this may be the intended behavior, however if the developer wants to be able to keep track of the users acting on behalf of the group this too is doable. In fact, the only two modifications are required. The first is that G's public data should also contain the PublicWritePermissions for all users in the group. The second is that instead of Alice granting G write permissions like TypeBasedPermission(InsertOperation, G's RSAPublicVerificationKeyPermission) instead grant a TypeBasedPermission(InsertOperation, GroupWritePermission(G's RSAPublicVerificationKeyPermission, GroupBox)). Then when the GroupWritePermission verified the signature it would check to see if any of the users in G's list of user PublicWritePermissions verified the signature.[2]

SECTION 5.3 ANONYMITY

For some systems anonymity might be an important requirement. Our definition of anonymity is that when a user Alice wants to share an object with Bob and Charlie she may know who Bob and Charlie are (or maybe not, she may just know which public key to use but not who it's for), but Bob and Charlie should not know who each other are. Additionally, Alice should not be able to pose as Bob or Charlie. Finally, the system should allow for the existence of an "auditor" who will in the event of a user performing malicious operations be able to identify the user responsible at least as well as the granter of

[2] Note that going through all the users is unnecessary, and a pointer such as "#5 in the list" could be made as part of the user's signature for the group.

the original permission (for instance the should be able to identify Bob at least as well as Alice can, assuming Alice is cooperative).

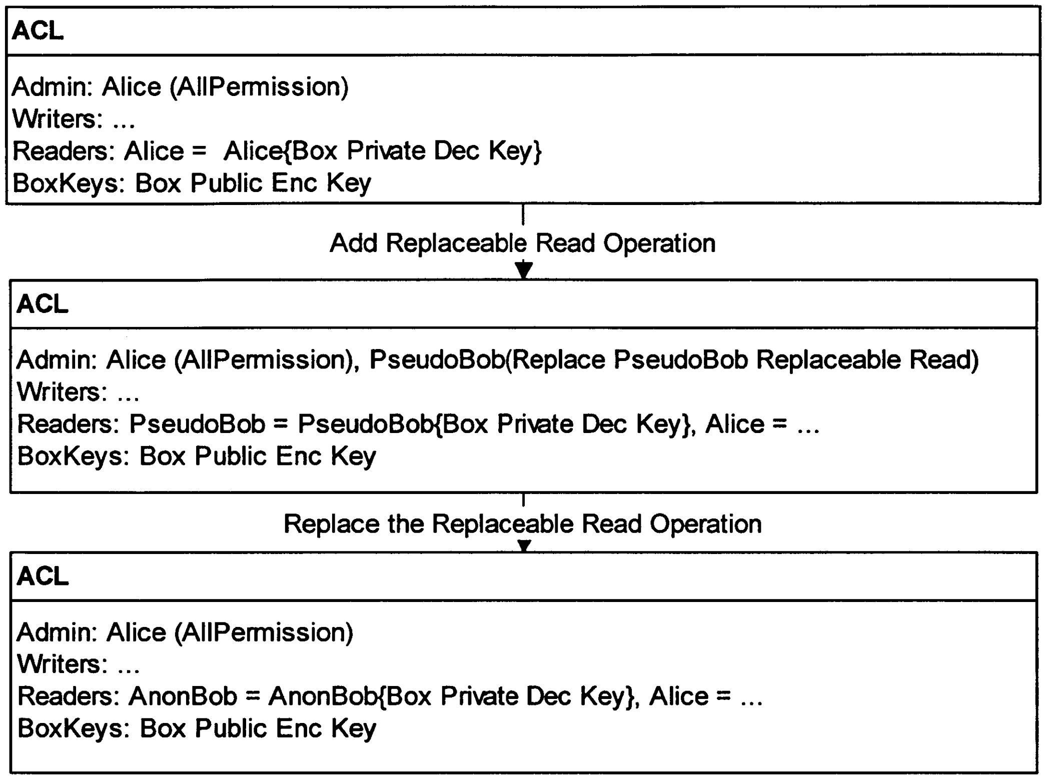
ACL
Admin: Alice (AllPermission)
Writers: ...
Readers: Alice = Alice{Box Private Dec Key}
BoxKeys: Box Public Enc Key
Add Replaceable Read Operation
ACL
Admin: Alice (AllPermission), PseudoBob(Replace PseudoBob Replaceable Read)
Writers: ...
Readers: PseudoBob = PseudoBob{Box Private Dec Key}, Alice = ...
BoxKeys: Box Public Enc Key
Replace the Replaceable Read Operation
ACL
Admin: Alice (AllPermission)
Writers: ...
Readers: AnonBob = AnonBob{Box Private Dec Key}, Alice = ...
BoxKeys: Box Public Enc Key

FIGURE 11 ALICE GRANTING BOB AN ANONYMOUS READ PERMISSION

To implement this we can use a two-step process for granting permissions as shown in Figure 11 for Anonymous Read Permissions and Figure 12 for Anonymous Write Permissions. First, if Alice wants to grant a permission to Bob then Alice generates a new asymmetric key pair and grants this new key pair the permission she was going to give to Bob. Alice then communicates the generated key pair (including the private key(s)) to Bob in the initial messages to him letting him know about his newly granted permissions. When Bob first accesses the file he then replaces the key pair used for the permission he received using the mechanism from "Replacing Compromised Credentials" with a new key pair generated just for that object. Now while Alice knows that Bob is the one using the replaced credentials Charlie does not, since the information regarding the initial set of (now replaced) asymmetric keys was done via the message that Alice sends Bob about his being granted a new permission, which is secret to Charlie.

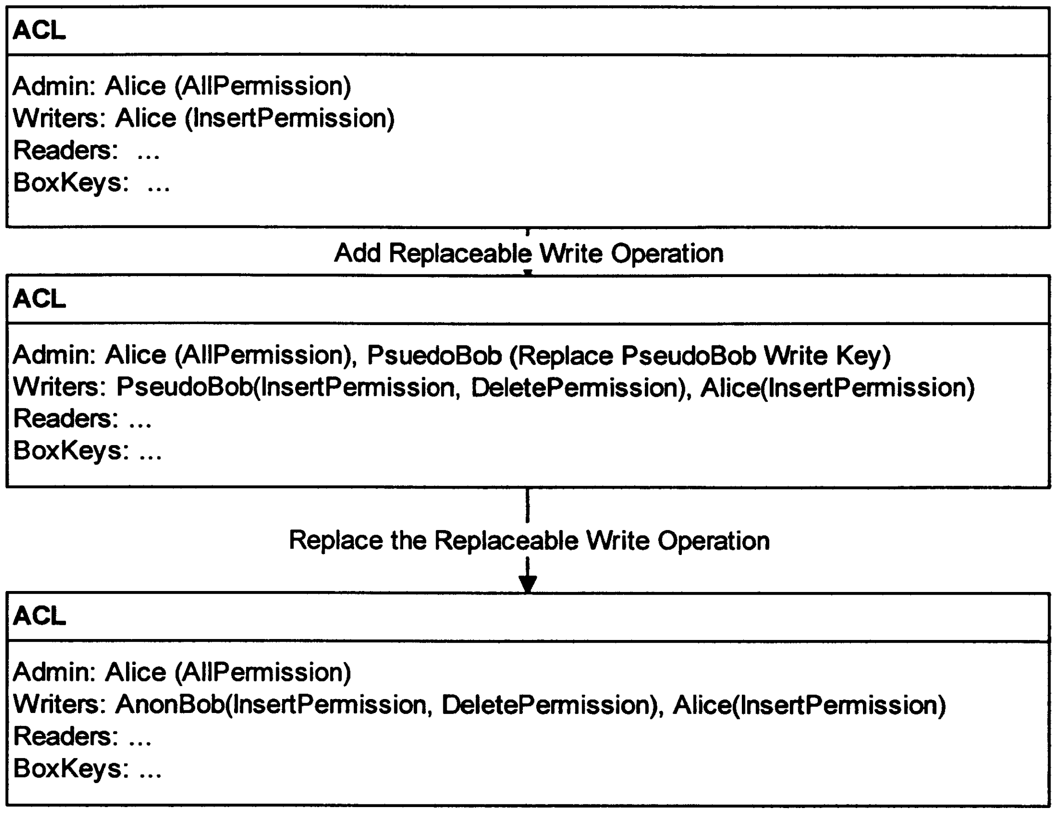
ACL
Admin: Alice (AllPermission)
Writers: Alice (InsertPermission)
Readers: ...
BoxKeys: ...
Add Replaceable Write Operation
ACL
Admin: Alice (AllPermission), PsuedoBob (Replace PseudoBob Write Key)
Writers: PseudoBob(InsertPermission, DeletePermission), Alice(InsertPermission)
Readers: ...
BoxKeys: ...
Replace the Replaceable Write Operation
ACL
Admin: Alice (AllPermission)
Writers: AnonBob(InsertPermission, DeletePermission), Alice(InsertPermission)
Readers: ...
BoxKeys: ...

FIGURE 12 ALICE GRANTING BOB AN ANONYMOUS WRITE PERMISSION

As mentioned above we may like an auditor to be able to verify that Bob is in fact the person associate with the new key K that Alice generated for him. To do so we augment the scheme above so that instead of Alice giving Bob K and granting K the permission instead Alice gives Bob K and grants K the permission to replace itself with the permission that Alice intends to give to Bob. When Bob goes to execute the replace operation using K he must also sign the new public key he intends to use for the object with his own private key and encrypt that blob for the auditor. In this way Alice and the auditor know what operations Bob has committed, but neither Charlie nor anyone else should be able to determine that information.

As an aside, there are a number of possible ways to deal with the issue of non-repudiation. For instance the above scheme works as long as the auditor can know for sure that Bob has made at least one commit using the key that replaced K. However, if Bob denies ever having had K it may no longer becomes possible to easily prove that he did. Some ways to deal with this include playing Alice and Bob off each other since ultimately if there is malicious behavior caused by someone who had access to K it must be either Alice's or Bob's fault/responsibility. One solution to this behavior is for Bob to encrypt the message signed for the auditor in the above scheme for both the auditor and Alice, and perhaps even have Alice sign this signed version to acknowledge that she believes Bob was making use of the key that replaced K. This scheme is secure, however it requires more state management then the other applications we have listed. However, it is likely that Bob will not be able to plausibly deny ever having had access to the object due to other mechanisms external to the system and therefore for many applications the first scheme mentioned may in fact be sufficient.

SECTION 5.4 INTERNAL MESSAGING

Throughout this paper there have been references to some "initial sharing message" between the granter of permissions to an object, Alice, and the one receiving the permissions, Bob. While many other systems such as SPORC and Dropbox tend to use external communication, such as email, for this initial message our system allows this communication to be done internally. The requirements of such a system are that any valid user of the system should be able to send a message to any other valid user, the message should only be readable by the intended receiver of the message, and that the message once received cannot be "un-received". As we are trusting the system managers for availability purposes we cannot, without external communication, resolve issues where the system managers prevent some users from communicating with other users whether maliciously or as a result of some unintentional system partition.

To do this we can make use of our standard shared objects where the name/identifier of the object is publicly and obviously related to the user's ID. Let our shared object be a list of "Message" objects, then the only operation that needs to be performed on it is the operation Append(message), and any user of the system should be able to perform that operation. Since, as mentioned in the system design, our system allows write-only permissions it is possible to grant users append permission without read permission or any other write permissions. In fact, implementing the operation transform aspects of Append is easy since we have no inherent requirements that one message is delivered before another so it does not matter whether the object looks like [message1, message2, message3] or [message1, message3, message2]. Finally, it is trivial to grant everyone in the system the Append permission by just having the client side validator always return true when asked if a user U has Append permission to the object.

SECTION 6 EVALUATION

SECTION 6.1 APPLICATION: COLLABORATIVE TEXT EDITOR

To see the effectiveness of SURGE and its associated library of permissions in creating new secure collaboration applications I decided to make a collaborative text editor. Both SURGE and the application were built in C#. The vast majority of the collaborative text editor code was in creating the GUI for the application and the hooks for thread-safe handling of updates. In fact, since the SURGE API and Library allows for most of the actions that a user might want to perform on the data (modify it, share it, modify permissions to it, etc.) to be as simple as for a non-secure alternative almost all of the non-GUI work came in implementing the text-editor specific operations. In particular, the text-editor needed to implement the Insert(Index, Data) and Delete(Index, Length) operations each of which was about 100 lines of code (although the logic was the same as for the Insert and Delete operations described in other operation transform documents such as the Google Wave paper and its open source counterpart). Additionally, as an implementation choice instead of attempting to keep track of the changes to the document as they occur I leveraged Google's open-source Diff Match and Patch (http://code.google.com/p/google-diff-match-patch/) to get the insert and delete operations by comparing the document contents between two commits.

Likely the most complicated and difficult part of developing the collaborative text editor was in deciding both what types of permissions to expose to the user (for instance, do I really want them to have write-only permission?) as well as how to develop a GUI to properly expose permissions, groups, etc. to the user. However, these are the types of concerns that application developers should have to deal with since they concern making active decisions about which security mechanisms to expose to their users, and how to do so. As a result it seems that SURGE, even as a prototype, makes developing a secure cloud-based collaboration application even easier than making an equivalent cloud-based collaboration application that attempts to secure user data by using server based authentication instead of cryptography. Moving forward it should hopefully be an easy choice for application developers looking to make new cloud-based collaboration applications like Google Docs or Office 365 to decide to use SURGE and real cryptographic security for their users as opposed to using server side authentication as Google and Microsoft do.

SECTION 7 RELATED WORK

There have been a number of papers in the area of secure file systems, including Cryptree (Grolimund, Meisser, Schmid, & Wattenhofer, 2006), Tresorium, Plutus (Kallahalla, Riedel, Swaminathan, Wang, & Fu, 2003), and SiRiUS (Goh, Shacham, Modadugu, & Boneh, 2003). While these systems manage to get multiple users to be able to securely share resources stored in the cloud, because they were developed as file systems they lack any reasonable ability for real time collaboration. Additionally, while some of these systems, such as Tresorium, allows for Groups they are not implemented in a generic way and therefore other benefits of SURGE (like non-administrative users being able to replace their own credentials) are unavailable. Certainly none of these systems offer the ability for write-only permissions. In the world of secure real-time collaboration SPORC stands out as the major solution providing offline access, confidentiality, integrity, dynamic ACLs and doing this all in real time. However, as mentioned above SPORC misses some aspects like having groups, anonymity, and in general just more flexible ACLs which would be quite useful for real world applications.

In particular, SPORC severely limits the ACL by restricting it to the 3 roles of read, read-write, and administrator. Additionally, SPORC lacks any sort of groups and anonymity, as well as relying on some external system to start the object sharing process. Even some basic real world security concerns are left out, like the ability for a user to update his/her keys used in accessing stored data.

SECTION 8 CONCLUSION

A major goal for SURGE was to allow developers to create the types of cloud based collaboration applications that have been in mainstream use for years, but in a way that does not require the clients to have much trust in the service provider. OT has been a big help even in non-secure collaboration systems like Google Wave by allowing these collaboration systems to be optimistic and even be able to work while offline. The flexibility granted by using signed operations on an ACL to allow arbitrary modifications to it leads directly to the ability to grant users the ability to update their credentials, use groups, amongst other advantages. SURGE packages these aspects into an easy to use framework complete with a library of useful permissions and operations for managing ACLs. Finally, SURGE's usability and usefulness was demonstrated by creating a secure collaborative text-editor. Going forward SURGE should hopefully enable new classes of secure collaborative cloud based tools to become available to the general public. In particular the last decade's developments in standard collaborative cloud based tools may be able to finally be used by the security conscious company or individual, allowing them to be more productive than was previously possible.

SECTION 9 REFERENCES

Feldman, A. J., Zeller, W. P., Freedman, M. J., & Felten, E. W. (2010). SPORC: Group collaboration using. *Proceedings of the 9th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 337–350.

Goh, E.-J., Shacham, H., Modadugu, N., & Boneh, D. (2003). SiRiUS: Securing Remote Untrusted Storage. *Proceedings of the 2nd USENIX Conference on File and Storage Technologies*, 29-42.

Grolimund, D., Meisser, L., Schmid, S., & Wattenhofer, R. (2006). Cryptree: A Folder Tree Structure for Cryptographic File Systems. *Proceedings of the 25th IEEE Symposium on Reliable Distributed Systems (SRDS)*, 189–198.

Kallahalla, M., Riedel, E., Swaminathan, R., Wang, Q., & Fu, K. (2003). Plutus: Scalable Secure File Sharing on Untrusted Storage. *Proceedings of the 2nd USENIX Conference on File and Storage Technologies (FAST)*, 29-42.

Lam, I., Szebeni, S., & Buttyan, L. (2012). Tresorium: cryptographic file system for dynamic groups over untrusted cloud storage. *Submitted to 4th International Workshop on Security in Cloud Computing*.