Collaboration-Preserving Authenticated Encryption for Operational Transformation Systems [⋆]

Michael Clear, Karl Reid, Desmond Ennis, Arthur Hughes, and Hitesh Tewari

School of Computer Science and Statistics,
Trinity College Dublin

Abstract. We present a flexible approach for achieving user-controlled privacy and integrity of documents that are collaboratively authored within web-based document-editing applications. In this setting, the goal is to provide security without modifying the web application's client-side or server-side components. Instead, communication between both components is transparently intercepted and processed (if necessary) by means of a local proxy or browser plugin. We improve upon existing solutions by securely preserving real-time collaboration for encrypted documents and facilitating self-containment of the metadata (an overhead of encryption) within the same document. An architectural generalization is also presented that permits generic transformations and fine-grained access control. Security is assessed with respect to several threat models, and performance is evaluated alongside other approaches.

1 Introduction

Privacy for data stored in the cloud is a growing concern. Although the ubiquity of TLS for securing the transport layer ensures confidentiality and integrity for data in-transit, reservations arise about the security of data persisted in the cloud. This is especially pronounced for Software as a Service (SaaS) offerings whereby an organization's security policy is incompatible with that adopted by the service provider. Since it is desirable to avail of the convenient functionality and resources supported by the provider, these concerns have often been incautiously ignored.

In recent years, there has been significant research in the area of computing delegation and data delegation as cloud computing has emerged as a predominant paradigm. Advances in fully homomorphic encryption since Gentry's 2009 breakthrough [2] such as [3], along with practical predicate encryption such as [4] proceed towards the realizing the goal of privacy in outsourced computations. However, in these settings, control is assumed over the remote server.

Some SaaS applications can retain their functionality while simultaneously allowing end-to-end data privacy, *without* performing computations homomorphically or relying on cryptographic techniques that require changes to be made to the server. Other applications can retain the bulk of their features under similar conditions. In fact, if control were assumed over customizing the server-side software to accommodate end-to-end data privacy and integrity, the task would be less challenging. By 'end-to-end', we mean that the client has exclusive access to any keys required to authenticate or decrypt the application data.

Consider a SaaS-orientated solution comprising two components - a user-facing client and a remote server. For several web applications such as Google Docs [5], there is a tight-coupling between both. Indeed, the client is not guaranteed to be static, and there may or may not be a published API between client and server. In the absence of a published API, the interface i.e. *protocol* must be reverse engineered.

There are several web-based real-time collaborative-editing systems based on the technique of Operational Transformation (OT). In essence, OT manages consistency of shared resources that are acted on concurrently by multiple collaborators. As collaborators relay their edit *operations* to each other (usually via a coordinator), these edits must be correctly *transformed* relative to other concurrent edits to preserve the consistency of the shared resource. Web-based OT applications include Google Docs [5], Etherpad [6] and Apache Wave [7].

We focus on Google Docs in this work for a multitude of reasons. Firstly, it is popular and free, allowing us to readily obtain empirical data from co-operative users. Secondly, alternative approaches for achieving

[⋆]This is the full version of a paper that appeared in ISC 2012 [1].

encryption have been explored in the literature, (most notably [8, 9]), thus allowing a comparative analysis to be conducted.

Achieving security for services that are operated by untrusted parties, and doing so without control over the server is a difficult challenge. Availing of security functions such as encryption in this scenario often disrupts core functionality of the service. For example in the case of collaborative editing, features such as spell checking and searching that are performed on the server-side are spoiled. Nevertheless, it may be satisfactory for users that the predominant functionality of such applications, namely easy sharing of documents, real-time collaborative editing and granular version control, remain unimpaired by added security functions. In addition, many users may only require encryption for a small portion of the document, for example, sensitive fields such as social security numbers.

The contributions of this paper are as follows:

1. A flexible approach is presented to achieve transparent application-layer security (integrity and authentication) for untrusted OT systems such as Google Docs. Unlike prior work, our solution does not use an out-of-band resource to store auxiliary information nor does it suffer from synchronization issues regarding its consistency. Instead, the presented approach inextricably couples the auxiliary information into the document representation in a manner that allows both efficient *incremental* encryption and preservation of collaborative functionality.
2. A model is formulated that expresses this approach generically for a certain class of OT systems. As a result, the precise preconditions and expected behavioral properties can be established.
3. A proof-of-concept implementation for the Firefox web browser is discussed and is shown to compare favorably against existing implementations. Note that the complex issue of key sharing is not addressed in this particular work. [⋆]

1.1 Related Work

The target problem exhibits overlap with several areas; in particular incremental cryptography [10] and secure timestamp systems [11].

This work was carried out independent to the contributions of Feldman et al. [12] and thus there are several similarities, although both works have a different focus. The major difference is that we concentrate on providing security for *existing, live* OT systems such as Google Docs whereas Feldman et al. devise a new OT-based framework to be deployed on an untrusted server. Furthermore, they consider protection against a variety of active attacks, such as forking, which we don't address in this paper.

Adkinson-Orellana et al. [9] present a security layer to provide encryption for Google Docs by means of a Firefox extension that mediates traffic to/from the Google Docs server in order to transparently encrypt/decrypt it. Since semantically secure encryption depends on a random component, it is necessary to include additional information with a ciphertext. The authors of [9] propose to employ a hidden document serving as an index table where this information can be stored. This approach introduces a host of synchronization issues - collaborating users must somehow synchronize updates to the shared index in order to maintain consistency. Furthermore, the auxiliary information is externalized; that is, stored out-of-band.

An improved solution appears in a more recent work by Huang and Evans [8]. The authors also develop a Firefox extension to transparently mediate Google Docs traffic. Their solution is based on incremental encryption, introduced by Bellare, Goldreich and Goldwasser [10]. In particular, they use a block cipher mode of operation known as RPC [13] for incremental *authenticated* encryption. A derivative of SkipList data structure is employed to accomodate efficient incremental encryption, and it supports complete document integrity. Its main drawback is synchronization limitations. Synchronization is difficult because it relies on a block table (the chaining of all the blocks together), which may give rise to conflicts if multiple users are editing simultaneously, although the authors cite other technical reasons. In the version of Google Docs targeted in this work, the size of a delta i.e. the number of characters modified in a single edit must be preserved by the browser extension in order to maintain consistency, certainly in the presence of active

[⋆]Nevertheless, key sharing has been incorporated into our proof-of-concept implementation by intercepting the requests that specifically relate to document sharing, and then enveloping the symmetric key using the target recipient's public key and depositing the envelope in a remote centralized data store. PKI is employed in the present proof-of-concept implementation, and ECC is used for efficiency.

collaborators. As a result, the approach taken by Huang and Evans where a block table is stored in the document content is rendered infeasible by our goal to support fuller collaboration. This reason along with our aim to avoid storing auxiliary data "out-of-band" motivated the revision-based approach adopted here.

1.2 Notation

In this paper, we index strings, sequences and vectors starting from 0. Given a string w in Σ^* for some alphabet Σ, the j-th character is denoted by $w[j]$, and a substring of w with starting index i and ending index (exclusive) j is denoted by $w[i, j]$. The symbol $\|$ denotes string concatenation. As a notational convenience, the length of all strings and sequences is given by the $|\cdot|$ operator. We denote the empty string by ϵ.

For a probability distribution $\mathcal{D}_S$ defined on a set S, we denote by $x \overset{\$}{\leftarrow} \mathcal{D}_S$ the fact that x is sampled from S according to $\mathcal{D}_S$. For any set T, the notation $t \overset{\$}{\leftarrow} T$ indicates that t is uniformly sampled from T.

All vectors and sequences used throughout this paper are denoted in boldface. The set $\mathbb{N}_0$ is considered to be the set of natural numbers with inclusion of 0.

A function is said to be *negligible* (in some parameter n) if for all polynomial functions $g(x)$: $f(n) < 1/g(n)$ for sufficiently large n.

1.3 Paper Organization

This paper is organized as follows: firstly in Section 2, an abstraction of an Operational Transformation system is formulated which models a variety of real-world collaborative editing systems. Our methodology for collaboration-preserving encryption and authentication of document content is then expressed within this model. A proof-of-concept implementation then follows for Google Docs in Section 3 where we instantiate our approach with cryptographic primitives. Performance is evaluated in Section 4. Finally, security properties are analyzed in Section 5.

2 Abstract Collaborative Editing System

An abstraction of a simplified collaborative editing system is presented in this section. The model described here will be referred to as the Simple Transactional Collaborative Editing (STCE) model. We intentionally omit the operational transformation aspects relating to consistency and concurrency control, which have been the subject of considerable research so far [14–16], and are outside the scope of this work. Accordingly, these properties are taken as assumptions.

The system is coordinated by a centralized server, denoted by C, referred to as the *coordinator*. The abstraction is simplified to center around a fixed document resource, which therefore obviates the need to reference a particular document. For the document resource implicit in the description, we assume that there are n connected collaborating *sites* labelled $s_1, \ldots, s_n$. The interface exposed to each site by C is outlined in Figure 1.

Let Σ be an alphabet. A document's content is typed as a string over this alphabet - an element of Σ^*. Each site s_i maintains a local state q_i, which is a pair (c, d), where $c \in \mathbb{N}_0$ is the site's cursor position within the document content and $d \in \Sigma^*$ is its local copy of the document content. Thus we define the state space $Q \triangleq \{(p, w) \in \mathbb{N}_0 \times \Sigma^* : 0 \leq p \leq |w|\}$. Upon establishing a session with C, a site initializes its cursor position to 0, and any changes to it must be synchronized with C. Therefore, C keeps track of the current cursor positions of all active collaborators facilitating control over synchronization and mutual exclusion.

2.1 Primitive Operations

There are a finite number of prescribed primitive operations that can be performed on a state. Such a primitive operation is termed a *mutation*. It is sufficient to describe two such types in this simplified system, namely for insertion and deletion. Define a set of symbolic tags $T \triangleq \{\tau_{\mathsf{ins}}, \tau_{\mathsf{del}}\}$. Now we define the set of insertion mutations $M_{\tau_{\mathsf{ins}}} \triangleq \{(\tau_{\mathsf{ins}}, v) : v \in \Sigma^*\}$ and the set of deletion mutations $M_{\tau_{\mathsf{del}}} \triangleq \{(\tau_{\mathsf{del}}, k) : k \in \mathbb{Z}\}$. Finally the total set of mutations M is the union $M_{\tau_{\mathsf{ins}}} \cup M_{\tau_{\mathsf{del}}}$.

Fig. 1. Interface to the Coordinator C

Service 1 EstablishSession

```
1: $\mathsf{siteId} \leftarrow n$
2: $n \leftarrow n + 1$
3: Initialize position for site $\mathsf{siteId}$ to 0
4: Initialize revision number for site $\mathsf{siteId}$ to $|\boldsymbol{H}|$
5: return $\mathsf{siteId}$
```

Service 2 UpdateCursorPosition

```
Input: $\mathsf{siteId} : \mathbb{N}_0$
Input: $p : N_0$
Require: $p \leq |\delta_{\boldsymbol{H}}(\epsilon)|$
1: Update cursor position for site $\mathsf{siteId}$ to $p$
```

Service 3 Edit

```
Input: $\mathsf{siteId} : \mathbb{N}_0$
Input: $\mathsf{mutations} : M^+$
Input: $\mathsf{revisionNum} : \mathbb{N}_0$
Require: $0 \leq \mathsf{siteId} < n$
Require: $\boldsymbol{H}[0, \mathsf{revisionNum}] \parallel \boldsymbol{r} \in \mathbb{H}$    ▷ where
    $\boldsymbol{r} := (\mathsf{position}_{\mathsf{siteId}}, \mathsf{mutations})$
1: $\boldsymbol{r'} \leftarrow$ result of OT processing on $\boldsymbol{r}$ with respect to
   recent revisions (since last revision committed by the
   site)
2: $\mathsf{newRevisionNum} \leftarrow |\boldsymbol{H}|$
3: $\boldsymbol{H} \leftarrow \boldsymbol{H} \parallel \boldsymbol{r'}$
4: return $\mathsf{newRevisionNum}$
```

Service 4 LoadRevisions

```
Input: $\mathsf{startRevNum} : \mathbb{N}_0$
Input: $\mathsf{endRevNum} : \mathbb{N}_0$
Require: $0 \leq \mathsf{startRevNum} \leq \mathsf{endRevNum} \leq |\boldsymbol{H}|$
1: return $\boldsymbol{H}[\mathsf{startRevNum}, \mathsf{endRevNum}]$
```

Service 5 GetHistorySize

```
1: return $|\boldsymbol{H}|$
```

Some mutations may not be compatible with a particular state. An example is deletion where the number of characters specified for deletion may exceed the range of the document string. Although removing as many characters as possible is satisfactory, an alternative that is useful later for other purposes is to augment Q with a ‘failure’ state. Thus, we define $\hat{Q} \triangleq Q \cup \{\perp\}$. A mutation description μ (i.e. a pair in M) induces a state transformation function $\delta_\mu : \hat{Q} \to \hat{Q}$. We have that

$$\delta_{(\tau_{\mathsf{ins}}, v)}(q) = \begin{cases} (p + |v|, w[0,p] \parallel v \parallel w[p, |w|]) & \text{if } q := (p, w) \in Q \\ \perp & \text{if } q = \perp \end{cases}$$

and

$$\delta_{(\tau_{\mathsf{del}}, k)}(q) = \begin{cases} (p + k, w[0, p+k] \parallel w[p, |w|]) & \text{if } q := (p, w) \in Q, k < 0 \leq p + k < |w| \\ (p, w[0,p] \parallel w[p+k, |w|]) & \text{if } q := (p, w) \in Q, 0 \leq k \leq p + k \leq |w| \\ \perp & \text{otherwise} \end{cases}$$

Informally, the former splices a new string into an existing string while the latter has the effect of removing a portion of the string before or after a specified position (it removes as many characters as possible of the specified number of characters). These transformation functions transform states, updating both cursor position and document content.

2.2 Composite Operations

A mutation is an atomic primitive operation. In order to specify an ordered sequence of mutations to be treated as an atomic unit, the system supports composite operations which behave as transactions. Any valid edit request produced by a site entails a non-empty ordered sequence of mutations termed a *revision* that is to be applied atomically. A revision also specifies the cursor position in the content string at which the mutations are to be applied. Define the set of revisions as $\mathcal{R} \triangleq \mathbb{N}_0 \times M^+$, We distinguish a revision $\boldsymbol{r} \in \mathcal{R}$ by boldface. If $\boldsymbol{r}$ is valid, the coordinator C stores $\boldsymbol{r}$ in its chronology, which is discussed further later. It also relays $\boldsymbol{r}$ to the $n - 1$ other sites and takes care of the appropriate operational transformations.

Let $\boldsymbol{r} := (p, \mu_0, \ldots, \mu_{\ell-1})$ be a revision. The transformation function $\delta_{\boldsymbol{r}} : \Sigma^* \cup \{\bot\} \to \Sigma^* \cup \{\bot\}$ induced by $\boldsymbol{r}$ is given by:

$$\delta_{\boldsymbol{r}:=(p,\mu_0,\ldots,\mu_{\ell-1})}(s) = \begin{cases} \mathsf{proj}_1((\delta_{\mu_{\ell-1}} \circ \cdots \circ \delta_{\mu_0})(p, w)) & \text{if } s := (p, w) \in Q \\ \bot & \text{otherwise} \end{cases} \tag{1}$$

where proj_1 is the projection map that returns the second (index 1) component of its vector argument.

We say two revisions $\boldsymbol{r_1}$ and $\boldsymbol{r_2}$ are equivalent, written $\boldsymbol{r_1} \sim \boldsymbol{r_2}$, iff $\delta_{\boldsymbol{r_1}} = \delta_{\boldsymbol{r_2}}$. It is easy to show that for every equivalence class in $\mathcal{R}/\sim$, there is a representative $\boldsymbol{r} \in \mathcal{R}$ of the form $(\tau_{\mathsf{del}}, k), (\tau_{\mathsf{ins}}, v))$ where $k \in \mathbb{Z}$ and $v \in \Sigma^*$. Such an $\boldsymbol{r}$ is said to be in normal form, and we make use of this in the security game in Section 2.3. This prompts the question: should a site normalize a revision prior to sending it to C? Alternatively, should C normalize the revision before recording it in its chronology? Clearly this depends on the requirements and feature set of the system. Each mutation corresponds to an editing event and these events may be bundled together periodically to form atomic revisions. Thus in order to facilitate an undo feature with arbitrary granularity, preserving the original sequence of events is necessary. Another motivation to preserve the original sequence of events in a persistent chronology is to allow for fine-grained logging. Normalization would lose any such information. Thus, in the system described here, we assume that a revision submitted by a site remains *structurally invariant* in persistent storage provided by C.

For each document, the coordinator C maintains a history $\boldsymbol{H} \subset \mathcal{R}^*$ of all revisions ordered by the time of commitment. It exposes a facility to make range queries on $\boldsymbol{H}$. We define $\delta_{\boldsymbol{H}} = \delta_{\boldsymbol{r_{m-1}}} \circ \cdots \circ \delta_{\boldsymbol{r_0}}$ where $\boldsymbol{H} = \boldsymbol{r_0}, \ldots, \boldsymbol{r_{m-1}}$. A history is a sequence of *valid* revisions i.e. reconstitution of the document string does not yield the 'failure' state. Therefore, we define the history space as $\mathbb{H} \triangleq \{\boldsymbol{H} \in \mathcal{R}^* : \delta_{\boldsymbol{H}}(\epsilon) \neq \bot\}$ where ϵ is the empty string.

2.3 Transformations

Consider useful types of pre-processing and post-processing that may be performed on revisions stored in a history. To capture the restricted transformations that can be applied to revisions in the system, we define a stateful (dependent on the history preceding a given revision) and reversible transformation as follows:

Definition 1. *Define* $\mathsf{HR} \triangleq \{(\boldsymbol{H}, \boldsymbol{r}) \in \mathbb{H} \times \mathcal{R} : \boldsymbol{H} \parallel \boldsymbol{r} \in \mathbb{H}\}$. *Define* $\hat{\mathcal{R}} \triangleq \mathcal{R} \cup \{\bot\}$. *A revision transformation is a pair* $(\mathsf{Map}, \mathsf{Invert})$, *where* $\mathsf{Map} \subseteq \mathsf{HR} \times \hat{\mathcal{R}}$ *is a relation and* $\mathsf{Invert} : \mathsf{HR} \to \hat{\mathcal{R}}$ *is a function, satisfying*
$\forall(\boldsymbol{H}, \boldsymbol{r}, \boldsymbol{r'}) \in \mathsf{Map} : \boldsymbol{r'} \in \mathcal{R}$, $\forall(\boldsymbol{I}, \boldsymbol{s}, \boldsymbol{s'}) \in (\mathbb{H} \times \mathcal{R} \times \hat{\mathcal{R}}) \setminus \mathsf{Map}$*:*

1. $\boldsymbol{H} \parallel \boldsymbol{r'} \in \mathbb{H}$.
2. $\mathsf{proj}_0(\boldsymbol{r}) = \mathsf{proj}_0(\boldsymbol{r'})$
3. $\mathsf{proj}_0 \circ \delta_{\boldsymbol{r}} = \mathsf{proj}_0 \circ \delta_{\boldsymbol{r'}}$
4. $\boldsymbol{r} \sim \mathsf{Invert}(\boldsymbol{H}, \boldsymbol{r'})$
5. $\mathsf{Invert}(\boldsymbol{I}, \boldsymbol{s'}) = \bot$

Intuitively, properties 1-3 capture the requirement that Map preserve both the validity of a revision, the cursor position it acts at and the change in cursor position it effects, whereas the final two properties constrain the transformation to be reversible. We will respectively view Map and Invert as a randomized algorithm and a deterministic algorithm. To apply such algorithms to a history $\boldsymbol{H}$, we define

$$\mathsf{InvertH}(\mathsf{Invert}, \boldsymbol{H}) = \begin{cases} \epsilon & \text{if } \boldsymbol{H} = \epsilon \\ \boldsymbol{r'} \parallel \mathsf{InvertH}(\mathsf{Invert}, \boldsymbol{H'}) & \text{if } \boldsymbol{H} = \boldsymbol{H'} \parallel \boldsymbol{r} : \boldsymbol{r} \in \mathcal{R} \text{ and } \boldsymbol{r'} \neq \bot \\ \bot & \text{otherwise} \end{cases}$$

where $\boldsymbol{r'} = \mathsf{Invert}(\boldsymbol{H'}, \boldsymbol{r})$ and

$$\mathsf{MapH}(\mathsf{Map}, \boldsymbol{H}) = \begin{cases} \epsilon & \text{if } \boldsymbol{H} = \epsilon \\ \boldsymbol{H''} \parallel \boldsymbol{r'} & \text{if } \boldsymbol{H} = \boldsymbol{H'} \parallel \boldsymbol{r} : \boldsymbol{r} \in \mathcal{R} \text{ and } \boldsymbol{r'} \neq \bot \text{ and } \boldsymbol{r'} \neq \bot \\ \bot & \text{otherwise} \end{cases}$$

where $\boldsymbol{H''} = \mathsf{MapH}(\mathsf{Map}, \boldsymbol{H'})$ and $\boldsymbol{r'} = \mathsf{Map}(\boldsymbol{H''}, \boldsymbol{r})$

Definition 1 generalizes several transformation functionalities including string isomorphisms. An important characteristic of a revision transformation is that it may rely on the structure of a revision to accomplish invertibility i.e. it may reorder mutations or add 'dummy mutations' to store auxiliary information. For example: suppose the desired transformation is information lossy, which implies that additional information must be stored to reverse it. Due to the property of structural invariance of revisions that is provided by the system, it is possible to map a revision to one with an appended insertion-deletion pair (an insertion of a string immediately followed by its deletion). Such a null sequence has no effect on the content but the information it encodes serves to inform the invertibility of the transformation. Building on the notion established by Definition 1, we can establish a family of transformations with common functionality parameterized by some key or index. Thus given the spaces K_1 and K_2, and a subset of their product $K \subseteq K_1 \times K_2$, a family of transformations $\mathcal{T}_K$ is defined as the set $\{(\mathsf{Map}_{k_1}, \mathsf{Invert}_{k_2}) : (k_1, k_2) \in K\}$.

Encryption A useful family of transformations is one whose function is to encrypt the content of a document. This is, for example, readily instantiated by a symmetric cipher. In this case, we have $K_1 = K_2$. For confidentiality only (or in addition to revision integrity as described in Section 5), the Map algorithm may be 'stateless' and act independently of a history argument. A generalized IND-CPA game for the confidentiality-only setting is sketched as follows:

Fix a security parameter λ. The challenger $\mathcal{C}$ generates some $(k_1, k_2) \in K$ from λ. The attacker $\mathcal{A}$ is granted oracle access to $\mathsf{Map}_{k_1}(\cdot)$ to make queries. $\mathcal{A}$ produces two distinct revisions $\boldsymbol{r_0}$ and $\boldsymbol{r_1}$ with the property that $\mathsf{proj}_0 \circ \delta_{\boldsymbol{r}_0} = \mathsf{proj}_0 \circ \delta_{\boldsymbol{r}_1}$, and sends them to $\mathcal{C}$. Accordingly, $\mathcal{C}$ samples a bit b uniformly at random, and returns $\boldsymbol{r_b}' = \mathsf{Map}_{k_1}(\mathsf{Normalize}(\boldsymbol{r_b}))$ to $\mathcal{A}$. The selected revision is normalized prior to application of Map_{k_1} to prevent trivial distinguishability. Finally, $\mathcal{A}$ outputs a bit b', and as usual, it is said to win the game if $b = b'$. As is standard, if the advantage of any polynomially-bounded $\mathcal{A}$ is negligible in λ, the system is IND-CPA-secure. Of course it is easy to extend this game to capture the IND-CCA1 and IND-CCA2 security properties by exposing an oracle to $\mathsf{Invert}_{k_2}(\cdot)$ with appropriate restrictions on queries.

For the remainder of this paper, we omit the first argument (history) from Map and Invert since we only consider per-revision integrity in this paper due to space constraints. However, complete integrity is not difficult to achieve and is discussed in Section 5.

2.4 Replay and Reconstitution Algorithms

Suppose we have a family of transformations $\mathcal{T}_K$ for some key space $K \subseteq K_1 \times K_2$. Let $(k_1, k_2) \in K$. Suppose that all sites apply $\mathsf{Map}_{\mathsf{k}_1}$ to each revision before sending it to C. A key goal is to minimize the number of applications of $\mathsf{Invert}_{\mathsf{k}_2}$ that must be applied to 'undo' the transformation for each revision. An important observation is that the number of such applications need only depend on the length of the reconstituted document string $\delta_{\boldsymbol{H}}(\epsilon)$ where $\boldsymbol{H}$ is a history that has not been transformed. Thus, the time complexity is linear in $|\delta_{\boldsymbol{H}}(\epsilon)|$ as opposed to linear in $|\boldsymbol{H}|$. Clearly, this is of considerable practical value since a document may have a long revision history whose early revisions no longer contribute to the present document content.

Firstly, we need to build a list of operations that must be executed in sequence to reconstitute the document string - see Algorithm 1. It is necessary to keep a reference to the 'parent' revision of a particular operation since it is such revisions that the $\mathsf{Invert}_{\mathsf{k}_2}$ algorithm is applied to in the end. Now any string that is inserted into the document as a result of a single revision may be broken up into many contiguous portions due to subsequent deletions. We will refer to these contiguous portions as *pieces* and each one points to its parent revision. A piece may be described by an offset and length with respect to the inserted string - this also obviates the need to allocate memory to store multiple substrings.

A self-balancing search tree of pieces ordered by document position is constructed in the replay algorithm. Roughly speaking, an insertion is handled by finding the piece that precedes it (if it exists) in the tree and splitting that piece if its range overlaps with the position of the string to be inserted. We can avoid shifting the positions of the pieces that follow it in the tree by instead translating the positions of subsequent operations (subtracting the number of characters inserted in the case of insertion). This is achieved by maintaining a 'delta' integer to be added to all operations. Deletion is handled by removing all pieces whose range completely overlaps with the deleted range, and appropriately splitting any pieces whose range partially overlaps; the delta is increased by the number of characters removed.

The replay algorithm (Algorithm 2) builds up a search tree of pieces. Recall that a piece specifies a portion of a string inserted in a revision in addition to referencing that revision. Subsequently, in the 'reconstitute' algorithm (Algorithm 4), the document's content is assembled by performing an in-order traversal of the tree, and for each piece: (1) applying Invert_{k_2} to its associated revision; (2) normalizing the result; and (3) projecting the string in its insertion mutation in accordance with the offset and length attributes of the piece. An outline of the these algorithms can be found in Figure 3.

The behavior of the replay algorithm for insertions is illustrated in Figure 2. In the figure, two simple revisions are considered that insert text into the document. In particular, this signifies how a piece is fragmented through successive replays while retaining a pointer to the original revision structure that generated it. In this example, no transformation has been applied to the text. Therefore the correct document content after revisions r1 and r2 have been applied may be reconstituted by performing an in-order traversal of the tree.

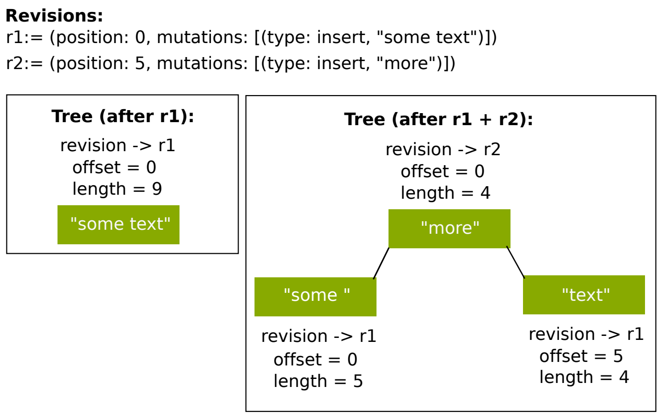

Fig. 2. Formation of the tree as revisions are replayed

3 Concrete Collaborative Editing System

The STCE model presented in the previous section is compatible with the supported features of Google Docs. Some of these features are directly exposed via standard APIs published by Google, whereas the availability of others may not be guaranteed at present - the latter were discovered by reverse engineering and experimentation. In brief, in order for our approach to work, it is necessary that such systems support both structural invariance for revisions and arbitrary range queries on the history. It will be shown later however that the latter requirement can be somewhat relaxed. The former is attractive because it facilitates unrestricted 'undo' (a feature supported for example by the open source editor Etherpad [6]) along with granular logging.

3.1 Correspondence between the STCE Model and Google Docs

Google Docs fits directly into our STCE model. Its server acts as the coordinator C and each collaborator session can be viewed as a site. Google Docs is a web application with a Javascript-based client-side front-end, referred to in this paper as the *client*. Edits to the document or movements of the cursor generate events that result in *mutate* requests being periodically sent to the server in an asynchronous manner; no such requests are sent during spans of inactivity. However, when events are being generated, the period between network requests is low enough to give rise to small revisions on average. An empirical analysis, examined later, reveals that on the order of 1 or 2 inserted characters are embedded in a typical update. This alone has security implications.

The client also receives periodic updates on the edits applied to the document in concurrent sessions, which is captured (only synchronously) in STCE by periodic invocation of $C.\mathsf{GetHistorySize}$ followed by

Fig. 3. Replay Algorithms, abstractly, as introduced in Section 2.4

Algorithm 1 MakeOpList

```
Input: $\boldsymbol{H} : \mathbb{H}$
 1: $m \leftarrow |\boldsymbol{H}|$
 2: $L \leftarrow$ new list
 3: $i \leftarrow 0$
 4: for $0 \leq i < m$ do
 5:     Parse $\boldsymbol{r} := (p, \mathsf{mutations}) \leftarrow \boldsymbol{H}[i]$
 6:     $\boldsymbol{r'} \leftarrow \mathsf{Normalize}(\boldsymbol{r})$
 7:     Parse $(\tau_{\mathsf{del}}, k), (\tau_{\mathsf{ins}}, v) \leftarrow \boldsymbol{r'}$
 8:     Append $(p, \boldsymbol{r}, (\tau_{\mathsf{del}}, k))$ to $L$
 9:     Append $(\min(p, p + k), \boldsymbol{r}, (\tau_{\mathsf{ins}}, v))$ to $L$
10: end for
11: return $L$
```

Algorithm 2 Replay

```
Input: $L$ ▷ List of operations outputted by MakeOpList.
 1: $T \leftarrow$ new search tree
 2: $\Delta_p \leftarrow 0$
 3: for all $(p, \boldsymbol{r}, \mu) \in L$ do
 4:     $p \leftarrow p + \Delta_p$
 5:     $\tau \leftarrow \mathsf{proj}_0(\mu)$
 6:     if $\tau = \tau_{\mathsf{ins}}$ then
 7:         $\Delta_p \leftarrow \Delta_p - \mathsf{ReplayInsertion}(T, p, \boldsymbol{r}, \mu)$
 8:     else if $\tau = \tau_{\mathsf{del}}$ then
 9:         $\Delta_p \leftarrow \Delta_p + \mathsf{ReplayDeletion}(T, p, \boldsymbol{r}, \mu)$
            ▷ ReplayDeletion entails more cases than ReplayInsertions but in many ways is similar. It is omitted in this version of the paper due to space constraints.
10:     end if
11: end for
12: return $T$
```

Algorithm 3 ReplayInsertion

```
Input: $T$
Input: $p$
Input: $\boldsymbol{r}$
Input: $\mu := (\tau_{\mathsf{ins}}, v)$
 1: $\mathsf{Pre} \leftarrow \{x \in T \mid \mathsf{x.position} < p\}$
 2: if $\mathsf{Pre} \neq \emptyset$ then
 3:     $\mathsf{prev} \leftarrow \max(\mathsf{Pre})$ (piece with max position)
 4:     if $\mathsf{prev.position} + \mathsf{prev.length} > p$ then
 5:         $\phi_1, \phi_2 \leftarrow \mathsf{Split}(\mathsf{prev}, \mathsf{p}, |v|)$ ▷ Split divides a piece about a position into two sub-pieces. The position of the second piece is shifted by the third argument.
 6:         Remove prev from $T$
 7:         Add $\phi_1$ to $T$
 8:         Add $\phi_2$ to $T$
 9:     end if
10: end if
11: Add new piece to $T$ with position:=$p$, length:=$|v|$, offset := 0, revision:=$\boldsymbol{r}$
12: return $|v|$
```

Algorithm 4 Reconstitute

```
Input: $T$ ▷ Tree of pieces outputted by Replay
Input: $f : \mathcal{R} \rightarrow \mathcal{R} \cup \{\perp\}$
 1: $w \leftarrow \epsilon$
 2: $U \leftarrow$ list of pieces from in-order traversal of $T$
 3: for all $u \in U$ do
 4:     $\boldsymbol{r'} \leftarrow f(u.\mathsf{revision})$
 5:     if $\boldsymbol{r'} = \perp$ then
 6:         return $\perp$
 7:     end if
 8:     $v \leftarrow \delta_{\boldsymbol{r'}}(\epsilon)$
 9:     if $v = \perp$ then
10:         return $\perp$
11:     end if
12:     $w \leftarrow w \parallel v[u.\mathsf{offset}, u.\mathsf{offset} + u.\mathsf{length}]$
13: end for
14: return $w$
```

invocation of C.LoadRevisions. An outline of the correspondence between STCE and Google Docs[**]. is given in Table 1.

STCE	Analog in Google Docs
C.EstablishSession	HTTPS GET *edit* request: Returns document content string, site/session id and most recent revision number.
C.Edit	HTTPS POST *mutate* request: Contains a sequence of JSON encoded mutations.
C.UpdateCursorPosition	HTTPS POST *mutate* request: Contains a JSON encoded structure containing the new starting and ending positions of the cursor (for selection).
C.LoadRevisions	HTTPS GET *load* request: Specifies starting revision and ending revision (inclusive) of the desired range. Returns a sequence of JSON encoded revisions.
C.GetHistorySize	HTTPS GET *bind* request: Returns updates made by other collaborators; includes most recent revision number at the time of request, which corresponds to $\|\boldsymbol{H}\|$.

Table 1. Correspondence between the interface to the coordinator C in our STCE Model and Google Docs

In particular, if formatting information is ignored, the document content is represented in Google Docs by a string of UTF-8 characters. Edits submitted by collaborators comprise an array of mutations (insertions or deletions) bundled into an atomic unit called a revision. Each revision is assigned a unique positive integer. The property of structural invariance of revisions, discussed in Section 2, is adhered to - potentially for the reasons mentioned earlier. Recall that structural invariance requires that the server persist the exact revision representation it receives from the collaborator without any manipulation. Arbitrary range queries are also supported; this functionality is provided by the official API [17]. It is important to note that the Google Docs interface briefly described in Table 1 pertains to the interface employed by the Google Docs client and not that specified in the API.

3.2 Implementation

A proof-of-concept browser extension was developed for the Firefox web browser [18] (tested with Firefox version 11).

The extension consists of two fundamental components - a lightweight module written in Javascript, which is loaded by the browser, and a more complex core written in Java that performs almost all of the processing. Intercommunication between Javascript and Java is facilitated by LiveConnect [19]. This solution offers far better performance for cryptographic computations than similar implementations developed entirely in Javascript [8, 9], albeit at the expense of portability among browsers. Like the other implementations, the extension intercepts and modifies requests and responses depending on whether the URI is determined to match for Google Docs. Such protocol messages are then processed by the Java component. In addition, our extension establishes a 'background' HTTPS connection with the Google Docs server. This is made possible because the cookie used for session authentication can be readily captured since we are listening to HTTP messages to/from Google Docs.

Recall the abstraction of a revision transformation defined in Section 2.3. In the prototype extension, the concrete transformation that is adopted performs authenticated encryption of revisions. Therefore, both confidentiality and integrity of the document content are protected. More precisely, assume a set of collaborators share a symmetric key for a document. Now let the key space K be $\{0,1\}^{\lambda}$.

Our overall approach is independent of the choice of cipher or MAC. For the prototype, AES in GCM mode was chosen, largely due to performance. In comparison to the mode (RPC) employed in [8], the throughput of CTR mode is higher due to the encryption of 'chaining' nonces in RPC. This becomes more important for revisions that insert sizable content strings. Furthermore, aside from content, revision metadata can be authenticated using the same GHASH operation (this also readily facilitates linear integrity of revisions), which requires only a single multiplication in $GF(2^{128})$ for each block of encrypted and authenticated-only

[**]This work targets the version of Google Docs available at the time of writing - April 2012.

data. Another motivation is the parallelizable nature of both GHASH and CTR mode. Although sufficient for our proof-of-concept implementation, replacing GHASH with a more secure MAC should probably be considered for a deployed implementation due to several vulnerabilities of polynomial MACs, especially those based on $GF(2^{128})$ [20].

The transformation is thus described by the pair of algorithms ($\mathsf{Map}_k := \mathsf{EncR}_k, \mathsf{Invert}_k := \mathsf{DecR}_k$). Informally, EncR_k is a randomized algorithm that is roughly described as follows[⋆⋆⋆]

Algorithm 5 EncR_k

```
Input: $\boldsymbol{r} \in \mathcal{R}$
 1: Uniformly sample $\mathsf{IV} \xleftarrow{\$} \{0,1\}^\lambda$
 2: $s \leftarrow \epsilon$
 3: for all insertion mutations $\mu_i$ in $\boldsymbol{r}$ do
 4:     Parse $\mu_i$ as $(\tau_{\mathsf{ins}}, v_i)$
 5:     $s \leftarrow s \parallel v_i$
 6: end for
 7: $c \leftarrow \mathsf{Base64Encode}(\mathsf{Encrypt}_{\mathsf{AES-GCM}_k}(s, \mathsf{IV}))$
 8: Replace each $v_i$ with characters from $c$ equal in length to the original $v_i$. Denote the updated $\boldsymbol{r}$ as $\boldsymbol{r'}$. Denote the
    remainder of $c$ as $c'$.
 9: Append the mutation sequence $((\tau_{\mathsf{ins}}, c'), (\tau_{\mathsf{del}}, -|c'|))$ to $\boldsymbol{r'}$.
10: return $\boldsymbol{r'}$.
```

In brief, Algorithm 5 encrypts and MACs, using AES in GCM mode, the concatenation of every content string that is featured in a revision. In Google Docs, a revision is a JSON object encoded in UTF-8. As such, the raw bytes constituting the ciphertext and tag (produced by the MAC) must be appropriately encoded - we use Base64. The resultant encoded string can be partially distributed over the original mutations, but the remainder must be stored somewhere within the revision to enable future MAC validation and decryption.

We omit a description of the deterministic algorithm DecR_k but as expected, it merely removes the auxiliary mutations and decrypts the content (after validating the MAC).

3.3 Replaying Revisions

When a document is initially opened, the browser extension uses the background HTTPS connection it has established in a new thread to asynchronously fetch the complete revision history for the document ($C.\mathsf{LoadRevisions}$). As revisions are received, the replay algorithm (Algorithm 2) described in Section 2.4 is executed. The following steps are executed:

1. Run Algorithm 1 followed by Algorithm 2 to build a tree of *pieces* (substrings of the content inserted in a revision) that contribute 'content' to the present document string. Each piece in the tree is linked to the revision that contains it.
2. Transform the 'content' of each piece in the tree back into plaintext by running the reconstitution algorithm (Algorithm 4) to assemble the decrypted content string. The reconstitution algorithm applies DecR_k to the revision linked to each piece. In summary, the following steps are executed for every piece in the tree.
   (a) Verify the authentication tag of the revision. Exit if verification fails.
   (b) Decrypt the 'content' of the piece. In practice, only the ciphertext blocks in the revision that correspond to the piece are decrypted. Thus, a stream encipherment mode such as CTR is advantageous.

[⋆⋆⋆] Support for complete integrity defined in Section 5 is omitted from this description. One way to achieve this is to let EncR_k take a history argument and include the tag from the last revision in the authenticated-only data in the MAC (GHASH in this case).

(c) Append the decrypted string to the document string

Concurrently, the extension intercepts the HTTPS response containing the complete document content that is used to render the document in the Google Docs editor. It then substitutes the reconstituted (decrypted) string for the original string in the response (which is merely ciphertext), and the Google Docs client renders the decrypted document.

3.4 Snapshot Optimization

A snapshot is a recording of the complete textual state of the document at a position in the discrete timeline of revisions. Snapshots have the same purpose here as 'checkpoints' in [12]. A snapshot consists of a string of text constituting the documents global content at the given time. Such a feature would be employed to optimize processing of a timeline, as a single revision subsumes the cumulative effect of a sequence of revisions. Accordingly, this would reduce the bandwidth consumption incurred by loading the revision history while also curtailing the number of replay steps necessary. This mechanism also helps to minimize the expansion overhead of encryption in addition to minimizing cost of decryption / MAC validation.

Concretely, a snapshot is a revision consisting of a deletion mutation that deletes the entire document content followed by an insertion mutation that restores it. Naturally, the snapshot's insertion mutation would be encrypted. We leave to future work the problem of systematically creating and locating snapshots. It seems that synchronization may be the most significant obstacle, but this can be mitigated by waiting for periods of inactivity.

Remark 1. The inclusion of a snapshot mechanism allows us to relax some of the assumptions made of the cloud provider. In particular, we only require that a constant number of most recent revisions in the timeline are subject to range queries and/or persisted.

4 Evaluation

4.1 Statistical Analysis of Revisions

A small collection of 20 documents authored in Google Docs were submitted by a small number[†] of distinct users for analysis. These documents, evolved organically, were shared among an average of 3 users. Analysis of these documents reveals some interesting findings, although a much larger and diverse sample size is needed to draw clear conclusions.

	Mean
Bytes Transferred Per Revision	701.2
Number of Inserted Characters Per Insertion Revision (filtered)	1.96
Number of Inserted Characters Per Insertion Revision (Unfiltered)	3.55

Table 2. Statistics of Revision Histories for Sample Documents

In Table 2, 'filtered' means that only insertion revisions with a total number of characters ≤ 10 were considered. This amounts to 99.3% of all revisions processed, which eliminates a significant number of revisions, though not all, generated by pasting or other means. The standard deviation in this particular case was 0.93. The high mean for the unfiltered case is explained by the occurance of considerably large revisions in the data set arising from pasting or other non-typing events. Of particular interest are those revisions arising from 'naturally' typed edits. Another interesting question was the observed distribution of the number of inserted characters in revisions incorporating insertions. A histogram based on all revisions across all sample documents is shown in Figure 4

[†]This is a correction from the proceedings version of this work.

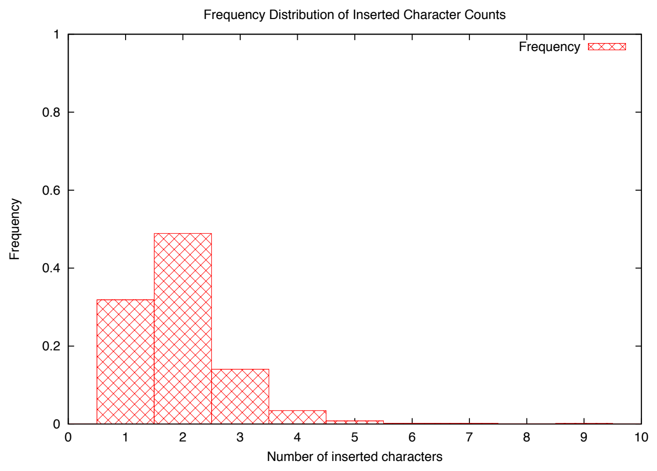

Fig. 4. Relative Frequency Distribution of the Number of Inserted Characters Per Insertion Revision

If the histogram is shifted to the left by 1, it bears some resemblance to the Poisson distribution with $\lambda \approx 1$. Although much work has shown a connection between the Pareto distribution and keystrokes sent on a network connection [21, 22], this distribution does not appear to fit our empirical data. Clearly, more data is needed to find an appropriate model.

Remark 2. Considerable variability has been observed with autosaving in Google Docs and thus there can be fluctuating burst lengths for a variety of reasons ranging from network conditions to local workloads. Hence, these results were obtained to help extract some macro-scale behavioral patterns.

The histogram demonstrates that there will naturally be suboptimal ciphertext expansion in the proposed system. Furthermore, the figure reveals significant information leakage that can be exploited by an attacker. Together with timing information, there is scope to learn information about the plaintexts. Such attacks are discussed in Section 5.

4.2 Performance

Overhead Size AES-GCM (128-bit block size) encryption is employed by the browser extension, and is configured to use a 16-byte MAC. Every revision is associated with a unique IV. Since a revision is an encoded JSON object, it is necessary to base64-encode the ciphertext + IV. Finally, two 'dummy' mutations are added - an insertion followed by a deletion. Consequently, the ratio of size of the processed mutations over the size of the single insertion mutation is approximately 4.82. To give concrete sizes, a mutation of 28 bytes grows to three mutations requiring an aggregate 135 bytes. Since revisions sent to the server typically contain additional metadata and formatting information, the percentage of bandwidth consumed by such overhead is less pronounced. This is highlighted in the results that were collected from analysis of sample documents, as shown in Table 3. It seems that on average approximately 4% of the bandwidth for insertion edits is consumed by the overhead incurred by authenticated encryption, although the overhead percentage is significant for the actual mutation sequence in a revision. Therefore, there is an appreciable impact on storage and bandwidth for initial loads, which further supports the future implementation of snapshots.

It must be emphasized that the bandwidth and storage overheads are markedly larger than the solution presented in [8], mainly due to constant-sized JSON formatting in the 'dummy' mutations. However, this compensated by our preservation of full collaboration.

Running Time A microbenchmark was conducted on a machine with an Intel Core i3 processor running at 3.2 GHz and 4 GB of RAM. The mean was taken of 1,000 executions for each case. Each test run involved generating a 16-byte IV and performing an AES encryption in the selected mode of operation. Our implementation used the Bouncy Castle cryptography library [23].

	Mean	**Standard Deviation**
Percentage Overhead Per Mutation Sequence	66.52%	13.18%
Percentage Overhead Per Revision	40.46%	5.8%
Percentage Overhead Per *Total* (headers + body) Mutate HTTP Message	4.37%	0.37%

Table 3. Overhead Incurred by Authenticated Encryption (Unoptimized) For Insertion Edits

Using the benchmark environment, the running times of the replay and reconstitution algorithms were evaluated. A small set of documents was employed, each containing approximately 10,000 revisions, half of which were encrypted using our browser extension. Table 4 gives a rough guideline of the real-time performance of the initial load process where the fetched revisions are used to build a tree. Unsurprisingly, an appreciable cost is incurred for the encrypted documents, although it is small enough to remain hidden from users. Nevertheless, this is clearly not scalable. As the revision history increases, the bandwidth costs to transfer the history in addition to the processing thereof become impractical. Therefore, the snapshot algorithm described in Section 3.4 must be employed as an optimization. In practice, the point at which the history size becomes infeasible is determined by the timeouts dictated by browsers. Hence, it is necessary that the latency of the load revisions requests along with tree processing and reconstitution are less than this limit. Empirical results would suggest that this limit is well beyond 20,000 revisions for common usage patterns, and thus snapshots would not have to be taken frequently.

	Mean	**Standard Deviation**
Running time of replay algorithm (documents without encryption)	$< 1\mu$s	$< 1\mu$s
Running time of replay algorithm (documents with encryption)	107 ms	11 ms
Running time of reconstitution algorithm (documents with encryption)	160 μs	54 μs

Table 4. Run-time performance of algorithms executed during a document's initial loading stage

5 Security Analysis

5.1 Threat Model

A threat model that only focuses on passive attacks is considered. A malicious cloud provider could alter the client-side part of the web application to inject Javascript that surreptitiously relays to the server the unencrypted content rendered in the browser. Therefore, we assume the client-side code executing in the browser is trusted. This is the position assumed in other works, most notably [8]. Additionally, a host of other active threats such as mounting a man-in-the-browser attack are also not covered in this threat model.

We return briefly to the STCE model wherein we can express the desired security properties. Disregarding the client-side code, the interface to the coordinator C is a good reflection of the view of the real system. Let P be a set of parties with access to the document. Naturally, we have that $C \in P$. Let $S \subset C$ be a set of sites who possess a *pre*-shared key k for the document in question. Let $\mathcal{A}$ be an adversary in $P \backslash S$. We define some desirable security properties:

1. **Confidentiality**: $\mathcal{A}$ has a negligible advantage in learning any information about a ciphertext under an adaptive Chosen Ciphertext Attack (negligible advantage in the IND-CCA2 game discussed in Section 2.3).
2. **Revision Integrity**: $\mathcal{A}$ has a negligible advantage in forging a revision in $\boldsymbol{H}$ that passes the integrity check performed by parties in S.
3. **Complete Integrity**: A stronger property than (2) requires that an attacker is unable to efficiently forge an alternate history whose revisions all satisfy (2) i.e. it has negligible advantage in the following game.

The attacker is permitted to make adaptive queries to the oracles $\mathsf{MapH}(\mathsf{Map}_{k_1}, \cdot)$ and $\mathsf{InvertH}(\mathsf{Invert}_{k_2}, \cdot)$. Suppose it makes $q = poly(\lambda)$ queries to the former. It records $\boldsymbol{H_i'} \leftarrow \mathsf{MapH}(\mathsf{Map}_{k_1}, \cdot)$ where $\boldsymbol{H_i} \in \mathbb{H}$ for $0 \leq i < q$. It wins the game if it outputs a non-empty history $\boldsymbol{H'}$ such that $\boldsymbol{H'}$ is not a substring of any $\boldsymbol{H_i'}$ for $0 \leq i < q$ and $\mathsf{InvertH}(\mathsf{Invert}_{k_2}, \boldsymbol{H'}) \neq \perp$.

Properties (1) and (2) follow from our adoption of AES-GCM. See Appendix A for more details. As stated earlier, a more secure alternative is to replace GHASH with a more secure MAC. Certain countermeasures are needed to limit the power of an adversary to choose arbitrary long messages; for example a revision may be broken up into independently encrypted/authenticated units whose size is limited.

Property (3) ensures that there exists some means to prevent C rearranging the revisions in a history to yield an alternative document content string. Thus, there needs to be a chain of dependence between successive revisions, for example by means of a linked authenticated structure. Consider a site with knowledge of a current history $\boldsymbol{H}$ (its local copy). Suppose it has to commit a new revision $\boldsymbol{r}$. A means to fulfill property (3) is to compute a linear hash chain of all revisions in $\boldsymbol{H}$. This is similar to the approach adopted in [12]. Naturally, only a single hash needs to be stored by a site, thus incrementality is not sacrificed. This hash along with a precise description of a revision is incorporated into the MAC that is included as part of the revision. Now it may be the case that other sites, who are synchronized with knowledge of $\boldsymbol{H}$, concurrently send revisions to C. These revisions can be arranged in an arbitrary order by C. However, the correct operational transformations must be performed on this ordering. Otherwise this can be detected (by virtue of including a site's position in the MAC). Hence, the only scope for re-ordering afforded to C is at such points of concurrency, which is inherent in the consistency notions of OT systems. As such, the game definition of property (3) deprives the adversary of this particular case. We defer to Appendix A a more formal security assessment of this property together with a description of other active attacks mountable by C such as dropping edits and forking documents as explored in [12]. Furthermore, this paper omits a detailed treatment of Property (3) from both the algorithm descriptions and the implementation, in particular verification of an entire history.

The issue of information leakage arising from timing patterns (including those due to typing patterns) is a fundamental concern. Timestamp information recorded in each revision facilitates the formation of a detailed profile of a user's editing characteristics. This prompts the questions: is it feasible for the browser extension to smoothen the distribution by inserting artificial delays? Perhaps a dedicated out-of-browser proxy would afford more control over this. Alternatively, perhaps another option is to implement a custom feature that allows a user to enter a particularly sensitive string of text (social security numbers etc...) that is then collectively sent as a single revision using the 'background' connection. This would serve to prevent information leakage sourcing from typing traits. In fact, given the scope for information leakage, users may only desire privacy for certain sensitive fields in their documents, and thus disable encryption for the remaining content.

5.2 Conclusions and Future Work

We have presented a new approach for protecting confidentiality and integrity of documents in collaborative editing systems. A proof-of-concept implementation developed for the Firefox web browser demonstrates the practicality of this solution over alternative work in the literature. In particular, this approach is the first, to the best of our knowledge, to preserve full collaboration and not to rely on externalized, out-of-band resources to store metadata for deployed untrusted OT systems.

Many questions have been raised by this research, which will be followed up as part of future work. Some of these question relate to ensuring protection against malicious active attacks such as forking documents, lessening information leakage from timing information, and investigating and implementing optimizations such as snapshots.

Acknowledgments. The authors would like to thank the anonymous reviewers of ISC 2012 for many helpful comments and suggestions. The first author's work was partly supported by the Irish Research Council. The work of the second and third authors was supported by funding from Enterprise Ireland.

References

1. Clear, M., Reid, K., Ennis, D., Hughes, A., Tewari, H.: Collaboration-preserving authenticated encryption for operational transformation systems. In Gollmann, D., Freiling, F.C., eds.: ISC. Volume 7483 of Lecture Notes in Computer Science., Springer (2012) 204–223
2. Mitzenmacher, M., ed.: Fully homomorphic encryption using ideal lattices. Number September, ACM Press (2009)
3. Brakerski, Z., Vaikuntanathan, V.: Fully Homomorphic Encryption from Ring-LWE and Security for Key Dependent Messages Advances in Cryptology – CRYPTO 2011. Volume 6841 of Lecture Notes in Computer Science. Springer Berlin / Heidelberg, Berlin, Heidelberg (2011) 505–524
4. Katz, J., Sahai, A., Waters, B.: Predicate encryption supporting disjunctions, polynomial equations, and inner products. In: Proceedings of the theory and applications of cryptographic techniques 27th annual international conference on Advances in cryptology. EUROCRYPT'08, Berlin, Heidelberg, Springer-Verlag (2008) 146–162
5. : Google docs (2012) `https://docs.google.com/`.
6. : Etherpad collaborative real-time editor (2012) `http://www.etherpad.com`.
7. : Apache wave (2012) `http://incubator.apache.org/wave`.
8. Huang, Y., Evans, D.: Private editing using untrusted cloud services. In: Distributed Computing Systems Workshops (ICDCSW), 2011 31st International Conference on. (2011) 263 –272
9. Adkinson-Orellana, L., Rodríguez-Silva, D.A., Rodríguez-Silva, D.A., Burguillo-Rial, J.C.: Privacy for google docs: Implementing a transparent encryption layer. In: 2nd Cloud Computing International Conference - CloudViews. (2010)
10. Bellare, M., Goldreich, O., Goldwasser, S.: Incremental cryptography: The case of hashing and signing. (1994) 216–233
11. Lipmaa, H.: Secure and Efficient Time-stamping Systems. Dissertationes mathematicae Universitatis Tartuensis. Tartu University Press (1999)
12. Feldman, A.J., Zeller, W.P., Freedman, M.J., Felten, E.W.: Sporc: group collaboration using untrusted cloud resources. In: Proceedings of the 9th USENIX conference on Operating systems design and implementation. OSDI'10, Berkeley, CA, USA, USENIX Association (2010) 1–
13. Buonanno, E., Katz, J., Yung, M.: Incremental unforgeable encryption. In Matsui, M., ed.: Fast Software Encryption. Volume 2355 of Lecture Notes in Computer Science. Springer Berlin / Heidelberg (2002) 317–325
14. Sun, C., Ellis, C.: Operational transformation in real-time group editors: issues, algorithms, and achievements. In: Proceedings of the 1998 ACM conference on Computer supported cooperative work. CSCW '98, New York, NY, USA, ACM (1998) 59–68
15. Li, D., Li, R.: Preserving operation effects relation in group editors. In: Proceedings of the 2004 ACM conference on Computer supported cooperative work. CSCW '04, New York, NY, USA, ACM (2004) 457–466
16. Li, R., Li, D.: Commutativity-based concurrency control in groupware. In: Collaborative Computing: Networking, Applications and Worksharing, 2005 International Conference on. (2005) 10 pp.
17. : Google documents list api version 3.0 (2012) `https://developers.google.com/google-apps/documents-list/`.
18. : Mozilla firefox (2012) `http://www.mozilla.org/firefox`.
19. : Liveconnect technical documentation (2012) `https://developer.mozilla.org/en/LiveConnect`.
20. Saarinen, M.J.O.: Cycling attacks on gcm, ghash and other polynomial macs and hashes. Cryptology ePrint Archive, Report 2011/202 (2011) `http://eprint.iacr.org/`.
21. Donoho, D.L., Flesia, A.G., Shankar, U., Coit, V.P.J., , Staniford, S., Coit, J., Staniford, S.: Multiscale stepping-stone detection: Detecting pairs of jittered interactive streams by exploiting maximum tolerable delay. In: in Proc. of The 5th International Symposium on Recent Advances in Intrusion Detection (RAID, Springer (2002) 17–35
22. Paxson, V., Floyd, S.: Wide area traffic: the failure of poisson modeling. IEEE/ACM Trans. Netw. **3** (1995) 226–244
23. : Bouncy castle crypto api (2012) `http://www.bouncycastle.org`.

A Security Considerations

A.1 Confidentiality and Revision Integrity

In regard to Properties (1) and (2) in Section 5 concerning confidentiality and revision integrity respectively, it is well established that using the approach of encrypt-then-MAC with an IND-CPA secure symmetric encryption scheme and a strongly unforgeable MAC yields an IND-CCA2 secure system. We chose an authenticated encryption mode of operation in the proof-of-concept implementation.

A.2 Complete Integrity

Informally, an attacker cannot forge verifiable history sequences beyond subsequences of those that it has been given. A common approach to achieving integrity for a list structure such as a revision history is to incorporate into any element the hash of the previous element with the exception of the first element of the list. This is referred to as a linear hash chain, and it is the approach adopted in [12] to provide integrity of the revision history.

Recall Property 3 from Section 5. The attacker is permitted to make adaptive queries to the oracles $\mathsf{MapH}(\mathsf{Map}_{k_1}, \cdot)$ and $\mathsf{InvertH}(\mathsf{Invert}_{k_2}, \cdot)$. Suppose it makes $q = poly(\lambda)$ queries to the former where λ denotes the security parameter. It records $\boldsymbol{H_i'} \leftarrow \mathsf{MapH}(\mathsf{Map}_{k_1}, \cdot)$ where $\boldsymbol{H_i} \in \mathbb{H}$ for $0 \leq i < q$. It wins the game if it outputs a non-empty history $\boldsymbol{H'}$ such that $\boldsymbol{H'}$ is not a substring of any $\boldsymbol{H_i'}$ for $0 \leq i < q$ and $\mathsf{InvertH}(\mathsf{Invert}_{k_2}, \boldsymbol{H'}) \neq \perp$.

Briefly, the linear hash chain approach can be realized as follows. Since these ideas are well-established, we give a brief overview here. In some authenticated encryption schemes such as GCM, authentication-only information can be incorporated into the MAC besides the ciphertext. Therefore the authentication *tag* τ_i of revision $\boldsymbol{r}_i$ can be incorporated as the authentication-only information when computing the *tag* τ_{i+1} for revision $\boldsymbol{r}_{i+1}$. We let $\tau_{-1} = \epsilon$ (empty string) to generalize for the special case of the first revision. We have that Invert involves performing MAC verification on τ_i by also incorporating τ_{i-1} into the authentication-only information. Assuming the MAC underlying the authenticated encryption scheme is unforgeable, an adversary cannot forge a "fresh" history except with at most negligible probability. Observe that there is nothing stopping the adversary "forking" a history given access to the oracles.

A.3 Other Security Concerns

If server C is malicious, there is plenty of scope for an active attack. One of the possibilities is dropping edits; C can choose to exclude the revisions supplied by particular subsets of users from the revision history such that the remaining users cannot determine that this has taken place. Another possibility is forking the revision chain where C partitions the set of users into disjoint sets and restricts sending notifications on edits by a user A to a user B unless A and B are in the same set. Resistance to this is considered in detail in [12]. Alas, in our setting, it is not possible to provide countermeasures against forking without appealing to out-of-band protocols.

Another assumption that is made in the threat model we consider is the fact that *none* of the sites are corrupt. Approaches other than simply employing symmetric encryption with shared keys must be considered to provide collusion resistance in the presence of corrupt parties.

A summary of other security concerns is as follows:

- Attacks based on timing patterns
- Complex user profiling (typing traits)
- Formatting information is not encrypted.
- Layout/structure of the document is not concealed.
- The server is assumed to not be malicious i.e. deviate from the protocol.
- Using a browser plugin leads to concerns about other plugins which may be malicious along with vulnerabilities that may exist in the browser.