SECRET: On the Feasibility of a Secure, Efficient, and Collaborative Real-Time Web Editor

Dennis Felsch
Horst Görtz Institute
for IT-Security, Chair for
Network and Data Security
Ruhr-University Bochum
dennis.felsch@rub.de

Christian Mainka
Horst Görtz Institute
for IT-Security, Chair for
Network and Data Security
Ruhr-University Bochum
christian.mainka@rub.de

Vladislav Mladenov
Horst Görtz Institute
for IT-Security, Chair for
Network and Data Security
Ruhr-University Bochum
vladislav.mladenov@rub.de

Jörg Schwenk
Horst Görtz Institute
for IT-Security, Chair for
Network and Data Security
Ruhr-University Bochum
joerg.schwenk@rub.de

ABSTRACT

Real-time editing tools like *Google Docs*, *Microsoft Office Online*, or *Etherpad* have changed the way of collaboration. Many of these tools are based on *Operational Transforms (OT)*, which guarantee that the views of different clients onto a document remain consistent over time. Usually, documents and operations are exposed to the server in plaintext – and thus to administrators, governments, and potentially cyber criminals. Therefore, it is highly desirable to work collaboratively on *encrypted* documents.

Previous implementations do not unleash the full potential of this idea: They either require large storage, network, and computation overhead, are not real-time collaborative, or do not take the structure of the document into account. The latter simplifies the approach since only OT algorithms for byte sequences are required, but the resulting ciphertexts are almost four times the size of the corresponding plaintexts.

We present *SECRET*, the first secure, efficient, and collaborative real-time editor. In contrast to all previous works, SECRET is the first tool that (1.) allows the encryption of whole documents *or* arbitrary sub-parts thereof, (2.) uses a novel combination of tree-based OT with a structure preserving encryption, and (3.) requires only a modern browser without any extra software installation or browser extension.

We evaluate our implementation and show that its encryption overhead is three times smaller in comparison to all previous approaches. SECRET can even be used by multiple users in a low-bandwidth scenario. The source code of SECRET is published on GitHub as an open-source project: **https://github.com/RUB-NDS/SECRET/**

ASIA CCS '17, April 02 - 06, 2017, Abu Dhabi, United Arab Emirates

© 2017 Copyright held by the owner/author(s). Publication rights licensed to ACM. ISBN 978-1-4503-4944-4/17/04...$15.00

DOI: http://dx.doi.org/10.1145/3052973.3052982

Keywords

Collaborative Editing; XML Encryption; JSON; Operational Transforms; Structure Preserving Encryption

1. INTRODUCTION

Services like Google Docs, Microsoft Office Online, or Etherpad use Operational Transforms (OT) [11] as fundamental technology to resolve editing conflicts by many concurrent users automatically. They are thus able to cope with arbitrary simultaneous and conflicting edit operations. These services have simplified our working life, and will further gain in importance.

However, they have a significant drawback: All documents are stored on a cloud server as plaintext, thus anyone with access to the server (system administrators, hackers, governments, etc.) may read and modify the documents. In their seminal paper, Feldman et al. [12] showed a direct way how to apply OT to encrypted documents: Their proposed system encrypts each edit operation separately on the client and sends them to a server that saves and distributes these edits to all other clients. Each client then has to reconstruct the document from these encrypted operations. Other proposals (cf. e. g. [17]) encrypt each plaintext byte separately on a client and use byte stream OT on the ciphertext blocks to synchronize them with a server. The general question if encrypted collaborative editing is possible was answered in the affirmative, and later improved by several publications [1, 8, 9, 27, 37].

Both approaches are generic, but also come with a generic penalty: The data the server has to store and transmit is significantly increased compared to the plaintext. Those papers that evaluated this overhead require an expansion factor of at least 3.75. In addition to data expansion, all previous approaches do not fit into the Software-as-a-Service (SaaS) model used by Google Docs and others, because they require additional software like Java applets or browser add-ons to execute cryptographic operations.

Another problem with previous approaches [1, 8, 9, 12, 17, 27, 37] is that if a new user has been granted edit rights on a document, he also would get history information. This can include sensitive data that was accidentally written (or pasted) by other users, e. g., comments or to-do annotations. To prevent this kind of information leakage, Micciancio [26] introduced the concept of obliviousness. While *obliviousness* can easily be achieved by creating a fresh copy of the document, this requires to re-encrypt all data and is inefficient for larger documents. Thus, a more scientific goal is to design a system offering this property directly.

In this paper, we explore the following questions with respect to a real-time and encrypted collaborative web editor:

- ▶ Can we eliminate the generic data expansion?
- ▶ Can we create a SaaS relying on a browser only?
- ▶ Can we achieve obliviousness?

To answer these questions, we created SECRET, a Secure, Efficient, and Collaborative Real-Time Editor. SECRET uses a novel combination of structure-preserving encryption and tree-based OT [10] to enable overcome the generic data expansion and to achieve obliviousness. SECRET only requires a browser offering state-of-the-art web technologies (e. g., WebSockets [13] and WebCrypto API [34]), thus being the first to eliminate all external software dependencies.

The design and implementation of SECRET is more than just combining existing technologies. We solved several challenges that lead to new insights of underlying concepts.

(1.) The W3C Web Cryptography API (WebCrypto API) enables cryptographic operations (symmetric / asymmetric encryption, digital signatures / MACs, generation of ephemeral keys), but it does not allow to handle long-lived, persistent keys. We require this in our approach.

(2.) We built our system upon ShareJS, a JSON-based collaborative cleartext web editing system. Since JSON Encryption does not support structure preserving encryption of document parts, we had to build an adaptation layer which maps XML and XML Encryption to JSON. This was challenging because of subtle differences in the document tree structures of both concepts.

(3.) Every edit in the plaintext (e. g., of an encrypted document), even if only a single byte is modified, results in a completely different ciphertext block, which has to be synchronized among all clients. In order to make real-time editing applicable and performant, we had to find an efficient (i. e. not too often, but often enough to avoid too many editing conflicts) way to trigger an encryption and synchronization event from changes in the DOM of the cleartext document in the browser.

Contributions.

- ▶ We describe a novel approach to combine structure-preserving encryption with tree-based OT, surpassing generic limitations of previous approaches (Section 4).
- ▶ We provide a comprehensive list of existing approaches, comparing and discussing their features and limitations (Section 5).
- ▶ We define requirements necessary for the creation of a secure, collaborative, real-time web editor and discuss challenges and solutions (Sections 6 and 7).
- ▶ We present SECRET, the first working prototype of a fully collaborative editing tool on encrypted documents (Section 8). In contrast to all previous approaches, SECRET is independent of external software and achieves *obliviousness*.
- ▶ We design and implement two complementary key management approaches: (1.) using a trusted keyserver and (2.) without relying on a trusted party by using the password-to-key functionality of the WebCrypto API (Section 8).
- ▶ We show the feasibility of SECRET by performing extensive evaluations and reveal that SECRET's encryption overhead is significantly smaller (92%) compared to all previous approaches ([8, with 382%], [17, with 275%]). We additionally compare SECRET with Google Docs and show that the overhead caused by the encryption is acceptable (Section 10).

The source code of SECRET is published on GitHub: **https://github.com/RUB-NDS/SECRET/**

2. FOUNDATIONS

Collaboration is appreciated in modern IT systems, and through appropriate tools the efficiency of such collaboration can be greatly enhanced. A widely adopted solution to coordinate software development projects are Version Control Systems (VCSs). Examples for VCSs are Subversion (SVN), Git, Mercurial, etc. The basic idea of a VCS is to record changes of a project's state. A set of changes is called a *revision*. It is common that a VCS allows *reverting* to older revisions or viewing the differences between them. In a VCS, changes are not created in real-time, but have to be triggered by a user explicitly, which is comparable to using a *save*-button. If conflicts between a user's local copy and the VCS arise, these usually have to be resolved manually.

Transferring this concept to create a real-time collaboration tool does not work for multiple reasons: The feature of a VCS to revert to an old revision gives a new user access to every intermediate state the document was in since it was created, which is unacceptable if the history of the document contains confidential information (e. g., trade secrets). Furthermore, changes simultaneously submitted to the server lead to merging problems. For instance, if the server created a new revision while a client is merging an older revision with its current local copy, the server would reject the merge and force the client to repeat the merge with the current revision. Real-time collaboration therefore requires a concept allowing the merge even if the views of the clients on the document diverge. In other words, a technique is required that can tolerate messages arriving in wrong order or messages getting lost. It has to guarantee that all clients end up in a consistent state.

2.1 Operational Transforms (OT)

Operational Transforms (OT) is a technique to automatically maintain a consistent view on a document even if many users are editing this document simultaneously. The idea behind OT is simple: Each user changes the document by performing a sequence of basic edit operations. If the document is modeled as a large sequence of bytes, each operation can be modeled as follows:

- ▶ `INS(sub, r)`: insert substring `sub` at position `r`.
- ▶ `DEL(r, s)`: delete all bytes from position `r` to `s`.

These operations are sent to a server that informs other users about the changes so that their local copies can be updated. However, simply forwarding the received basic operations will not work: If another user simultaneously inserted or deleted text before the actual position `r` of an operation, then different bytes will be deleted or `sub` will be inserted at a different position than intended.

Therefore, the task of the server is to *transform* the basic operations for each user if necessary, and to forward these transformed operations. Consider the following simple example originally taken from Nichols et al. [28]:

Alice and Bob are editing the same string: `ABCDE`. Alice executes `DEL(4,4)` and locally gets `ABCE`. At the same time, Bob applies `DEL(2,2)` to get `ACDE`. If these operations are simply exchanged, Alice executes `DEL(2,2)` to get `ACE`, while Bob applies `DEL(4,4)` and gets `ACD`. An inconsistent state appears. To avoid this problem, the server applies a transformation `T` on the second operations and `T` will change Bob's second operation from `DEL(4,4)` to `DEL(3,3)`. For a detailed description of `T`, we refer to Nichols et al. [28].

If the document is modeled differently, then the operations and the necessary transforms may be different. Note that `T` may also be computed on the client.

2.2 XML and JSON

The eXtensible Markup Language (XML) is a platform-independent text format that defines rules to encode documents [6]. XML documents form a tree structure, starting at the document *root* and consists of Elements with an opening tag (`<x>`) and a matching closing tag (`</x>`). Elements are nested and can additionally contain text contents (e. g., `<x>text</x>`) and attributes as key/value pairs (e. g., `<x name="value">`).

The JavaScript Object Notation (JSON) [5] is an open standard that defines an alternative to XML. Like XML, JSON is platform-independent and describes a tree structure. Basically, it can be used to describe the same data structures as XML, but uses a shorter notation. JSON uses curly brackets to declare objects and name/value pairs to define data (e. g., `{"x":"text"}`). JSON can also be used to describe tree like structures (e. g., `{"x":1,{"y":2}}`).

2.3 Encryption

Collaborative editing requires fast symmetric encryption and decryption. Cryptography implemented in pure JavaScript has proven to be slow (see e. g. [8, 12]). As an efficient alternative, modern browsers offer the Web Cryptography API [34]. The WebCrypto API is a W3C recommendation that describes a JavaScript Application Programming Interface (API) for basic cryptographic operations in web applications. It offers interfaces to generate keys, to encrypt and decrypt as well as to sign, verify, and hash data. We implement SECRET by using the block cipher AES with a key length of 128 bit in Galois Counter Mode (GCM).

An interesting aspect on the WebCrypto API is, that it does not offer a specific mechanism to store keys in the browser persistently (cf. [34, sect. 5.2]). This means, that the WebCrypto API does not offer a real key management. A key is technically a JavaScript object that is created whenever the API is used to generate or import a key. The key material itself can be protected by setting the `export` property to `false`. In this manner, the key can be used by its object reference, but the key bytes cannot be exported.

3. FORMAL MODEL

We designed SECRET using the following formal model.

3.1 Computational Model

Let $\mathcal{U} = \{U_1, ..., U_n\}$ be the set of users of SECRET, and let $\mathcal{S} = \{S_1, ..., S_p\}$ be the set of storage servers. We define a *session* of SECRET to be a tuple $\sigma_j = (M_j, S_i, k_j)$, where $M \subseteq \mathcal{U}$, $S_i \in \mathcal{S}$ and k_j is a randomly chosen, symmetric masterkey used with SECRET.

In each session σ_j, the set of users M_j use SECRET and k_j to compute a plaintext document m_j^t and a partially encrypted document $c_j^t = Enc_{k_j}(m_j^t)$, where t indicates the t-th snapshot of the documents.

Each plaintext document m_j^t is a tree, with element nodes as intermediary nodes, and content and attribute nodes as leaves. The corresponding ciphertext document is created by replacing a subtree (i. e. an element together with its contents) with a ciphertext element.

3.2 Security Model

The goal of an adversary is to learn the plaintext content of a ciphertext element of some c_j^t. Several security models can be used to define the security of SECRET. We analyze SECRET in all three models in Section 9.

Honest-But-Curious Cloud Server. In this model, the cloud provider is honestly providing storage services (it does not forge it), but it is passively reading the stored data and can forward this data (voluntarily or on court request) to third parties. This is the standard security model for cloud storage and for encrypted document editing and also considered in previous work (e. g., [37]).

Passive Man-In-The-Middle. In this model, the attacker passively reads all network traffic, but does not perform any active attacks like deleting or altering network packets. This model could also be called "honest-but-curious network".

Web Attacker Model. This model is the standard model for proving security of web applications [3]. An adversary in this model can access any open web application, learn its client-side code, send emails and other messages, and can set up their own (malicious) web application. The web attacker is unable to forge web origins [2], because this would undermine the security of any web application.

3.3 Obliviousness Model

In the obliviousness game, the adversary $\mathcal{A}$ can act as a user and join a session σ_j. When he is added to this session, he gets access to the key k_j and the actual snapshot m_j^t of the document. The adversary breaks the obliviousness property if he is able to compute a previous snapshot $m_j^{t'}, t' < t$ of the plaintext, with probability better than just guessing. We exclude trivial cases, e. g., the empty document ($t' = 0$).

4. NOVEL TREE-BASED ENCRYPTED OPERATIONAL TRANSFORMS

In this section, we present our novel approach of combining tree-based operational transforms with structure-preserving encryption.

4.1 Structure Preserving Encryption

Usually, encryption is applied to byte streams that are not required to have a structure. The encryption algorithm itself may impose its own structure, e. g., by subdividing the stream into blocks of equal length, by padding, and by adding prefixes or appendices (e. g., IV, MAC).

A *structure-preserving* encryption operation will ensure that the document structure is kept intact, and will encrypt different parts of the document separately. Amongst all encryption standards used today, only two have this property: With XML Encryption [16], each element or the content of each element can be encrypted separately, and the resulting `<EncryptedData>` element will be inserted as a replacement of the plaintext at the exact same position in the document tree. An alternative to XML Encryption is JSON Web Encryption (JWE) [20], where an encrypted JSON object is again a JSON object using *JWE JSON Serialization*.

The advantage of using structured encryption in collaborative editing is that encrypted parts of a document blend in with the rest of the structure. This way, major portions of the software do not have to take care whether an encrypted or plain part is processed.

4.2 Tree-based Operational Transforms

The naïve approach to implement OT on tree-structured documents (e. g., JSON or XML based) is to serialize these documents into a string and to synchronize this string among all clients. Any time a modification occurs, the modified tree is serialized, and the resulting string is compared with the serialization of the most recent snapshot of the document tree. Any differences detected will be encoded as operations and transmitted to the OT server.

Since the server does not know about the semantics of the string, situations occur where the application of OT would create unusable documents. Consider the JSON object `{"1":1}`. Alice adds a property named `"2"` with the value `2` and locally gets `{"1":1,"2":2}`. This is translated to the edit operation `INS(',"2":2', 6)`. At the same time, Bob deletes the property `1` to get `{}`. This is translated to the edit operation `DEL(1, 5)`.

The server receives both edits and detects that the `DEL`-command does not need to be transformed. The `INS`-command however has to be transformed to `INS(',"2":2', 1)` before it is forwarded to Bob. In the end, both Alice and Bob reach a consistent state, but the resulting string is `{,"2":2}` which is no valid JSON document.

In consequence, OT for tree-structured data requires a server implementation that is aware of the structure so that it can keep it intact. With ShareJS 0.6 [14], a web application library supporting OT on JSON documents is available.

4.3 Combining both primitives

For the purpose of implementing collaborative editing a disadvantage of JWE becomes visible: JWE requires key, metadata, and ciphertext to be part of the same data structure. This way, JSON documents containing multiple encrypted objects quickly grow in size which is what we would like to avoid. With XML Encryption in contrast, most of the metadata is optional and keys may be referenced and reused. Therefore, we use XML Encryption to implement SECRET.

Modern browsers provide rich support for XML, but do not support XML Security out of the box. Especially, they do not support XML Encryption. Therefore, we implemented a JavaScript library that supplies the necessary logic to create and process encrypted XML trees or parts thereof.

Even though the feature sets of XML and JSON are fundamentally different, there is a conceptual layer at which there are similarities between these two standards. Seen from the perspective of an XML element, its children are an ordered list of elements, like an array of objects in a JSON object. Its attributes in contrast are an unordered list of key-value-pairs, much like named properties in a JSON object. Text content is simply a string which is not different from the classic OT approach. The nesting of elements in XML is not as flexible as with JSON (e. g., attributes cannot be nested) and using certain data types (e. g., numerical literals) is not as intuitive as in the JSON case.

Our OT-compliant library for XML documents is an extension of those parts of ShareJS that were originally written to support JSON objects. This way, we can reuse the communication subsystem of ShareJS that uses WebSockets to realize asynchronous messaging. Our extension takes advantage of the common semantic layer outlined above to support the following 7 basic operations on XML documents:

(1.) Insert a new element based on an XML string.
(2.) Delete an existing element.
(3.) Move an element among its siblings.
(4.) Insert text into a text node.
(5.) Delete text from a text node.
(6.) Set or overwrite an attribute.
(7.) Delete an attribute.

More problematic are XML features that JSON does not provide like namespaces, comment nodes, and CDATA sections. From the perspective of the OT algorithm, we model namespaces as special case of an element's name. Comments and CDATA sections are modeled as special elements that have a different string representation and no attributes, but apart from that they can be manipulated like normal elements.

This design then allows us to apply XML Encryption to an OT-synchronized XML document. Because of using the structure-preserving property, the OT algorithm does not need to care about whether an element is encrypted or not – XML Encryption can be applied transparently. We thus achieve our goal and have an approach that enables OTs on (partly) encrypted documents.

5. RELATED WORK

The idea of Operational Transforms was originally invented by Ellis and Gibbs in 1989 [11]. Based upon this work, other publications refined the idea to enable concurrent editing of plain text (cf. e. g. [28, 31, 33]). Real-time collaborative working has become famous with Google Wave, which has been renamed to Apache Wave. It was announced in 2009 by Google and allows multiple users to edit the same document simultaneously. Although the Wave project was stopped by Google, its idea and concept was integrated into Google Docs. There are other real-time collaborative editing web applications, like Microsoft Office Online and Etherpad.

In 2002, Davis et al. described OT on Standard General Markup Language (SGML) documents [10], a predecessor of XML. This work was later extended to support collaborative editing of XML documents in centralized architectures with manual conflict resolution [19, 29] as well as in peer-to-peer environments [18].

The security property *obliviousness* was first introduced by Micciancio [26]. Its definition was later modified by

Author	**Src**	**Approach**	**OT**	**Struct. OT**	**Obliv**	**RT Coll.**	**Key Mgmt.**	**Vanilla Browser**	**Server**
Feldman et al.	[12]	Journal	**Yes**	*No*	*No*	**Yes**	**Yes**	*No* (Java-PlugIn)	*Custom*
Clear et al.	[8]	Journal	**Yes**	*No*	*No*	**Yes**	*No*	*No* (Firefox Ext.+Java)	Google
D'Angelo et al.	[9]	Document	*No*	*No*	*No*[1]	*No*	**Yes**	*No* (Firefox Ext.)	Google
Adkinson-Orellana et al.	[1]	Document	*No*	*No*	*No*[1]	*No*	**Yes**	*No* (Firefox Ext.)	Google
Huang and Evans	[17]	Document	**Yes**	*No*	*No*[1]	*No*	*No*	*No* (Firefox Ext.)	Google
Zhang et al.	[37]	Document	**Yes**	*No*	*No*[2]	*No*	*No*	*No* (Java Program)	Dropbox
Michalas and Bakopoulos	[27]	Document	*No*	*No*	*No*[1]	*No*	**Yes**	*No* (Greasemonkey Ext.)	Google
SECRET		Document	**Yes**	**Yes**	**Yes**	**Yes**	**Yes**	**Yes**	*Custom*

Table 1: Comparison of related work on cloud based encrypted editing. SECRET fulfills all requirements.

Buonanno et al. to apply it to incremental encryption schemes [7, Sect. 2.5].

There are two approaches to realize collaborative editing on encrypted documents in the literature (see Table 1):

Journaling Approach to Encrypted Collaboration. The first approach is to let each client encrypt every single edit operation before it is sent to the server. The server then only stores a long list of encrypted edit operations and provides it to the clients. We call this the *journaling approach*. A client has to download this list before the document can be displayed and used, which requires large storage and network overhead [8, Section 4.2]. This problem can be alleviated by coalescing edit operations on the list or by introducing *snapshots* of the document that can be used as starting point for a new client. However, such computationally heavy cleanup work has to be done by one of the clients since the server cannot decrypt the operations. Furthermore, if edit operations conflict with each other, the server cannot resolve these conflicts. Instead, the clients have to compute the conflict resolving OT algorithms themselves. By definition, journaling based approaches are not oblivious. Even if snapshots are used, the revision history can be gathered up to the point where the last snapshot was created.

In 2010, Feldman et al. proposed a collaboration tool [12] that invented the journaling approach. To the best of our knowledge, this work was the first one to combine encryption, integrity protection, and OT in a single application. However, their solution requires a custom server implementation. This disadvantage was compensated by a proposal by Clear et al. in 2012 [8]. In their tool, the custom server implementation is replaced by Google Docs. The client is built using a combination of a Mozilla Firefox extension and a standalone Java module for cryptographic operations.

Document Approach to Encrypted Collaboration. The second approach does not protect the edit operations, but the document itself. We call this the *document approach*. With this approach, a client does not need to download the document's history; the server can provide a client with the most recent snapshot directly. This approach also offloads the computationally expensive conflict resolution to the server. Since this approach does not require retaining the history of a document, it allows to gain obliviousness. However, this approach has other problematic sides. Modern cryptographic modes of operation are not incremental, for instance. Therefore, small changes in the plaintext lead to huge changes in the corresponding ciphertext. This results in an increase in network traffic and a certain flurry in the document if they are used for collaborative editing.

In 2010, D'Angelo et al. as well as Adkinson-Orellana et al. created Mozilla Firefox extensions to create an encryption wrapper for documents stored within Google Docs [1, 9]. But these publications focused neither collaboration nor real-time usability.

The first collaborative solution with a document based approach was presented by Huang and Evans in 2011 [17]. For this, they made use of the OT algorithms already implemented in Google Docs and therefore did not need to implement OT within their client, also a Mozilla Firefox extension. The problem with inefficient re-encryption using well-known modes of operation was solved using Incremental Encryption (cf. [4]) and specially crafted data structures. However, the tool by Huang and Evans does not work if multiple users edit a document simultaneously (cf. [17, sect. VII.A.]). To the best of our knowledge there is no solution in the literature that realizes encrypted real-time collaboration using the document approach.

Editing Structured Documents. All aforementioned publications work on whole documents; they do not consider protecting only parts of them. A publication by Zhang et al. [37] proposes to split documents into parts and to encrypt them separately. The intent for this is reducing load on light-weight devices and avoiding conflicts since they do not use OT. Furthermore, those publications that mention the concrete implementation of OT algorithms, use ones that were designed for unstructured documents (e. g., documents are treated as large arrays of characters).

Client Software. Interestingly, the majority of publications concerning collaboration on encrypted documents used custom browser extensions (or scripts for an existing extension like Michalas and Bakopoulos [27]) to realize their functionality. Apart from custom client implementations, only Feldman et al. mention a standalone web application that requires Java as client for their tool.

Maliciously Behaving Servers. Several publications assume that a server may not be honest but act maliciously by tampering with the client- or server-side code. Venus [32], SUNDR [22], and Depot [23] guarantee integrity and fault tolerance in this scenario. Mylar [30] provides confidentiality on the server by using searchable encryption based on bilinear maps and verifies the client-side code using digital signatures in a browser extension. However, several system design flaws of this tool were found recently [15].

[1]Google Docs allows a user to access the history of a document, see e. g. http://features.jsomers.net/how-i-reverse-engineered-google-docs/

[2]DropBox allows reverting to previous snapshots of files, see e. g. https://www.dropbox.com/en/help/11

6. DESIGN GOALS

From the literature, we derived a set of six properties for a collaborative editor that we use as design goals:

A. **Confidentiality.** Motivated by unprotected systems like Google Docs or Microsoft Office Online, we require to use proper encryption for reaching end-to-end confidentiality. This enables using untrusted storage servers that are honest-but-curious. The main challenge here is to cover all necessary cryptographic primitives and operations as-well-as the key management without any extra software on the client.

B. **Real-Time Collaboration.** Interactive collaboration requires all users that open a document to see edits of other users with little to no delay. As outlined before, OT algorithms enable this functionality while providing all users a consistent view onto a document.

C. **Software-as-a-Service (SaaS).** The success of the SaaS delivery model shows that users demand applications that can be used without having to install a software component first. Ideally, a collaborative editor should be available as a web application since that enables using it on different platforms, including mobile, and makes it independent of the underlying operating system.

D. **Flexibility.** While previous approaches require all users editing a document collaboratively to have access to the same symmetric key, we want to go one step further. With SECRET, documents may be edited by multiple user-groups with different access rights. Sections of documents should be protected in a fine granular manner for a specific user, a user-group, or any combination of these. Only users that are allowed to view and edit the particular section of a document can get access to the corresponding key.

E. **Obliviousness.** A collaborative editor should hide the revision history of a document or a section even to someone knowing the corresponding secret key. Motivation for this property is that a new user joining the collaboration should not be able to learn about the edits made in the course of creating the document.

F. **Efficiency.** We require SECRET to be usable in low bandwidth scenarios, such as mobile devices. Therefore, the network traffic after initial loading the editor page must not exceed typical mobile bandwidth (32 Kbit/s).

We also discuss **integrity** to prevent unwanted document modifications (see Appendix A).

7. SECRET

With SECRET, we show that the properties discussed in Section 6 can be realized. For *confidentiality*, we refer to Section 9.

7.1 Real-Time Collaboration

Each user enters data via the SECRET editor (see left side in Figure 1). Edits are processed once a second by the SECRET core that orchestrates the functionality: Modified elements are encrypted and formatted as XML Encrypted Data elements. These Encrypted Data elements are synchronized with the untrusted storage server via our XML-enabled variant of *ShareJS*.

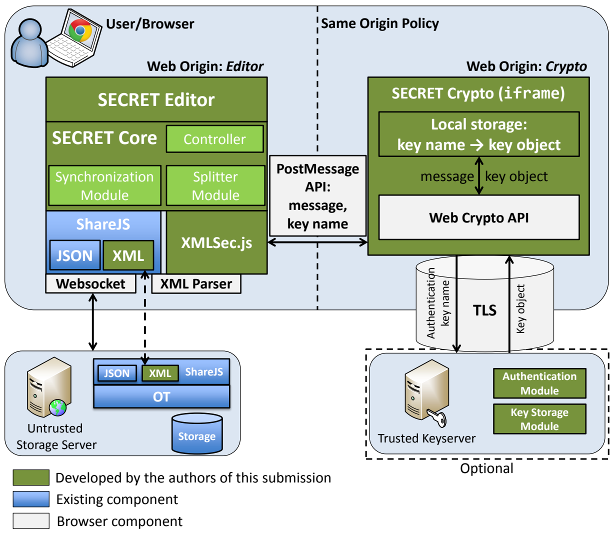


Figure 1: SECRET's detailed architecture overview.

7.2 Software-as-a-Service (SaaS)

SECRET must be easily available without any extra software or installation process. Therefore, we implemented it as a web application using CoffeeScript and Node.js. In order to use it, only a recent browser is required. SECRET has been tested with Google Chrome 50.

7.3 Flexibility

The flexibility to protect sections of documents respecting fine granular access rights can be achieved since we only consider *structured* documents and apply our new approach combining tree-based OT with structure-preserving encryption. We treat subtrees of the document tree as our basic units for OT, and replace them by ciphertext elements in the stored document while maintaining the document structure. This concept is depicted in Figure 2. Since the editor we use makes sure that an edit affects exactly one subtree, each edit operation will result in one subtree being sent to the server. Thus each edit operation will only affect one ciphertext element.

The fine granular access rights in SECRET require distributing a set of keys to every user. A classical approach to this problem is using a Public Key Infrastructure (PKI) where each user holds a key pair. In such an approach, the keys for document sections would be wrapped using a user's public key and could be integrated in the document structure. However, the PKI approach has several drawbacks:

1. Key objects created by the WebCrypto API like a user's private key are not persistent. For key storage, the WebCrypto API refers to the Indexed Database API [24]. However, private keys stored in such a database are prone to be deleted by mistake (e. g., if the *Clear Browser Data* feature is used).

2. Private keys are not distributed. A user that wants to continue editing a document with a different device would need to import her private key to that device. However, if the key is protected (`export` set to `false`), this is impossible.

3. Public keys are not distributed. If user A wants to give user B access to a document, A would need to know B's

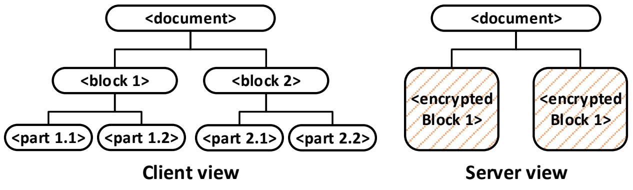


Figure 2: Views of clients and untrusted storage server.

public key first. To solve this, a directory for public keys would be required.

To cope with this problem, we offer two different solutions to fit different use-cases (cf. Section 8.1). Both approaches allow relying solely on symmetric cryptographic primitives without being affected from the drawbacks of a PKI. Keys are either requested from a keyserver by indicating the *key name* given in the encrypted document, or derived from a password using the WebCrypto API. In both cases, the symmetric keys are then stored in the local storage of a SECRET crypto module, where they are protected from unauthorized access. The crypto module itself is bound to particular web origin (named *Crypto* on the right side of Figure 1) and thus protected by the Same-Origin-Policy from illegitimate access.

7.4 Obliviousness

Obliviousness cannot be achieved with the journaling approach to encrypted collaboration. Therefore, we choose the document approach to realize SECRET.

7.5 Efficiency

For a flexible document editing solution with fine granular access control for groups having access only to some document parts, encrypting larger document parts (e. g., sections or pages) is an obvious solution. In case of multiple users issuing frequent edits to a document however, this approach is inefficient since even a small change requires re-encryption and transmission of the whole encrypted part. This problem was already identified by Huang and Evans [17] and was tackled by using an uncommon encryption mode of operation called *RPC* (cf. [4]) that supports efficient incremental updates of ciphertexts. This mode of operation is however not supported by the WebCrypto API, which only offers CTR, CBC, CFB, and GCM modes of operation. In our scenario, all these modes of operation have a major downside: A small change (e. g., inserting a single character) to the content leads to a new ciphertext, which has to be synchronized.

In contrast to Huang and Evans, we rely on the structure of a document rather than treating it as a large array of characters. We keep efficiency by encrypting smaller parts of the document using the well-known GCM mode of operation. This way, we also keep SECRET's bandwidth requirement low enough to use it with a slow mobile connection.

8. SECRET'S ARCHITECTURE

A detailed architecture overview on SECRET is depicted in Figure 1. In general, there are three different entities taking part:

1. An *optional trusted keyserver* managing the symmetric keys used to encrypt/decrypt blocks of the document. If no keyserver is used, keys are managed by deriving them from passwords.
2. An *untrusted storage server*, for example, a cloud service, that hosts the web application, stores the encrypted document, and provides a graphical editor to visualize and edit the document.
3. Users who have access to different parts of the document. Their access to the document is restricted by the access to the key that is necessary to decrypt the document part.

8.1 Key Management

SECRET's key management consists of two parts. On the one side, there is a *key concept* that allows to encrypt different document parts with different keys. On the other side, there is SECRET's crypto module that executes all cryptographic operations and locally caches keys.

Key Concept with Keyserver. Our first key concept variant makes use of a keyserver. On a quick peek, one might think, that using a keyserver simply moves the trust from one party to another one (e. g., in comparison to unencrypted Google Docs). This is true, when considering individual end-users for SECRET. But a keyserver is an important and desired goal in business use-cases: here, a company only maintains a key server, which can be a small and easy-to-backup server included into an isolated network environment. All company members have access to it and can thus use SECRET. The encrypted working data itself can be hosted on an untrusted storage server at low cost and high availability (Google, Amazon, . . .).

SECRET's key concept differentiates two types of symmetric keys: Group Keys (GKs) and Block Keys (BKs). GKs are accessible by a group of users. They are stored on the keyserver and provided to authorized users. Each GK is labeled with a unique ID that is used to identify the key when requested by the browser.

BKs are used to encrypt a specific part of the document encapsulated in blocks. If a group is meant to have access to the block, its BK is wrapped (encrypted) using the GK of the group. This wrapped key is stored in the XML structure of the block, together with the ID as information to which group it belongs to. If multiple groups have access to a block, the (identical) BK is wrapped using all corresponding GKs and deposited within the block (see Figure 3).

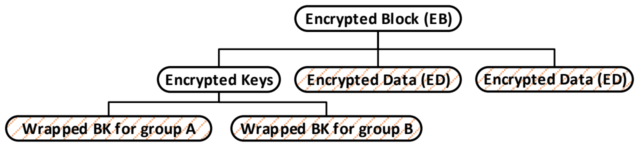


Figure 3: Components included in each Encrypted Block.

Key Concept *without* Keyserver. A different use-case exists if end-users not related to a specific company want to use SECRET. In this case, a keyserver does not make sense, because the end-user would have to trust it. Because of this, we designed a key concept without the usage of a keyserver.

This is realized by using a Password-Based Key Derivation Function (PBKDF) and works as follows: Instead of enforcing a user to authenticate to a keyserver (e. g., with username/password), we can use a password to derive the GK. For this purpose, the WebCrypto API offers a `crypto.deriveKey()` function that can trigger PBKDF2.

The password is then a secret that has to be shared between the parties, for example, via email or instant messaging.

SECRET's Crypto Module. The crypto module executes *all* cryptographic operations. It works as an encryption/decryption oracle for the editor component. In other words, the editor can *ask* the oracle to encrypt or decrypt a specific block. If the user is granted to do so, for instance, if he is able to unwrap the BK, the crypto module executes the operation. It also offers generating fresh BKs and wrapping those using existing GKs.

Results are returned to the editor by using the PostMessage API [35] enabling the communication between different origins within the browser. The purpose of this design is to ensure, that GKs never leave the crypto module and that BKs are only exported in encrypted form. All communication between the editor page and the crypto module uses well-defined interfaces. Technically, we designed the crypto module as an `iframe` element that is embedded in the editor component. The web origin [2] of the crypto module is set to a different domain (e. g., to the keyserver's domain), so that the DOM access from the editor is restricted by the Same Origin Policy.

The connection of SECRET's crypto module to the keyserver requires a secure channel. This can be achieved by using TLS with mutual authentication via client certificates, and by manually registering each client's certificate at the keyserver. After the user's authentication with the keyserver, the crypto module can request keys to which the user has access to.

8.2 Untrusted Storage Server

The untrusted storage server is a publicly available web application providing three main features:

Editor. A graphical editor delivered to the client and executed by the browser. It enables the visualization and editing of the document in the browser.

OT with ShareJS. The server component of ShareJS keeps a list of operations that were sent by clients in a database. After receiving a new operation from a client, the server checks the snapshot of the document to which the client refers to and whether there are other operations that are conflicting with the new one. If that is the case, the server applies OT algorithms to generate operations shifting every connected client to the latest snapshot of the document based on the snapshot of the document the server suspects a client to have. This process is complex and computationally expensive. We did not modify the ShareJS server component, but added our XML API (cf. Section 4).

To identify a component of the document (i. e. an element, attribute, text, etc.) to operate on, the server expects a path to that component. A path is technically a list of zero-indexed integers and names that identify child nodes below each respective element. For example, consider the following document: `<root><a></a>text<x><b>456</b></x></root>`. A path to identify the `b` element is `[2,0]` since the `x` element is the third node below `root` and `b` is the first node below `x`. Paths can even identify single characters in text nodes; the character `5` in the text node `456` has the path `[2,0,0,1]`.

Storage. For each document, the untrusted storage server stores a gzip compressed file containing the latest snapshot and a list of recent operations related to these documents in a database. Compressing XML documents is a common technique (e. g., for office files). Note that the untrusted storage server has no access to any key and therefore cannot decrypt protected parts.

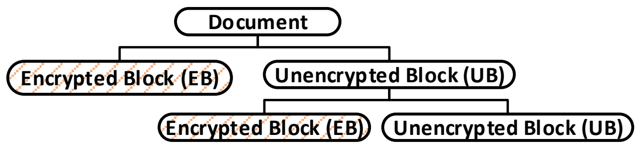


Figure 4: Example document containing encrypted and unencrypted blocks.

8.3 SECRET in the Browser

SECRET consists of different components executed in the user's browser. Figure 6 shows a screenshot of our proof-of-concept implementation that we developed and evaluated.

8.3.1 SECRET Editor

If multiple users simultaneously edit a document, each client has to handle two types of events: (1.) the user itself generates *local edits* of the document and (2.) the untrusted storage server provides *external edit* operations executed by other users to the client.

Local Edits. For local edits, SECRET uses a combination of editable HTML elements (i. e. `contenteditable` is set to `true`) and Mutation Observers [36]. Every second the editor checks whether the user modified text contents. If that is the case, the implementation fetches the XML element in which the edit took place and extracts the modification. Then, this modification is forwarded to the Controller which handles the rest of the processing (see below).

External Edits. Whenever an operation sent by the untrusted storage server is ready to be displayed, the Controller forwards it to the editor. The editor is then responsible for identifying the corresponding UI component and applying the operation to it. There have been some technical challenges in the implementation, which we outline in Section 8.3.6.

8.3.2 SECRET Controller

The Controller module has several interfaces to the other modules of SECRET in the browser, which are explained in the following sections. The job of the Controller is to route edit events and messages from the untrusted storage server to the responsible modules.

Initially, the browser receives a document snapshot, which can contain multiple Unencrypted Blocks (UBs) and Encrypted Blocks (EBs) as shown in Figure 4. Note that SECRET supports any mix of UBs and EBs. Particularly, a snapshot may contain only a single EB for protecting the whole document as well as only UBs leaving the document unprotected. The UBs can be directly displayed in the editor, but the EBs have to be further processed: Next to the required key, an EB contains multiple Encrypted Datas (EDs) (see Figure 3).

An ED element contains an XML tree compatible to the XML Encryption specification [16]. Thus, it contains meta-data information like key references, information about the used cryptographic algorithms, etc. Usually, there are multiple ED elements within one block, which we added for performance reasons. Further details on this are presented in Section 8.3.4. Once the ED is decrypted, the browser puts all blocks together and hands them over to the editor displaying the content.

If the Controller receives a modification from the editor, it forwards the modification to the other modules, which are explained below. Lastly, after the local processing is completed, the Controller triggers the creation of a ShareJS operation that is sent to the untrusted storage server.

8.3.3 Synchronization Module

Our system stores three different document views in the browser: the encrypted document, the decrypted document, and the content displayed in the editor. Edit events in the editor trigger modifications in the decrypted document as well as in the encrypted one. An edit may have happened in either an UB or an EB. Therefore, the synchronization module keeps track of whether a block is an UB or EB. This is used in order to recognize which block is affected and probably needs to be re-encrypted. A similar problem arises in the opposite direction: Since an operation that modifies a ciphertext does not tell any information about the underlying edit (i. e. the plaintext element that was edited), it is difficult to tell where to update the user interface. This problem is solved by the synchronization module which artificially creates a one to one correlation between plain elements and their encrypted counterparts.

8.3.4 Splitter Module

If an encrypted document contains large blocks of text without any further structure, AES-GCM becomes inefficient as outlined in Section 7.5. To prevent this inefficiency, the splitter module artificially splits such text blocks in smaller parts (by using multiple `<span>...</span>` elements) that are encrypted individually. Each encrypted `<span>` is stored in a separate *Encrypted Data* element, see Figure 3. The splitter module also removes `<span>` elements if the user deletes their content. However, it is unclear how big or small these parts of a document have to be to have an optimal balance between network and storage overhead. This problem is evaluated in detail in Section 10.

8.3.5 XMLSec.js

Our XMLSec library creates the necessary XML Encryption data structures. It uses our SECRET crypto module as cryptographic oracle. The communication between the crypto module and the XMLSec library is done via the PostMessage API. In the end, the XMLSec library makes use of the WebCrypto API to provide encryption and decryption functionality.

8.3.6 Implementation Challenges

During the development of SECRET we stumbled upon interesting implementation challenges. We outline three of these here to highlight the gap between *specifying the architecture of a system* and *implementing* it. This might help developers and researchers in their future projects.

Cursor Placement with Splitting. Consider the user having entered the string `1234`, resulting in the following document fragment: `<span>1234</span>`. The cursor is placed within the `<span>` behind the `4`. If we assume that the implementation is configured to use a split size of 4, then entering one more character (e. g., `5`) creates the following document fragment: `<span>1234</span><span>5</span>` which is displayed to the user as `12345`. However, the browser keeps the cursor in the first `<span>` element, so that entering a new character is prepended (instead of appended) to the `5`. The editor displays `1234`*`6`*`5`. We solved this by programmatically moving the cursor after splitting into the correct `<span>` element by using the JavaScript *Range* API (`createRange()`, `setStart()`, . . .).

Synchronization and Deletion. Another problem becomes apparent if a user deletes text in a `<div>` element with `contenteditable` set to true. Consider a user editing the document fragment `<span>1234</span><span>5</span>`, placing the cursor behind the `5`, and pressing *backspace* to delete a character. Intuitively one would expect the second `<span>` to be empty and the cursor being placed in it. But this is not the case. Instead, Google Chrome removes the second `<span>` completely and places the cursor at the end of the previous span. This is a huge problem since SECRET is unable to detect a *Local Edit*, because the second `<span>` is deleted. To solve this problem, we use Mutation Observers [36]. They allow – besides other operations – to get notified if an object is deleted (e. g., the respective `<span>` element), and we are thus notified that an edit appears.

Preventing Edit Loops. A technical challenge is to make sure that applying an external edit to the editor is not mistaken for a local edit: suppose that user A receives an edit operation executed by user B. If it is accidentally handled as a *Local Edit*, it is executed and then again send to the untrusted storage server, which then sends the operation back to user B and so forth. This will result in an endless loop of edit events and render the system unusable.

To resolve this, our editor caches the content that an element is supposed to contain (based on the decrypted document maintained by the Synchronization Module) in a custom cache property in the DOM object. If this cache and the actual content differ, then the user must have edited the content. If an external edit deleted an element, we set a custom flag on the corresponding HTML element before deleting it. This helps discarding the mutation event that is fired after the deletion.

9. SECURITY DISCUSSION

In this section, we analyze SECRET in three models.

Honest-But-Curious Cloud Server. A cloud server S – the adversary – only sees c_j^t for any session $\sigma_j = (*, S, *)$. He is able to learn the structure of the document, consisting of plaintext and ciphertext nodes, but he learns nothing about the plaintext contents of a ciphertext node. A client may simply exchange such a node with a totally different ciphertext node of equal length, and the adversary is not able (due to the IND-CPA property of the used encryption mode AES128-GCM) to distinguish between these two nodes.

Passive Man-in-the-Middle. For a passive Man-in-the-Middle attacker, the same argument holds: by passively observing all network traffic, this adversary may be able to reconstruct the document structure, but due to the IND-CPA property of the encryption, he is not able to learn the content of the ciphertext nodes.

Web Attacker Model. It is not possible to "prove" a complex web application like SECRET secure in the web attacker model – even Google and Facebook suffer from time to time from vulnerabilities under this attacker model. However, the design rationales for SECRET can be given here.

The client application of SECRET is conceptually divided into two components, loaded from different web origins (cf. Figure 1). We assume that the session key k_j is only available in the SECRET Crypto component (right). This component is configured to accept PostMessageAPI calls only from a well-known web origin. Thus if the adversary simply copies the left component to his own malicious server, this copy is unable to establish a connection to the SECRET crypto component, and thus remains unable to decrypt ciphertext.

Attack on Web Origins. If we allow the adversary to manipulate web origins [2], SECRET is, like any other web application, no longer secure.

1. The attacker builds his own malicious web application, by simply copying the left part (cf. Figure 1) of the SECRET code (this part is public and does not contain the session key) and adding a new function `transferCleartext()`.
2. The attacker tricks the victim to access this malicious application, e. g., by DNS Spoofing, and forges the web origin with the same techniques. He loads the right part of the code (including the secure key management) from the original web origin.
3. The malicious application can now systematically load encrypted documents for all sessions in which the user participates, decrypt them, and send the plaintexts to the adversary. SECRET's crypto module will cooperate with the malicious application, since it has the correct (but forged) web origin.

The same attack holds for Cross-Site-Scripting (XSS). Here, the attacker can inject and execute a malicious JavaScript into so benign SECRET editor. This script then runs in the web origin of the original editor. The security of SECRET can be increased against XSS by using further techniques (e. g., Content-Security-Policy), which is out-of-scope.

Information Leakage. Information about the structure of a document may leak. We are aware of this issue, and have included it in the formal model (cf. Section 3). We consider it a design feature that is appreciated by applications.

10. PERFORMANCE EVALUATION

Using encryption for collaborative editing leads to two problems: storage overhead and network overhead. Consequentially, the question rises how large this overhead is for SECRET and if our system is usable despite these overheads. Additionally, the splitter module SECRET provides is meant to optimize the performance and reduce the overheads. This is evaluated in the following in order to find a reasonable boundary defining when to split.

10.1 Evaluation Setup

To simulate real edits, we used Google Chrome 50 with Selenium 2.49.1[3]. This enables the automated invocation of URLs, webpage navigation and performance measurement in the browser. We use Selenium's *WebDriver* API to start a browser and to navigate to the untrusted storage server and start SECRET.

[3]http://docs.seleniumhq.org/

Tests. Once the editor is loaded we generate random strings with different lengths (64, 128, 512, 1024, 2048, and 4096 bytes) and simulate keystrokes typing each string. We insert a character every 300 milliseconds, which results in 200 key strokes per minute. This is a typical speed of an experienced typist, for example, an office assistant.

During the tests we use two typing areas: one ends up in an encrypted block, the other one in an unencrypted block. We issue the same key strokes to each area. The splitting into separate parts is carried out in both areas. We repeat all tests for different split sizes (32, 64, 128, 256, and 512 bytes).

Measurement. Selenium allows injecting JavaScript code into a loaded website and accessing defined variables. Thus, we are able to get the size of the encrypted and unencrypted document during testing. In addition, we get access to the operations sent to the server. Thus, we can measure the network traffic triggered by editing the document.

10.2 Storage Overhead

In Figure 5a, the document size of an unencrypted document with random characters in dependence on the content length is presented. Independent from the fact that the content is not encrypted, the content is split. This splitting leads to an overhead since each part of the content is encapsulated in a `<span>...</span>` element. As a result, the more the content gets split, the more `<span>` elements are added causing storage overhead. Observing Figure 5a, one can see that compression is very effective for our structure. Comparing the same document with split size 32 and with split size 512 after compression shows only 5.7% size increase despite it contains 16 times the number of `<span>` elements.

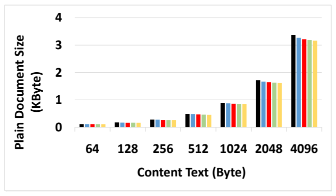
Figure 5b shows the same comparison for an encrypted document. The values in this case are much larger than in the unencrypted case. This has several reasons: First, the ciphertext is binary data that has to be Base64 encoded before it can be stored in XML, This alone results in an expansion factor of 1.33. Second, each split operation leads to the creation of a new ED element. Since an ED element contains metadata information like a random ID, the resulting encrypted document's size is further increased.

Split size	32	64	128	256	512
Storage expansion	3.50	2.46	1.92	1.66	1.53

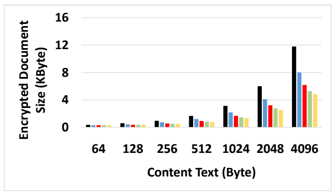
Table 2: Ciphertext expansion of a 4096 byte document.

As Table 2 shows, avoiding small split sizes alleviates the overhead. For example, a document with 4096 bytes of plain content with a split size of 128 byte has an encryption overhead factor of 1.92 (= 92% overhead). This number look high, but in fact it is three times better in comparison to previous work. For example, Clear et al. [8] achieve an expansion factor of 4.82 (= 382% overhead), Huang and Evans [17] get 3.75 (= 275% overhead).

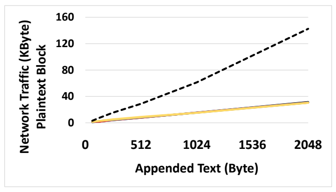
Summarized, the larger the split size gets, the less storage overhead it becomes. This is an obvious result since each split operation leads to additional overhead. However, another side effect needs consideration: the communication overhead. This overhead is directly related to the content, which has to be synchronized.



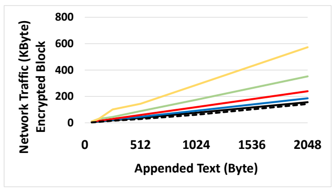
(a) Storage requirements for unencrypted documents.



(b) Storage requirements for encrypted documents.



(c) Network overhead in an unencrypted document.



(d) Network overhead in the encrypted document.

Figure 5: SECRET evaluation results. Larger pictures can be found in Figure 8 (Appendix).

10.3 Network Overhead

We evaluated the network communication overhead by inserting the same text with different lengths in the unencrypted and encrypted area. During typing, we collected all operations transmitted to the untrusted storage server and measured the transmitted bytes. In order to compare SECRET with a well-spread collaboration platform, we repeated the same edits on Google Docs and measured the network overhead.

10.3.1 Unencrypted Network Overhead

Figure 5c shows the network communication (in bytes) caused by edits in an unencrypted document in dependence on the appended text length and chosen split size. There is a linear relation between the inserted amount of text and the network traffic. The reason for the growth in traffic is the addition of new `<span>` elements and the protocol overhead of ShareJS. Even for the smaller split size values, the network utilization is almost identical.

In order to compare SECRET with Google Docs we observed the communication between our browser and Google during editing a freshly generated document. Google Docs synchronizes the content via HTTP POST requests transmitting a parameter named `bundles` that contains OT information. For our measurement, we extracted only this HTTP parameter and left out all other parameters[4].

Compared to Google Docs, SECRET needs less communication to synchronize the unencrypted document content. The main reason for this difference is the meta information that Google Docs transmits within each OT synchronization request (e. g., session management information). Some of this information is even redundant since it is sent simultaneously as a GET and POST parameter.

10.3.2 Encrypted Network Overhead

In Figure 5d the network overhead of editing an encrypted block in SECRET is displayed. With encryption enabled, almost every key stroke causes a re-encryption of an existing ED element, which is then sent over the network. This leads to a substantial growth of network traffic.

The traffic utilization of SECRET is always above the one of Google Docs[5], regardless of the chosen split size. This is an expected result caused by the enabled encryption. One can see that SECRET has a linear relation between the inserted text and network traffic, similar to the unencrypted Google Docs.

[4]Since we did not reverse-engineer Google Docs, it is possible that our measurement technique misses some OT related messages. However, these messages would only increase the bandwidth required by Google Docs.

[5]Since no related work measured network traffic utilization, we cannot compare our solution with other encrypted approaches.

10.4 Bandwidth Requirements

Despite the overhead of our approach, the system is usable: At a typing speed of 200 key strokes per minute, entering a 1024 byte document takes 307 seconds. With encryption and a split size of 128 bytes, the total payload traffic sent by the client is roughly 119 Kbyte. However, the traffic being transmitted is larger. The two main reasons for this is URL encoding that is applied to the messages and HTTP headers being added to the messages.

We measured the required traffic bandwidth of SECRET using the network analysis tool *Wireshark*. Using the settings given above, our tool requires a bandwidth of 14 kbit/s (see Figure 7).

If other users of the system type at the same speed, then each of them generates incoming traffic at a similar rate. Considering that even very slow mobile connections provide a connection speed of 32 Kbit/s or more, it would require at least a couple of users typing simultaneously to exceed such a connection.

Summarized, a split size value of 128 bytes seems to be a good trade-off between the storage overhead and the network overhead. With this setting, SECRET requires a storage overhead of factor 1.92 and a network utilization that does not exceed even very slow connections.

SECRET is the first considering network bandwidth in addition to storage overhead in its evaluation.

11. CONCLUSION AND OUTLOOK

We presented SECRET, the first fully collaborative web editor on encrypted data with obliviousness using a document based approach. By that, we have answered a research question that has been open for the last four years. Our extensive evaluation reveals that – with the right choice of parameters – SECRET achieves three times lesser storage overhead compared to all existing solutions. SECRET is frugal with network bandwidth and could also be applied to mobile low-bandwidth application areas.

Although SECRET is fully working, there are still open research questions. Most notably, whether it is possible to include SECRET's concept to full-fledged office documents such as Microsoft Office, LibreOffice, or OpenOffice; maybe by integrating it into their online collaboration services. To help the research community in this field, we published SECRET and its novel structure preserving encryption concept using OTs as open source software.

12. ACKNOWLEDGMENTS

This research results from work in the research projects *SyncEnc* and *VERTRAG*, which are funded by the German Federal Ministry of Education and Research (BMBF, FKZ: 16KIS0412K and 13N13097), as well as the *FutureTrust* project funded by the European Commission (grant 700542-Future-Trust-H2020-DS-2015-1).

References

[1] L. Adkinson-Orellana, D. A. Rodríguez-Silva, F. Gil-Castiñeira, and J. C. Burguillo-Rial. Privacy for google docs: Implementing a transparent encryption layer. In *CloudViews*, pages 20–21, 2010.

[2] A. Barth. The Web Origin Concept. RFC 6454 (Proposed Standard), Dec. 2011. URL http://www.ietf.org/rfc/rfc6454.txt.

[3] A. Barth, C. Jackson, and J. C. Mitchell. Securing frame communication in browsers. In *USENIX Security*, 2008.

[4] M. Bellare, O. Goldreich, and S. Goldwasser. Incremental cryptography: The case of hashing and signing. In *CRYPTO*, pages 216–233. Springer, 1994.

[5] T. Bray. The JavaScript Object Notation (JSON) Data Interchange Format. RFC 7159 (Proposed Standard), Mar. 2014. URL http://www.ietf.org/rfc/rfc7159.txt.

[6] T. Bray, F. Yergeau, E. Maler, J. Paoli, and M. Sperberg-McQueen. Extensible markup language (XML) 1.0 (fifth edition). W3C recommendation, W3C, Nov. 2008.

[7] E. Buonanno, J. Katz, and M. Yung. Incremental unforgeable encryption. In *FSE*, pages 109–124. Springer, 2001.

[8] M. Clear, K. Reid, D. Ennis, A. Hughes, and H. Tewari. Collaboration-preserving authenticated encryption for operational transformation systems. In *ISC*, pages 204–223. Springer, 2012.

[9] G. D'Angelo, F. Vitali, and S. Zacchiroli. Content cloaking: preserving privacy with google docs and other web applications. In *SAC*, pages 826–830. ACM, 2010.

[10] A. H. Davis, C. Sun, and J. Lu. Generalizing operational transformation to the standard general markup language. In *CSCW*, pages 58–67. ACM, 2002.

[11] C. A. Ellis and S. J. Gibbs. Concurrency control in groupware systems. In *SIGMOD*, volume 18, pages 399–407. ACM, 1989.

[12] A. J. Feldman, W. P. Zeller, M. J. Freedman, and E. W. Felten. SPORC: Group collaboration using untrusted cloud resources. In *OSDI*, pages 337–350, 2010.

[13] I. Fette and A. Melnikov. The WebSocket Protocol. RFC 6455 (Proposed Standard), Dec. 2011. URL http://www.ietf.org/rfc/rfc6455.txt.

[14] J. Gentle, N. Smith, and Others. ShareJS. https://github.com/share/ShareJS/tree/0.6. (Retrieved: October 2016).

[15] P. Grubbs, R. McPherson, M. Naveed, T. Ristenpart, and V. Shmatikov. Breaking web applications built on top of encrypted data. In *CCS*, pages 1353–1364. ACM, 2016.

[16] F. Hirsch, T. Roessler, J. Reagle, and D. Eastlake. XML encryption syntax and processing version 1.1. W3C recommendation, W3C, Apr. 2013.

[17] Y. Huang and D. Evans. Private editing using untrusted cloud services. In *ICDCSW*, pages 263–272. IEEE, 2011.

[18] C.-L. Ignat and G. Oster. Peer-to-peer collaboration over xml documents. In *CDVE*. Springer, 2008.

[19] C. L. Ignat, G. Oster, et al. Flexible reconciliation of xml documents in asynchronous editing. In *ICEIS*, pages 359–368, 2007.

[20] M. Jones and J. Hildebrand. JSON Web Encryption (JWE). RFC 7516 (Proposed Standard), May 2015. URL http://www.ietf.org/rfc/rfc7516.txt.

[21] H. Krawczyk, M. Bellare, and R. Canetti. HMAC: Keyed-Hashing for Message Authentication. RFC 2104 (Informational), Feb. 1997. URL http://www.ietf.org/rfc/rfc2104.txt. Updated by RFC 6151.

[22] J. Li, M. N. Krohn, D. Mazières, and D. Shasha. Secure untrusted data repository (SUNDR). In *OSDI*, page 9, 2004.

[23] P. Mahajan, S. Setty, S. Lee, A. Clement, L. Alvisi, M. Dahlin, and M. Walfish. Depot: Cloud storage with minimal trust. In *TOCS*. ACM, 2011.

[24] N. Mehta, J. Sicking, E. Graff, A. Popescu, J. Orlow, and J. Bell. Indexed database API. Recommendation, W3C, Jan. 2015.

[25] R. C. Merkle. A digital signature based on a conventional encryption function. In *CRYPTO*, pages 369–378. Springer, 1987.

[26] D. Micciancio. Oblivious data structures: applications to cryptography. In *STOC*, pages 456–464. ACM, 1997.

[27] A. Michalas and M. Bakopoulos. SecGOD Google Docs: Now I feel safer! In *ICITST*. IEEE, 2012.

[28] D. A. Nichols, P. Curtis, M. Dixon, and J. Lamping. High-latency, low-bandwidth windowing in the jupiter collaboration system. In *UIST*. ACM, 1995.

[29] G. Oster, H. Skaf-Molli, P. Molli, H. Naja-Jazzar, et al. Supporting collaborative writing of xml documents. In *ICEIS*, pages 335–341, 2007.

[30] R. A. Popa, E. Stark, S. Valdez, J. Helfer, N. Zeldovich, and H. Balakrishnan. Building web applications on top of encrypted data using mylar. In *NSDI*, pages 157–172, 2014.

[31] M. Ressel, D. Nitsche-Ruhland, and R. Gunzenhäuser. An integrating, transformation-oriented approach to concurrency control and undo in group editors. In *CSCW*, pages 288–297. ACM, 1996.

[32] A. Shraer, C. Cachin, A. Cidon, I. Keidar, Y. Michalevsky, and D. Shaket. Venus: Verification for untrusted cloud storage. In *CCSW*. ACM, 2010.

[33] C. Sun, X. Jia, Y. Zhang, Y. Yang, and D. Chen. Achieving convergence, causality preservation, and intention preservation in real-time cooperative editing systems. *TOCHI*, 5(1):63–108, 1998.

[34] M. Watson. Web cryptography API. W3C recommendation, Jan. 2017.

[35] WHATWG. Html – posting messages. Online, https://html.spec.whatwg.org/#posting-messages, October 2015.

[36] WHATWG. Dom – mutation observers. Online, https://dom.spec.whatwg.org/#mutation-observers, May 2016.

[37] C. Zhang, J. Jin, E.-C. Chang, and S. Mehrotra. Secure quasi-realtime collaborative editing over low-cost storage services. In *SDM*, pages 111–129. Springer, 2012.

APPENDIX

A. INTEGRITY PROTECTION

Ensuring the integrity of a whole document is easy: One can simply append a Message Authentication Code (e. g. an HMAC [21]) or use an authenticated encryption scheme like AES-GCM. This has been done before, for example, by Clear et al. [8]. However, if parts of a document are to be left unprotected intentionally, it is crucial to identify all parts which must not be left unprotected in order to keep integrity for the sensitive parts. The remainder of this section, we describe a concept to provide integrity we are currently working on.

A.1 Definition of Integrity

We assume, that encrypted parts are leafs in the tree, which holds for ED elements, and that the leafs are individually integrity protected by AES-GCM. This ensures that even an adversary changing a single bit in the ciphertext would be detected. However, if we furthermore assume an adversary that may insert, delete, or reorder parts of a document, we would not be able to detect if the adversary modified the document structure (e. g., by swapping ED elements without changing any bit within these elements). For that reason, we need to categorize nodes in the document tree.

Definition 1. *A* ***critical node*** *is a node in a document tree that has at least one critical node as child. All ED elements are critical nodes.*

The intuition behind this definition is as follows: If there is any encrypted data in a document tree, then all nodes on the path from the document root to that ED element are critical. If there are at least two ED elements in a document tree, then the critical nodes build a subtree of the document tree.

We define integrity of a partially encrypted document tree as follows:

Definition 2. ***Integrity*** *of a partially encrypted document tree is given if no attacker is able to efficiently modify the structure of the subtree formed by the critical nodes or the content of ED elements in it without being detected.*

Informally spoken, this definition prevents all unauthorized modifications of the document structure except of parts that consist entirely of unencrypted branches and leafs. Note that this definition only covers the tree structure. The content of a critical node may be modified without violating the integrity[6].

A.2 Technical Concept

For our integrity concept, we use the fact that every ED element in the document tree has a randomly generated unique ID in form of an XML attribute. These IDs are used as Additional Authenticated Data (AAD) in AES-GCM upon encryption of the content of that element. Every other critical node is also provided with an ID attribute. The value of that attribute is computed as the hash value of the ID attributes of all critical child nodes. This way, we build a Merkle-Tree [25] using the IDs to protect the structure. At the root node, we introduce another XML attribute named `IntegrityTag`. Its value is either an HMAC or a digital signature of the root's ID, depending on whether public verification of the integrity is required or not.

[6]This probably sounds undesirable. However, if the content is not meant to be modifiable, one should simply put the corresponding subtree into an ED element.

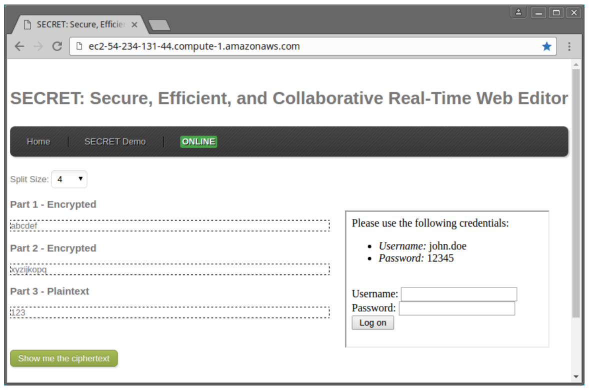
A.3 Security Consideration

If a collision-resistant hash function is used, the security properties of Merkle-Trees guarantee that every modification in the structure of critical nodes result in a different ID of at least the root node. Given that the `IntegrityTag` was build using a secure construction (i. e. an attacker cannot forge a valid value), every unauthorized modification of the structure can be detected.

There is a hypothetical modification an unauthorized attacker can perform that remains undetectable for some users of the system: An attacker could replace the ciphertext of an ED element with a different ciphertext (probably generated under the same key), but keeping the original ID. This modification is detected once a user tries to decrypt that ED element since different AAD were involved in the original encryption. However, all users who do not have access to the key for that ED element cannot detect this attack. They can only check the ED elements they have keys for and compute and compare the `IntegrityTag`, which succeeds since the attacker did not change any IDs or other ED elements.

Nevertheless, this issue is negligible since the modification is instantly detected by users who have the corresponding keys. Users, who do not have the corresponding keys, are not concerned of that particular ED element anyway since they cannot decrypt it.

B. ADDITIONAL FIGURES


SECRET: Secure, Efficie
ec2-54-234-131-44.compute-1.amazonaws.com

SECRET: Secure, Efficient, and Collaborative Real-Time Web Editor

Home | SECRET Demo | ONLINE

Split Size: 4

Part 1 - Encrypted
abcdef

Part 2 - Encrypted
xyzijkopq

Part 3 - Plaintext
123

Please use the following credentials:

- *Username:* john.doe
- *Password:* 12345

Username:
Password:
Log on

Show me the ciphertext

Figure 6: Screenshot of our proof-of-concept demo.

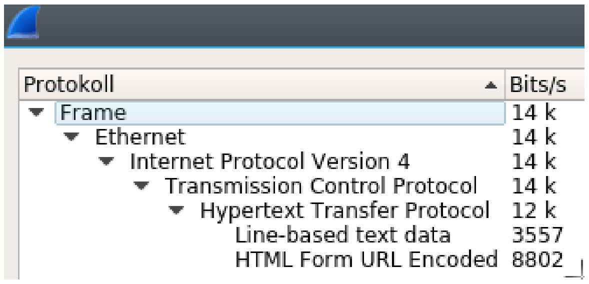
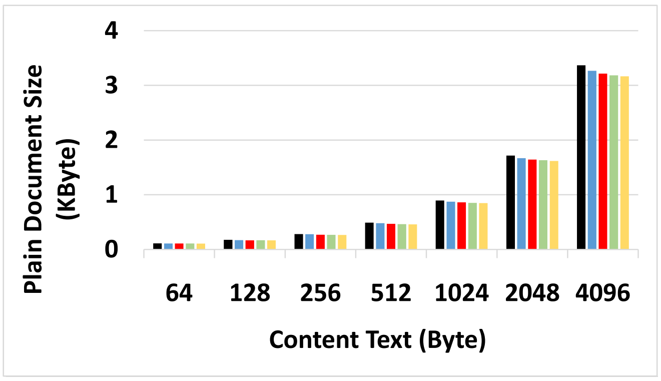
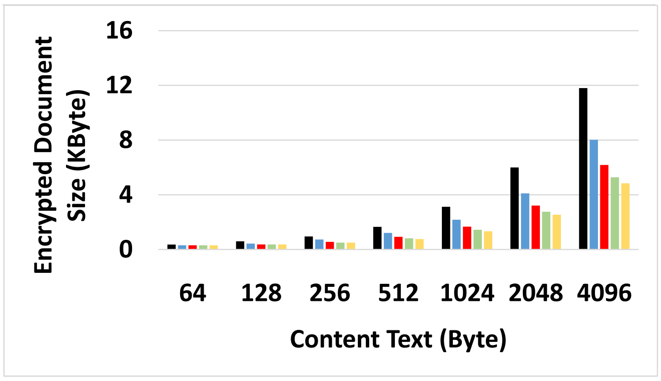

```
Protokoll                                        Bits/s
▼ Frame                                          14 k
   ▼ Ethernet                                    14 k
      ▼ Internet Protocol Version 4              14 k
         ▼ Transmission Control Protocol         14 k
            ▼ Hypertext Transfer Protocol        12 k
                 Line-based text data            3557
                 HTML Form URL Encoded           8802
```

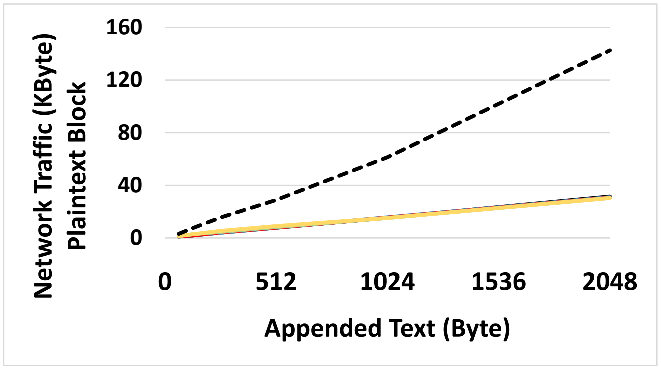
Figure 7: Screenshot from *Wireshark* measuring the required bandwidth for SECRET at 200 key strokes per minute with a split size of 128 bytes.



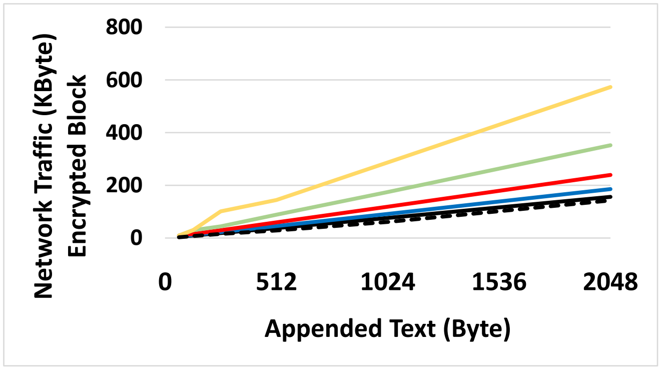
(a) Storage requirements for unencrypted documents.



(b) Storage requirements for encrypted documents.



(c) Network overhead in an unencrypted document.



(d) Network overhead in the encrypted document.

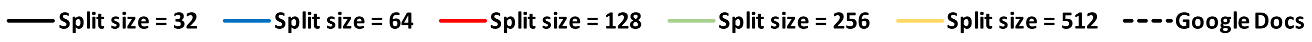


Figure 8: SECRET evaluation results.