In *Second International Workshop on Security and Privacy in Cloud Computing*, Minneapolis, MN. 24 June 2011.

Private Editing Using Untrusted Cloud Services

Yan Huang and David Evans
University of Virginia
MightBeEvil.com

***Abstract*—We present a general methodology for protecting the confidentiality and integrity of user data for a class of on-line editing applications. The key insight is that many of these applications are designed to perform most of their data-dependent computation on the client side, so it is possible to maintain their functionality while only exposing an encrypted version of the document to the server. We apply our methodology to Google Documents and describe a prototype extension tool that enables users to use a cloud application to manage their documents without sacrificing confidentiality or integrity. To provide adequate performance, we employ an incremental encryption scheme and extend it to support variable-length blocks. We analyze the security of our scheme and report on experiments that show our extension preserves most of the cloud application's functionality with less than 10% overhead for typical use.**

***Keywords*-cloud security, incremental cryptography, data privacy.**

I. Introduction

The idea of software-as-a-service, combined with client side technologies that make rich content web pages possible, enables web applications to supplant their desktop peers. Example applications include Google Docs [16], Microsoft Office Live [24], and the Mozilla Bespin code editor [27]. Moving document editing applications to external servers offers many advantages including availability (documents are accessible from any Internet connection), reliability (cloud providers use redundancy to provide resilience to failure), reduced cost (most of these services are free to end-users), low maintenance costs (the service can be transparently updated by the server), as well as enhanced opportunities for collaboration.

Despite these advantages, many people are understandably reluctant to hand over their confidential documents to a third party. Unlike local files, the user must fully trust the service provider and delegate the file's access control to the web application. Trusting users may rely on the provider's privacy policies, but even if a user trusts the provider to behave responsibly, service implementation errors can lead to data compromise. For example, Google Docs makes embedded images publicly available and leaks information about previous versions of documents [1]. In March 2009, a bug in Google's server inadvertently shared some documents in Google Docs contrary to the user's selected permissions [20].

The goal of our work is to provide a way for users to benefit from the advantages of using cloud applications to edit documents without requiring them to expose their sensitive data to the service provider. Our approach is to encrypt the submitted user content communicated between the client and server. Instead of storing the plaintext document in the cloud, the cloud stores the ciphertext document which is decrypted by a trusted extension running in the client. Messages to the server that update the document are transformed into messages that produce the corresponding update on the ciphertext document.

To evaluate the feasibility and efficiency of our approach we develop a prototype browser extension for Google Documents that satisfies these design goals:

1) The contents of the file are protected (both confidentiality and optionally integrity) even against attacks from a possibly malicious cloud service provider.
2) The extension has minimal impact on the existing functionality of the cloud application and requires no cooperation from the application provider. Section IV describes our Google Documents extension; to demonstrate the generality of our approach, we also built similar extensions for Mozilla Bespin and Adobe Buzzword.
3) The incurred runtime and bandwidth costs are acceptable for typical uses. We achieve this by using a new data structure that supports variable-length blocks in an incremental encryption scheme (Section V). Section VII reports results from our performance experiments.

Section II describes our threat model. Our design is secure against an adversary who does not maliciously manipulate the client application, which limits the adversary but is a realistic threat model for many cloud application scenarios. Providing full confidentiality against an adversary who maliciously constructs the client is probably infeasible as it requires defeating a rich set of possible covert channels as we discuss in Section VI-B. We do not aim to improve availability here. Our approach still relies on the cloud provider to store the user's data, so a malicious or incompetent cloud provider can easily prevent users from accessing their documents. This could be addressed using replication with multiple cloud providers, but this is outside the scope of this paper. Our goal is to protect users from cloud providers that expose or alter their data. This could happen because of malicious actions by the service provider, in response to a subpoena, as part of a business model to provide targeted advertising, or through a bug in the server application's implementation.

Prior Work. Most previous secure storage systems were built around existing OS file systems [5, 23, 25, 26]. Our work relies on the presumption that we can treat the web application server as a storage server, at least for a restricted set of contents. In addition, our work resembles Blaze's [5] and Microsoft's [25] in that the storage server is not trusted. However, our research is distinct

from previous work our focus on security issues of contents in on-line editors presented in the form of web applications, instead of the file system component in a operating system. Here the support for incremental cryptographic operations is of great importance.

In addition, several other projects consider security issues in cloud computing. The work most similar to ours is CoClo [12], which provided a lightweight, client-side tool for preserving privacy in cloud applications including Google Docs. Compared to their work which requires reencrypting and transmitting the entire document for every update, we focus on integrating incremental encryption which is vital for efficiently editing medium to large size documents. We also develop a novel data structure to reduce ciphertext size explosion associated with incremental encryption. Section VIII covers more broadly related work.

Contributions. The main contributions of this work are:

1) We describe an easily-deployable, lightweight solution to mitigate the threat of data compromise inherent in all cloud computing document editing applications. Our approach works for an important class of such applications, and requires no modification of the service or client program (Section III).
2) To enable an efficient solution, we develop a technique that integrates incremental encryption with a tree-based data structure (Section V). The multiple-character block extension enables performance tradeoffs between ciphertext size and encryption time (Section V-C).
3) We develop and analyze a prototype browser extension that demonstrate our approach for several representative applications. We provide an analysis of the security (Section VI) and performance of our extension for Google Documents (Section VII). For most user editing operations, the performance overhead is less than 10%.

Target Applications. Our approach can be applied to a broad category of web applications. The common characteristic of those applications is that the server can be thought of just a glorified data store that does not do any processing of data but merely provides storage and retrieval. That is, none of the computations done by the server depend on the actual data so it maintains its core functionality even if the data is encrypted.

Many web applications are designed this way, driven by the design philosophy that the server should offload as much work as possible to the client to improve application responsiveness. The server performs computation only when necessary, usually for information portability reasons. Example applications that mostly fit this model include Google Documents, Mozilla Bespin, and Adobe Buzzword. Nearly all of their essential functionality of these applications can be maintained with encrypted data.

To verify the generality of our approach, we developed proof-of-concept extensions for the Google Documents[1] word processor; Adobe Buzzword, a document editing application; and Mozilla Bespin, an on-line source code editing tool. Section IV describes our extension for Google Documents in detail. We chose Google Documents for demonstration not only because it is a very widely-used on-line office application, but also because it adopts an incremental document update protocol that enables an efficient and interesting approach to secure updates.

Our approach does not work for applications where core functionality is performed by the server. For example, Google's spreadsheet application performs computations on cell contents at the server side. Encrypting cell contents would break essential features such as computing sums of cell values. Providing privacy with a spreadsheet application would require either moving this functionality to the client-side, or providing heavyweight mechanisms to support computation on encrypted data.

II. Threat Model

Our work is focused on the scenario where an end user wants to use a cloud computing application to manage and edit documents, but does not want to trust the application provider with confidential data. We consider security threats that come from the cloud service provider, including attackers that compromise the service provider's infrastructure or use legal mechanisms like a subpoena to obtain access to stored data. We assume an adversary with computational power equivalent to a probabilistic polynomial-time Turing machine who fully controls all the data users store at the server as well as all the messages between the server and the client. The adversary can launch both passive and active attacks on the application's users.

We consider two different threat models: *benign client*, where the web application's client-side code is trusted to be non-malicious; and *malicious client*, where there are no assumptions about the client-side code. The benign client model is appropriate for the scenario where the client code is received from a fairly trusted service provider, but the threats are external attacks against the service provider or security vulnerabilities in the server's implementation.[2] This scenario also applies in cases where neither the server nor the client application provider is trusted, but they are independent parties that are not expected to collude. The malicious client model assumes the service provider controls the client-side application code and can construct a malicious client specifically to transmit confidential data back to the server using covert channels. We do not claim any strong defense against such a powerful adversary, but do provide mechanisms that can limit the amount of confidential data they could obtain.

We assume our browser extension along with the client's browser and host is not compromised. Attacks such as installing a keylogger on the client's machine are outside the scope of our work and better dealt with anti-malware approaches.

Many cloud servers operate without SSL/TLS protection to provide improved performance and reduce server costs.[3] This opens

[1]Google's document service is confusingly named: *Google Docs* refers to the suite of applications which includes word processor (originally known as *Writely* before it was purchased by Google), spreadsheet, and presentation applications. The word processor, which is our target, is known as *Google Documents*.

[2]Contrary to what our project's domain name, mightBeEvil.com, might suggest, this is a reasonable threat model for most users of Google Documents.

[3]In addition, some jurisdictions force ISPs to block specific services over SSL/TLS. The first author has observed this actually happening in China for Google Docs. China currently seems to block all HTTPS access to Google Docs but allows HTTP access.

the door for any malicious party to eavesdrop on client/server communication easily. For these scenarios, our tool also provides some protection against network eavesdropping and tampering. Our focus, however, is on the malicious or buggy cloud provider threat which is not reduced by using SSL to protect the client-server traffic.

In our prototype, users control the security of their data using per-document passwords and may select either a confidentiality-only scheme or one that provides both confidentiality and integrity. Poor selection and management of user passwords is often the weakest link in a security system, but improving this is outside the scope of our work. In our scheme, sharing encrypted documents requires also sharing passwords. We assume parties sharing a document can securely establish a shared password, although there are many better ways of managing passwords for shared documents.

III. Approach

Applying our idea to an on-line editing web application involves two main steps: (1) reverse engineering the application-specific transmission protocol used by the web application; and (2) intercepting all client-server traffic and modifying messages to encrypt all data related to the document's contents. For security, all requests other than those that can be interpreted and encrypted must be blocked.

The reverse engineering can be done by observing network traffic or analyzing the client code. We do this manually, although several recent works indicate that it may be possible to automate the task of reverse engineering network protocols [7, 10, 22].

Several different approaches are possible for the traffic interception and modification:

1) *Standalone proxy.* This is the most general approach, which could work for even non-browser applications (e.g., components of Microsoft Office Live). The main disadvantage of using a proxy is the difficulty in handling encrypted SSL/TLS communication.
2) *Browser extension.* A browser extension can be easily deployed for web applications, and can be implemented using the fairly mature browser extension infrastructure.
3) *User JavaScript.* User JavaScript [21, 29] is a convenient way to inject a piece of JavaScript to run with the same privilege as scripts originally coming from a web site. However, it provides no interface to directly manipulate network traffic. Implementing the transformer using User JavaScript requires deeper understanding of the client code and rewriting relevant components.

We implement our tool as a Firefox browser extension because it is the easier for individual users to deploy than a proxy and the Firefox extension model provides mechanisms for processing SSL-encrypted packets before encryption or after decryption.

Next, we describe the extensions for Bespin and Buzzword, both of which can be built straightforwardly using this approach. Supporting incremental updates, as done by Google Documents, is more interesting, and described in more detail in Section IV.

Bespin. Mozilla Bespin [27] is an on-line source code editing tool developed by Mozilla Labs. It is designed to be an extensible code editing tool inside a browser. All the cosmetic work and editing-related functions (e.g., code highlighting) are provided by extensible client-side JavaScript/CSS modules, while the server is only used as backend storage. The APIs between the client and server are well-defined and open [28]. It simply uses HTTP PUT requests to send user content back to the server stored as a file. No incremental update mechanisms are found in Bespin. By wrapping the PUT request with code that encrypts all user data, the server only sees encrypted contents.

Buzzword. Adobe Buzzword is a document editing service similar to Google Documents. On every update, the client sends back the whole document content as a XML file encapsulated in a HTTP POST request. By encrypting the text embedded in <textRun> tags, we keep submitted document content secure.

IV. Extension for Google Documents

Figure 1 illustrates the design of the extension we built to provide privacy with Google Documents.

A. Analyzing Network Traffic

To protect the confidential content, we need to ensure that no messages are sent to the network that contain unencrypted information about the document's contents. This requires understanding the network traffic well enough to encrypt sensitive data without altering the control parts of messages. To analyze the network communication between a Google Documents client and server, we set up a Squid proxy [32] to mediate requests and responses. Each function is tested separately to examine relevant messages. We observed that only a few of the features provided by Google Documents generate server requests (including "save", "download as", and "check spelling"), while most actions (e.g., content formatting) are taken care of by client scripts.

Client/Server Communication Overview. The client/server update messages appearing in a typical editing session are visible in the right side of Figure 1 (ignoring the messages between the extension and client). When we create a new document or open an existing document, a new edit session is initiated as a POST request sent to http://docs.google.com/Doc?docID=*id*. This editing session lasts until the document is closed. In each editing session, the first save command always results in a POST request in which the docContents field contains the entire document contents. All subsequent save commands in the edit session trigger POST requests that only carry document difference information in the delta parameter. Update deltas are periodically sent back to the server due to automatic save requests triggered by client side timeouts.

The server responds to the content update messages with an Ack message that contains contentFromServer and contentFromServerHash fields, conveying the current content to the best of the server's knowledge. We did not reverse engineer these fields, but instead found that in a single user editing session the client works flawlessly when the values are replaced with an empty

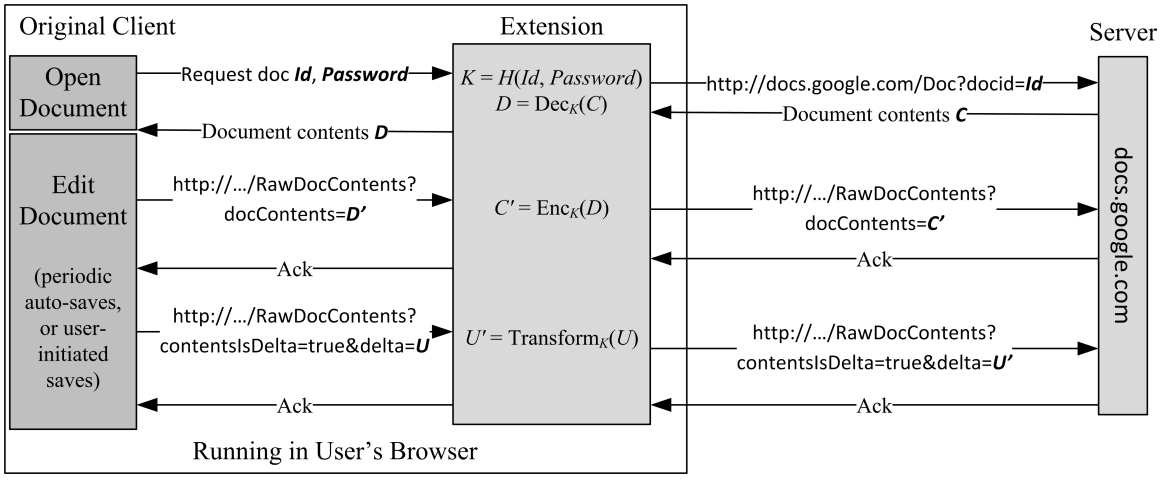


Fig. 1. Private Editing. The server maintains the ciphertext document, C. The browser extension intercepts all client-server traffic, encrypting as necessary.

string for contentFromServer, and 0 for contentFromServerHash. When multiple users edit the same document simultaneously, this causes unnecessary conflicts (see Section VII-A).

Incremental Updates. The document text is represented as a one-dimensional string. For presentation purposes, we introduce an imaginary cursor. For each incremental update, the cursor is initially at position 0, the beginning of the string. The update is described by a delta message, a sequence of operations, separated by tabs. Three types of operations are supported:

- *=num* move the cursor forward *num* characters.
- *+str* insert the string *str* at the current position and advance the cursor to the end of the inserted string.
- *num* delete *num* characters starting from the current cursor.

For example, a delta value of "=2 -5" turns *"abcdefg"* into *"ab"*; and the value "=2 -3 +uv =2 +w" turns *"abcdefg"* into *"abuvfgw"*.

B. Extension

Our extension intercepts all messages generated by the client. Unrecognized messages are blocked, and messages that contain document contents are modified to use the ciphertext document. Figure 2 shows pseudocode for intercepting relevant requests and replacing their contents with encrypted data. The enc_scheme object provides three public interfaces: encrypt, decrypt, and transform_delta. It also maintains a copy of the state of the ciphertext document which is needed to transform the delta. Section V describes the encryption scheme.

C. Using the Extension

Our tool is available at mightBeEvil.com as a Firefox extension. To use the extension, a user first installs the extension and activates it (by clicking an item in "Tools" menu). Then, the user goes to docs.google.com and uses its existing interface to "Create new Document". The extension intercepts this request and prompts the user to set a password. The newly created document is now an encrypted document. The user can edit it normally using Google Documents, but the server only sees

```
Mediator.prototype = {
 onModifyRequest : function (oHttp) {
  ... // error checking elided
  var clientString = this.getPostData(oHttp).body;
  if (clientString.match('docContents=')) { // full update
   var content = clientString.match(/docContents=([^&]*)&/)[1];
   var ctxt = Base32.encode(enc_scheme.encrypt
                            (decodeURIComponent(content)));
   this.sendRequest(oHttp,
       clientString.replace
         (content, encodeURIComponent(ctxt)))
  } else if (clientString.match('delta=')) { // incremental update
   var p_delta = clientString.match(/delta=(.*?)&/)[1];
   var c_delta = enc_scheme.transform_delta
                  (decodeURIComponent(unescape(p_delta)));
   clientString = clientString.replace
       (original_delta, encodeURIComponent(deltaContent));
   this.sendRequest(oHttp, clientString);
  } else { this.dropRequest(oHttp); } // drop all unknown requests
  ...
```

Fig. 2. Request Mediation Code Sketch

the encrypted contents of the document. When a document is loaded, our extension prompts the user with a dialog asking for various encryption parameters (e.g., password and schemes). If the document is an encrypted document, it appears as ciphertext unless the user enters the correct password.

Users can share an encrypted document by sharing the document in Google Docs, and then sharing the password using some other secure channel. This is, admittedly, not the most secure way to establish a shared secure document, but better ways of establishing a shared key outside the scope of this work.

V. Encryption

To prevent exposing the document contents to the application provider, the version of the document stored by the provider is encrypted with a key known only to the client. Our extension supports *insert* and *delete* update operations. To save both CPU

time and network bandwidth, we use incremental encryption to produce updated ciphertext.

A. Incremental Encryption

An incremental encryption scheme is a 4-tuple:

$$(\mathcal{K}, \mathsf{Enc}, \mathsf{Dec}, \mathsf{IncE})$$

where $\mathcal{K}$, Enc, Dec conform respectively to the key generation function, encryption function, and decryption function as defined in traditional encryption schemes [19]. The fourth element IncE is the incremental encryption function that takes an edit operation *Op* and optionally the previous plaintext M and ciphertext C, and computes an updated ciphertext.

Bellare, Goldreich, and Goldwasser introduced incremental symmetric encryption and gave a specific case study for incremental hashing and signing [2]. Incremental encryption schemes generally support three types of updates: *replace*, *insert*, and *delete*. Early research efforts focused mainly on inventing incremental MAC schemes restricted to the easier *replace* updates [2, 4]; later schemes also supported *insert* and *delete* updates [3, 6, 15]. In our scenario, we need all three types of update.

An incremental encryption scheme is called *ideal* if its IncE function has running time that is independent of $|M|$ and $|C|$. Nevertheless, in view of Enc's time complexity linear to the length of the plaintext, it may still be desirable to have an non-*ideal* but sub-linear time (in terms of $|M|$ and $|C|$) IncE function.

Another property of incremental encryption schemes for signing is whether or not they prevent *substitution attacks*. A substitution attack is possible because in order for an incremental signing algorithm to be *ideal*, it does not have adequate time to verify the integrity of the supplied (*message, signature*) pair. For instance, the hash-then-sign [2] and XOR [3] schemes are all subject to *substitution attacks*. On the other hand, IncXMACC [15] and the hash tree [3] schemes achieve true tamperproofing but at the cost of $O(n)$ size of signature, and $O(\log(n))$ time complexity, where both n refers to the length of the entire document. Fischlin gave a proof that for an single block accessing, incremental signing scheme supporting *replace* update to prevent *substitution attack*, the signature size is $\Omega(n)$ [15]. In our application, we are more interested in confidentiality-only and confidentiality-and-integrity services, than integrity-only services. The good news is that integrity can be obtained at marginal cost if it is added onto a confidentiality-only services.

B. Transforming Updates

For Google Documents, we see a delta in the client's outgoing messages, which describes the edits that transform a document from its previous version to the latest one. Therefore, what we need is a translation function that maps delta into cdelta, a corresponding delta value for the ciphertext document.

As depicted in Figure 1, the ciphertext document is maintained by the application provider; the plaintext document is what the client sees and edits. The extension mediates all client-server traffic, encrypting the document contents and updates as necessary for the server to maintain the ciphertext document.

We support different encryption schemes for confidentiality-only and for both integrity and confidentiality. For confidentiality-only scenarios, we use the randomized ECB (rECB) encryption mode [6]. Let the original document be $(d_1, d_2, \cdots, d_n)$, where d_i are characters of the document. With rECB mode of encryption, the ciphertext blocks are:

$$F_{sk}(r_0), F_{sk}(r_0 \oplus r_1, r_1 \oplus d_1), F_{sk}(r_0 \oplus r_2, r_2 \oplus d_2),$$
$$\ldots, F_{sk}(r_0 \oplus r_n, r_n \oplus d_n)$$

where r_i's are nonces and F_{sk} is a secure block-cipher with key sk. We `xor` each plaintext block with a 64-bit random nonce and encrypt the result with AES. To decrypt the block d_k, it suffices to decrypt the first ciphertext block (to get r_0), the $2k+1^{th}$ (to get r_k), and the $2k+2^{th}$ cipher block (to get d_k).

For scenarios that require integrity, we use the RPC mode [6] with a security amendment to thwart some forgery attacks [35]. This uses slightly more, but constant, extra resources compared to rECB scheme. The main idea of RPC mode is to use random nonces to chain together neighboring plaintext blocks before applying a block cipher like AES. The amendment suggests always include the length of the document in the last ciphertext block. Using RPC mode, the ciphertext is:

$$F_{sk}(r_0, \alpha, r_1), F_{sk}(r_1, d_1, r_2), F_{sk}(r_2, d_2, r_3),$$
$$\ldots, F_{sk}(r_n, d_n, r_0), F_{sk}(\oplus_{i=0}^{n} r_i, \oplus_{i=1}^{n} d_i, \oplus_{i=1}^{n} r_i)$$

where α is an arbitrary symbol used to mark the start, r_i's are random nonces, d_i's concatenate to form the plaintext document.

C. Multiple-Character Blocks

With a one-character per block rECB (or RPC) scheme, every 8-bit character is encapsulated in a 128-bit AES block, so the ciphertext blow-up is $16x$. Since Google currently enforces a maximum file size of 500 kilobytes, this blow-up greatly limits the size of documents, as well as increasing the bandwidth required. To reduce the ciphertext blow-up, we combine multiple characters into a single encryption block. This slightly increases the time overhead for incremental encryption.

Our design groups plaintext into blocks of up to b characters, where b is a user-adjustable parameter. The main challenge is to manage the blocks so that asymptotically fast insert/delete by index operations can be provided. No data-structure exists that can simultaneously provide both constant time updates and indexing. A straightforward approach would require re-aligning and re-encrypting all subsequent blocks when a single character is inserted or deleted. This would require re-encrypting the entire document for each edit, rendering incremental encryption useless.

We developed a data structure, called an IndexedSkipList, to cope with this dilemma. It is based on the SkipList data structure [30]. The idea of *indexing* could also be applied to any of the well-known balanced tree data structures (e. g., AVL tree, 2-3 tree, etc.) to develop a similar non-probabilistic data structure.

A SkipList stores a list of elements in order and associates a pole of probabilistic height with each element. Searching over a

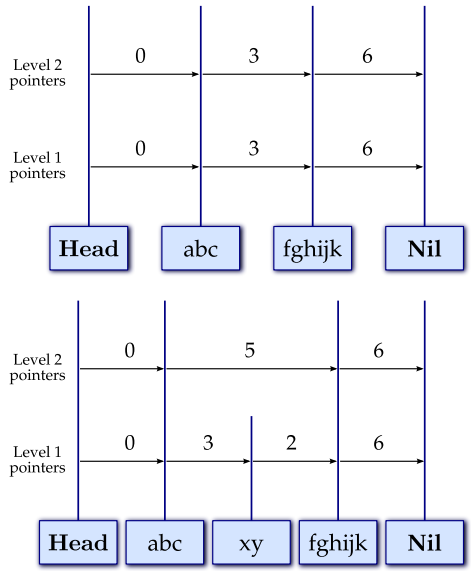


Fig. 3. SkipIndexList Insertion (insert "xy" at index 3 of "abcfghijk")

SkipList starts from the highest poles, descending down the poles as the position of searched element is known more precisely.

An IndexedSkipList insertion is illustrated in Figure 3. In our implementation, a skip_count field is associated with every list pointer at all levels (shown as numbers over the pointers in the figure).

The algorithm used to implement the IndexedSkipList Find operation is shown in Algorithm 1. It is similar to that for a normal SkipList except for the four highlighted lines. Line 3 shows the searching is based on index instead of a search key. At lines 4 and 7, the index is adjusted using the skip_count, and in the end the indexed value is returned immediately, unlike in a SkipList which checks if the input value is found. The Insert and Delete algorithms are adapted similarly. Following the analysis of the original SkipList algorithms [30], the Find, Insert, and Delete operations on an IndexedSkipList also have running times that are logarithmic (on average) in the number of blocks.

The IndexedSkipList data structure supports blocks of arbitrary size. However, if the block size is not statically fixed, we have to store the block character counters so that we remember block boundaries. Due to the fixed block size of AES, we choose a maximum of 8 characters (64 bits) per block for our incremental encryption scheme. With a block cipher of different width, other block sizes might be desirable.

Algorithm 1 Find(list, index)

```
1:  x ← list.head;
2:  for i := list.level downto 1 do
3:      while index > x.forward[i].skip_count do
4:          index ← index − x.forward[i].skip_count;
5:          x ← x.forward[i].point_at;
6:      end while
7:      index ← index − x.forward[i].skip_count;
8:      x ← x.forward[i].point_at;
9:      return x.value[index];
10: end for
```

VI. Security Analysis

We first discuss the security properties of our extension tool against an adversary who does not have malicious control of the client application (the *benign client* threat model introduced in Section II. Then, we consider a stronger threat model in which the client program itself could be malicious.

A. Benign Client Application

We assume AES is secure and that there is a good source of cryptographic random numbers. We also assume the encryption key is always kept secure on client computers. The security proof of the basic incremental encryption scheme is given by Buonanno et al. [6]. We consider all four categories (ciphertext-only, known plaintext, chosen-plaintext, chosen-ciphertext) of attacks.

In a ciphertext-only attack, the server is only allowed to infer information from its observation of received contents. Since each block is padded with a nonce before encryption, the scheme is secure against a ciphertext-only attack under the assumption that the AES cipher is secure. Even if the server knows some plaintext-ciphertext pairs, or can obtain any desired plaintext-ciphertext pairs as in known plaintext and chosen-plaintext attacks, it doesn't help in understanding unknown ciphertext because of the random padding. A chosen-ciphertext attack cannot be successful if the privacy-and-integrity encryption scheme is used, because invalid decryption requests will be distinguished and declined, making the attack equivalent to a chosen-plaintext attack.

The nonce length n serves as an important parameter that affects the security of the encryption schemes. We set the n to 64 bits. If the attacker does a brute-force search to find r_0 and the secret key, they can decrypt the whole document. So, assuming AES is secure, attacker needs to search 2^{64} possible r_0 values with 2^{128} possible keys.

In addition to passively guessing information from ciphertext, the server could launch active attacks such as replicating part of the ciphertext, threatening content integrity. Our privacy-only encryption scheme cannot withstand these attacks, but the privacy-and-integrity scheme does. This type of active modification attack is thwarted by circular-chaining the plaintext blocks together with nonces. As long as the AES cipher is secure, the adversary cannot guess the chaining nonce and any modification will be detected.

Information Leaks. Our schemes reveal positional and timing information about edits. For optimizing performance, the client sends encrypted deltas, which contain plaintext information on the edit operation type and position. These updates necessarily reveal information to the server, that could be used to infer aspects of the document. However, in most cases these edits will not reveal significant confidential information. The Google Documents client sends periodic updates at regular intervals, so does not leak precise timing information on edits. In addition, when using multi-character blocks, the precise information about update positions is no longer revealed.

Availability. The server could also destroy user-submitted content, so that documents cannot be retrieved appropriately. We do not address this type of denial of service attack, because we assume that the server provides a reliable storage service hence this attack already indicates disruption of ordinary services. We assume the server stores user-submitted content literally. This is reasonable since even though the server could process and store the content in any format it defines, for a subset of all possible content (e.g., the hexadecimal codes of ciphertext), the client has to be able to render exactly what the user submitted in order for the editor to work correctly.

The server could recognize the use of encryption and refuse to store any content that appears to be encrypted. To cope with this situation, our tool could be extended using existing results in stenography to make it difficult for the server identify encrypted documents. We have not yet investigated this possibility, though, and it may be impractical for realistic applications.

A web application could also thwart our approach by obfuscating its communication protocols to make reverse engineering more difficult. Users seeking privacy, however, would be disinclined to use a service that intentionally obfuscates its protocols to thwart privacy extensions.

B. Malicious Client Application

The benign client model assumes the client application software is trusted by the user. Since this software typically is provided by the application provider, though, the application provider could maliciously construct the client software to leak information about the user's documents back to the server.

Overt channels are blocked by the extension directly. Only messages that match a specified narrow interface are permitted to be sent back to the server, and all variable parts of those messages are encrypted content. This leaves the possibility of covert channels. It is nearly impossible to completely eliminate covert channels in such a scenario, but can take countermeasures to limit the bandwidth of undetectable covert channels.

One possible covert channel is the update messages. The length of the encrypted update is a function of the plaintext update (now controlled by the adversary), so could be used as a covert channel. The timing of the update messages could also be used as a covert channel. To disrupt these covert channels, we could add random delays (without noticeably disrupting the user experience since the updates are asynchronous) to every outgoing update request and could randomly pad the content (without affecting the correctness of the content) before encryption.

A more challenging covert channel is the delta values themselves. Although these values are encrypted, the encrypted values still contain information that could be used as a covert channel. For example, many different sequences of delta commands could produce the same editing outcome, so the malicious client could select different sequences to encode additional information. As an extreme example, consider a client that when the user enters character q sends a sequence of $Ord(q)$ (where $Ord(q)$ is the ordinal of q in the alphabet) single-character inserts followed by $Ord(q)$ deletes followed by the actual insert. From the number of consecutive deletes in the message, the server knows the character that was entered. This could be mitigated by maintaining each group of delta updates and merging them into a canonical form before sending an update to the server, or by using trusted code to compute the delta values from the two versions of the document directly instead of using the delta values computed by the provided client.

Finally, the document itself could be used as a covert channel. Since the server only sees the encrypted document, the only aspect of the document that is visible to the server is its length, which is roughly preserved by the encryption. A malicious client can add invisible content to the document (for example, formatting codes) to modify the actual document length. The updates would necessarily reveal the length of the encrypted document, since that is maintained by the server. This could be used to transmit a few bits of information with each edit. The extension can mitigate this by making the length of the encrypted document a less predictable function of the length of the plaintext document.

Discovering and disrupting covert channels is a seemingly unending arms race. In our situation, we have the advantage that the interface between the client and server can be arbitrarily narrow and is controlled by the extension, so the maximum bandwidth available for covert channels is limited. To thwart such powerful attacks completely, however, the user probably needs to use a client provided by a trusted third-party.

VII. Results and Evaluation

Our proof-of-concept tool is available as a Firefox extension. It preserves most of the functionality of Google Documents with low performance overhead. Our implementation uses a fast JavaScript implementation of AES developed by Stanford [33].

A. Functionality

Because the Google Documents server now only has access to an encrypted document, some features now become unavailable: (1) translation; (2) spell checking; (3) drawing pictures (the client sends drawing primitives to the server, where pictures are generated and sent to the client); and, (4) exporting to a document format. It is not surprising that these features (at least except for drawing pictures) require the server to manipulate the plaintext document since they depend on a large amount of data maintained by the server. Other core features such as various content formatting tools and the word counting tools work fine with our extension since they operate on the client side.

Collaborative editing is partially functional in that every passive reader gets automatic content refreshing. Simultaneous editing by different parties leads to client's complaints of multiple people editing the same region of the document. This is due to the fact that the extension does not update the contentFromServerHash value sent by the server (based on the encrypted document) to a value that is correct for the plaintext document. The SPORC project [14] investigated the problem of collaborative editing using untrusted server and developed a solution. The main difference from our work is that they assumed control over the server and designed a new server with well-defined limited assumptions, whereas we aimed to add privacy to existing commercial services over which we have no control.

B. Micro-Benchmarks

We created micro-benchmarks to measure the performance of cryptographic operations inside a browser. The benchmark aims to model user editing operations using probabilistically generated test cases. Each test case is a pair of strings (D, D'). The strings D and D' are chosen randomly with length uniformly distributed between 100 and 10000. For every (D, D') pair, a delta string is derived such that it transforms D to D'. We measured the time to encrypt D, the time to transform delta, and the time to decrypt D'. Figure 4 shows the wall clock encryption time per character. The experiments are done on a typical desktop (Intel core 2 dual E6550 2.33GHz, 2GB Memory, and Firefox 3.0.15). The average throughput is 9.1–11.8kB of plaintext per second. The performance of confidentiality-only mode is slightly better than RPC, which provides both confidentiality and integrity.

	Average (per char)
encryption (D)	.091ms
decryption (D')	.085ms
incremental encryption	.110ms

Fig. 4. Micro-benchmark Results for RPC Mode (averages from 1000 tests).

C. Macro-Benchmarks

The macro-benchmarks model user experience better in that they account for not only the time spent by our extension for mediating the messages, but also the time spent on waiting for server's responses. A test case in the macro-benchmark is a whole document save followed by either replacing an existing sentence with a different one or inserting or deleting an arbitrary sentence or group of sentences. Using the Selenium scripting tool [31], we ran macro-benchmarks on both small ($\approx$ 500 characters) and large ($\approx$ 10000 characters) files. We executed each test case both with and without our extension enabled, and measured the latency. The results are shown in Figure 5.

The overhead required for the initial loading is significant, but this is only done once at the beginning of an editing session. For the majority of editing operations, the performance overhead is less than 10%, with a maximum overhead of 13% associated with running RPC mode encryption with relatively large files. The performance impact of cryptographic manipulations is offset by communication and server processing time. Since all editing update requests are sent asynchronously, the user can continue to edit without waiting for the server acknowledgments.

D. Multi-character Incremental Encryption

Figure 6 shows the performance of our multi-character incremental encryption scheme using a synthesized micro-benchmark similar to that of Section VII-B, except that we fixed the length of the original document to be 10000 characters. We evaluate the performance impact of block size for each type of encryption operation. The underlying encryption mode is rECB.

The cost of encryption decreases as the block size increases, as expected for all categories. There is some noise due mainly to the probabilistic nature of the SkipIndexList and the variability in the location of edits. Even though our JavaScript implementation of SkipIndexList introduces appreciable overhead for the editing operations (compared to those offered by the built-in JavaScript Array and String), as evidenced from the data for 1-char blocks, this cost is well compensated by setting the block size to 7 or above. The ciphertext blowup is reduced substantially as shown in Figure 7. The actual reduction is less than the ideal reduction due to fragmentation.

	Small ($\approx$ 500 characters) files			
	rECB		RPC	
	mean	dev.	mean	dev.
initial load	25.0%	.044	24.0%	.065
inserts only	6.2%	.049	7.0%	.040
deletes only	3.1%	.012	4.5%	.019
inserts & deletes	7.4%	.059	9.0%	.053
	Large ($\approx$ 10000 characters) files			
	rECB		RPC	
	mean	dev.	mean	dev.
initial load	43.0%	.051	45.0%	.085
inserts only	8.2%	.050	10.0%	.047
deletes only	3.9%	.014	4.3%	.023
inserts & deletes	11.0%	.059	13.0%	.060

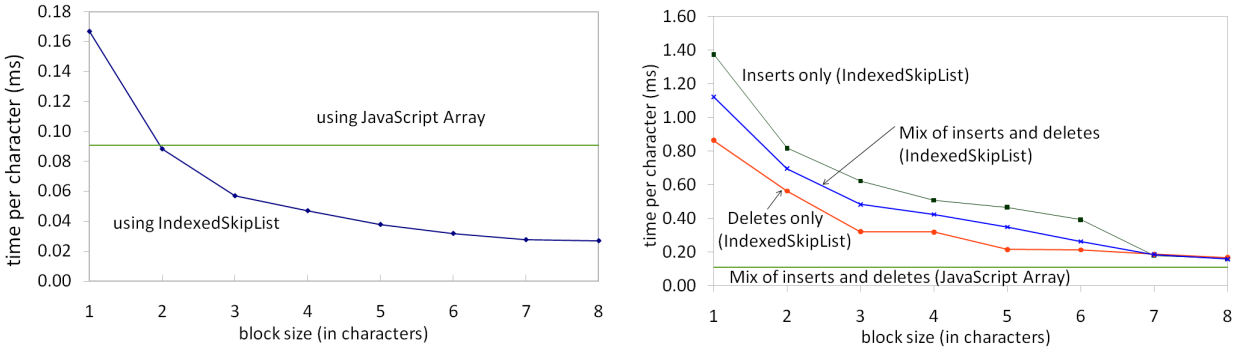
Fig. 5. Macro-benchmark Results (performance degradation).

Figure 8 shows the performance results of running the macro-benchmark using the 8-character per block rECB incremental encryption scheme. Compared to Figure 5, the performance overhead increases slightly, but the ciphertext blowup is reduced from $23x$ to less than $5x$.

VIII. Related Work

We covered related work on secure file systems and the CoClo project in the introduction, and on incremental encryption in Section V. Here, we briefly survey other related work on cloud computing security and computing on encrypted data. These works have similar goals to ours in enabling computing services to be used without exposing confidential data, but differ from our work in the tradeoffs they make between performance and generality and the amount of cooperation they need from the service provider.

Hsu and Chen [17] proposed a cloud service paradigm in which the file storage systems are separated from web applications to solve the problem in general. This only mitigates the third-party data control threat since users still have to trust at least one of the secure file system service providers, and any web application still gets access to users' plaintext data once granted. Wang et al. [34] considered the problem of allowing a third party to provide a user data auditing service without compromising the privacy of user contents. They also investigated the problem of using a trusted third party to efficiently audit dynamically changing user data [36]. Erway et al. [13] devised protocols that enable a cloud server to efficiently generate proofs that it possesses an integral version of dynamically updating user data. Similar to our work,



(a) Encrypting Whole Documents

(b) Incremental Updates

Fig. 6. Impact of Block Size

block size	1	2	3	4	5	6	7	8
blowup	21.00	10.71	7.35	6.09	4.83	4.41	3.78	3.75
reduction	0%	49%	65%	71%	77%	79%	82%	82%

Fig. 7. Ciphertext Blowup Reduction

file size ≈10000 characters	rECB	
	mean	dev.
initial load	18%	.047
inserts only	8.8%	.058
deletes only	7.5%	.034
inserts and deletes	12.6%	.082

Fig. 8. Macro-benchmark results of Multi-character Incremental Encryption (performance degradation).

one of their protocols was also built on a variant of the skip list data structure.

The FireGPG [11] browser add-on provides a convenient user interface for applying GnuPG to web content in the browser. It offers an integrated interface to GMail cloud service to allow using GPG's function directly in webmail. However, it provides no support for incremental updates.

Many efforts aim to resolve data privacy and integrity issues merged with web services [8, 9, 12, 18, 37]. SIF presented an approach to use language-based information flow analysis and automatic program partition technique to provide data confidentiality and integrity for the server against malicious clients [8]. Xu et al. present a formal framework for specifying data privacy policies and usage models in a composite web service involving multiple participants [37]. They also assume a trusted service provider. Christodorescu depicted a blueprint for inserting a cryptographic layer between display-related and network-related components and suggested a list of open research problems on utilizing *untrusted* servers [9]. iDataGuard [18] gives a middleware level solution to outsource user data files to untrusted internet data services. Compared to our work, these works assumed availability of each service's APIs, and focused on tackling services' heterogeneity issues.

IX. Conclusion

Though on-line document editing and management web applications offered as cloud services have many advantages, they require users to trust the provider with the contents of their documents. The goal of this work is to enable users to use on-line document editing services without needing to trust the application provider with their confidential data.

Our approach requires reverse engineering the application-level communication protocol used for document updates and modifying protocol messages to encrypt all user data. We implemented a proof-of-concept Firefox browser extension tool that demonstrates our idea for Google Documents, Mozilla Bespin, and Adobe Buzzword, and developed an incremental encryption scheme to support incremental updates in Google Documents. The experiments show the user experience degradation incurred by the extension is minimal, and the delay in response time is reasonably low.

Although much work has been done on heavyweight approaches to support computation on encrypted data, our work shows that lightweight techniques may be sufficient for many interesting applications where most of the data-dependent computation can be done by the client. Such techniques cannot provide the highest level of privacy, especially against a malicious adversary with control over the client application, but provide a useful point in the design space for enhancing privacy with minimal cost or disruption to functionality.

Acknowledgments

This work was partly supported by grants from the National Science Foundation and a MURI award from the Air Force Office of Scientific Research. The authors thank Emily Stark, Mike Hamburg, and Dan Boneh for making their JavaScript fast encryption library available, as well as Einar Lielmanis et al. for creating the `jsbeautifier` tool and their timely responses to our bug reports. We also thank the anonymous reviewers for their thorough and helpful reviews, and abhi shelat for insightful comments on this work.

References

[1] A. Barkah. Security Issues with Google Docs. http://peekay.org/2009/03/26/security-issues-with-google-docs/.

[2] M. Bellare, O. Goldreich, and S. Goldwasser. Incremental Cryptography: The Case of Hashing and Signing. In *CRYPTO*, 1994.

[3] M. Bellare, O. Goldreich, and S. Goldwasser. Incremental Cryptography and Application to Virus Protection. In *Symposium on Theory of Computing*, 1995.

[4] M. Bellare and D. Micciancio. A New Paradigm for Collision-Free Hashing: Incrementality at Reduced Cost. In *EUROCRYPT*, 1997.

[5] M. Blaze. A Cryptographic File System for UNIX. In *ACM Conference on Computer and Communications Security*, 1993.

[6] E. Buonanno, J. Katz, and M. Yung. Incremental Unforgeable Encryption. In *International Workshop on Fast Software Encryption*, 2002.

[7] J. Caballero, P. Poosankam, C. Kreibich, and D. Song. Dispatcher: Enabling Active Botnet Infiltration Using Automatic Protocol Reverse-engineering. In *ACM Conference on Computer and Communications Security*, 2009.

[8] S. Chong, K. Vikram, and A. C. Myers. SIF: Enforcing Confidentiality and Integrity in Web Applications. In *USENIX Security Symposium*, 2007.

[9] M. Christodorescu. Private Use of Untrusted Web Servers via Opportunistic Encryption. In *Web 2.0 Security and Privacy*, 2008.

[10] W. Cui, J. Kannan, and H. J. Wang. Discoverer: Automatic Protocol Reverse Engineering from Network Traces. In *USENIX Security Symposium*, 2007.

[11] M. Cuony. FireGPG Home Page. http://getfiregpg.org/.

[12] G. D'Angelo, F. Vitali, and S. Zacchiroli. Content Cloaking: Preserving Privacy with Google Docs and Other Web Applications. In *ACM Symposium on Applied Computing*, 2010.

[13] C. Erway, A. Küpçü, C. Papamanthou, and R. Tamassia. Dynamic Provable Data Possession. In *ACM Conference on Computer and Communications Security*, 2009.

[14] A. J. Feldman, W. P. Zeller, M. J. Freedman, and E. W. Felten. SPORC: Group Collaboration Using Untrusted Cloud Resources. In *USENIX Symposium on Operating Systems Design and Implementation*, 2010.

[15] M. Fischlin. Incremental Cryptography and Memory Checkers. In *EUROCRYPT*, 1997.

[16] Google Inc. Google Docs. http://docs.google.com/ (launched in 2006).

[17] F. Hsu and H. Chen. Secure File System Services for Web 2.0 Applications. In *ACM Workshop on Cloud Computing Security*, 2009.

[18] R. C. Jammalamadaka, R. Gamboni, S. Mehrotra, K. E. Seamons, and N. Venkatasubramanian. iDataGuard: Middleware Providing a Secure Network Drive Interface to Untrusted Internet Data Storage. In *International Conference on Extending Database Technology*, 2008.

[19] J. Katz and Y. Lindell. *Introduction to Modern Cryptography*. Chapman & Hall/CRC, 2007.

[20] J. Kincaid. Google Privacy Blunder Shares Your Docs Without Permission. *TechCrunch*, March 2009.

[21] A. Lieuallen. Greasemonkey API Usage. http://www.greasespot.net/.

[22] Z. Lin, X. Jiang, D. Xu, and X. Zhang. Automatic Protocol Format Reverse Engineering Through Context-Aware Monitored Execution. In *Network and Distributed System Security Symposium*, 2008.

[23] D. Mazières, M. Kaminsky, M. F. Kaashoek, and E. Witchel. Separating Key Management from File System Security. *SIGOPS Operating Systems Review*, pages 124–139, 1999.

[24] Microsoft Corporation. Microsoft Office Live. http://www.officelive.com/ (launched in 2008).

[25] Microsoft Corporation. Encrypting File System for Windows 2000, 1999.

[26] E. Miller, D. Long, W. Freeman, and B. Reed. Strong Security for Distributed File Systems. In *IEEE International Performance, Computing, and Communications Conference*, 2002.

[27] Mozilla Labs. Bespin — Code in the Cloud. https://bespin.mozilla.com/ (launched in 2009).

[28] Mozilla Labs. Bespin/ServerAPI. https://wiki.mozilla.org/Labs/Bespin/ServerAPI (launched in 2009).

[29] Opera Software. Take Control with User JavaScript. http://www.opera.com/browser/tutorials/userjs/.

[30] W. Pugh. Skip Lists: A Probabilistic Alternative to Balanced Trees. *Communications of the ACM*, 33(6):668–676, 1990.

[31] Selenium Contributors. Selenium Web Application Testing System. http://seleniumhq.org/ (launched in 2008).

[32] Squid Project. Squid: Optimising Web Delivery. http://www.squid-cache.org/.

[33] E. Stark, M. Hamburg, and D. Boneh. Symmetric Cryptography in Javascript. In *Annual Computer Security Applications Conference*, 2009.

[34] C. Wang, Q. Wang, K. Ren, and W. Lou. Privacy-Preserving Public Auditing for Data Storage Security in Cloud Computing. In *INFOCOM*, 2010.

[35] C. C. Wang, M.-C. Kao, and Y.-S. Yeh. Forgery Attack on the RPC Incremental Unforgeable Encryption Scheme. In *ACM Symposium on Information, Computer and Communications Security*, 2006.

[36] Q. Wang, C. Wang, J. Li, K. Ren, and W. Lou. Enabling Public Verifiability and Data Dynamics for Storage Security in Cloud Computing. In *European Symposium on Research in Computer Security*, 2009.

[37] W. Xu, R. Sekar, I. V. Ramakrishnan, and V. N. Venkatakrishnan. An Approach for Realizing Privacy-preserving Web-based Services. In *International Conference on World Wide Web*, 2005.