# Content Cloaking: Preserving Privacy with Google Docs and other Web Applications

Gabriele D'Angelo
Dept. of Computer Science
University of Bologna, ITALY
gda@cs.unibo.it

Fabio Vitali
Dept. of Computer Science
University of Bologna, ITALY
fabio@cs.unibo.it

Stefano Zacchiroli*
PPS, Université Paris Diderot
UMR 7126, FRANCE
zack@pps.jussieu.fr

## ABSTRACT

Web office suites such as Google Docs offer unparalleled collaboration experiences in terms of low software requirements, ease of use, data ubiquity, and availability. When the data holder (Google, Microsoft, etc.) is not perceived as trusted though, those benefits are considered at stake with important privacy requirements. *Content cloaking* is a lightweight, cryptographic, client-side solution to protect content from data holders while using web office suites and other "Web 2.0", AJAX-based, collaborative applications.

## Categories and Subject Descriptors

H.3.4 [**Information Storage and Retrieval**]: Systems and Software—*World Wide Web (WWW)*; K.4.1 [**Computers and Society**]: Public Policy Issues—*Privacy*

## General Terms

Design, Security

## Keywords

Privacy, Web 2.0, Google Docs, AJAX

## 1. INTRODUCTION

*Online (or Web) office suites* are web applications that are becoming more and more popular. Collaboration on common office artifacts (e.g. text documents, spreadsheets, etc.) is at present easily available to everyone equipped with a commodity web browser. Google Docs [18] is a peculiar example of such web office suites and aims at providing a full online replacement for legacy desktop applications, such as Open Office and Microsoft Office. The number of alternatives to Google Docs is also growing (Office Live, IWork, Zoho, OpenGoo, etc.), paving the way to further diffusion of such web applications.

*Partially supported by the European Community FP7, MANCOOSI project, grant agreement n. 214898


*SAC'10* March 22-26, 2010, Sierre, Switzerland.


Technically, web office suites are based on *AJAX* [9], the main technical ingredient of the so called Web 2.0 [15]. AJAX provides a framework to implement rich web applications by composing together technologies readily available in web browsers. In spite of its ground breaking effects on web usage, AJAX is rather simple. A web page is initially loaded as with traditional web applications. Then, user actions on visual elements (links, forms, text, etc.) are intercepted to trigger JavaScript callback functions. The effects of those callbacks, which usually implement the application logic, are shown by in-place modifications of the page, through a manipulation of the corresponding DOM tree. When interaction with the server-side logic is needed (e.g. to fetch additional data unpredicted by the initial page load), the data actually transferred is limited to whatever relevant fragment of content needs to be updated and usually the application makes use of significantly compact data encoding, such as JSON [6]. The main advantage offered by AJAX is its potentiality to improve the user experience by increasing application *responsiveness* (a well-known requirement for proper usability). That alone does not justify the relevance of web office suites over their legacy counterparts; a few additional advantages that justify their relevance can be summarized as follows:

- *Low software requirements.* Web applications are trivial instances of the *Software as a Service* (SaaS) deployment paradigm [2]. As such, they only rely on a thin client (the browser) and delegate all software maintenance tasks, such as upgrade management, to the software provider (e.g. Google).
- *Content ubiquity.* SaaS comes with the ability to allow content access wherever network connectivity and the required thin client are available. For web applications such clients, i.e. current generation browsers, are easily available in every Internet point of the world.
- *Content availability.* When the web application is offered by mainstream organization such as Google, the user enjoys reasonable guarantees of content replication and fault tolerance [10] which can hardly be matched by legacy software (and hardware).
- *Collaboration.* Web office suites are very well equipped for collaboration. Not only they offer access control mechanisms that enable to *invite co-authors* and distinguish among read-only and read-write authors, but they also offer simultaneous co-authoring capabilities. Google Docs for instance triggers AJAX updates to

the server every 30 seconds [7], achieving both an intriguing "live" user experience and low conflict likelihood (due to the small size of content chunks which get updated). Another kind of collaboration offered by web office suites is *publishing*, which consists in making content available as public web pages. This capability simplifies the publishing work flow of traditional office suites by removing the extra "upload" step.

- *Versioning.* Given the sheer size of storage resources available to SaaS providers, web office suites are able to offer versioning and unlimited rollback capabilities.

What is the price to pay for all these advantages? Usually it is not monetary, but rather it implies a fundamental *loss of control* over content that users believe to own. The content is no longer confined to the user file system and its access control lists cannot protect it from the software provider. The only offered guarantee is the provider *privacy policy*, which is clearly no silver bullet as long as control *has to* be handed over. More precisely, the problem is not the existence of a trade-off between offered features and privacy risks, but rather that users are faced with an *all-or-nothing* proposition: either they enjoy all of the above features and accept whatever privacy policy is offered (additionally needing to trust the provider not to violate it...) or they enjoy no features whatsoever. The issue is exacerbated by the practice of changing privacy policies over time—which entails other all-or-nothing choices: either the user accepts the new terms or she is "free" to renounce to the service—hereby hindering making long term commitments to a given web application whenever privacy issues are a concern.

This paper presents a client-side solution that enables users to retain control over their content without renouncing to most of the features offered by AJAX-based web applications. The solution, called *content cloaking* (or CoClo for short), relies on a thin software layer running in the browser and playing "man in the middle" for content exchanges with the server. CoClo ensures that each content chunk conveyed by AJAX chats gets encrypted as soon as it leaves the browser and symmetrically gets decrypted while flowing back in. This way, content stored on the server is not immediately accessible to the service provider and its clients.

CoClo is presented in Section 3, which is preceded by an overview of related work. Further on, Section 4 details a proof-of-concept implementation specific to Google Docs. Section 5 presents several use cases that generalize CoClo to web applications other than those discussed before.

## 2. RELATED WORK

Many studies have focused on the privacy risks involved in web applications and, in general, with the storage of personal data "in the cloud" [4, 17]. The problem is only partially new as it can be seen as typical of many distributed systems. Yet, with SaaS scenarios come very new factors such as the huge amount of data collected by service providers and the interactive nature of the provided services. As usual, several different approaches have been proposed to deal with user concerns about the security retained data.

Focusing on technological solutions, new methods have been proposed to preserve privacy in web services [16] and build high-assurance web applications with end-to-end confidentiality and integrity policies [3]. The usage of these methods in existing web applications however requires a significant effort from application providers that are not interested in losing any control on the data stored on their servers. Client-side interoperability middleware solutions have been proposed [12] to cope with untrusted Internet data storage. While interesting, this approach is not intended to deal with the actual web applications that provide user interfaces to the underlying Internet data storage services (e.g. Google Mail, Google Docs, and Flickr).

Just after our initial efforts [8], Christodorescu proposed to use opportunistic cryptography to enhance the privacy of data sent to web servers [5]. That solution shares with ours the very general idea of encrypting the data stored on servers; despite that, the two approaches are different. First of all [5] claims to generically addresses untrusted web servers without considering specific challenges posed by AJAX-like scenarios. Furthermore, it proposes to split "client-side web application *display* code" from "client-side web application *communication* code" without detailing how to do that: those details can be very specific to the underlying technology. In our case, the proposed method is validated by an implementation that assesses its feasibility for AJAX applications. Moreover, the goal of [5] is to *minimize* the impact of the solution on already-deployed web applications, in our case the applications are *not modified* at all. Finally, we consider different perspectives and applicability.

## 3. CONTENT CLOAKING

Web office suites, as well as many other web applications, make users face a binary choice: either using the apps and hand-over any control on their content, or avoiding using it entirely. In practice, users have to compare the benefits offered by the application with the risks derived from the spreading of their information on a cloud architecture outside their control. The main goal of *Content Cloaking* (CoClo) is to weigh in on this trade-off, giving users more control on their data, while reducing to the bare minimum the amount of functionalities they need to renounce to. In particular, the user interface and authoring features should remain unchanged. It is now appropriate to define more in detail the terms of the confidentiality problem faced by users, the main risks can be summarized as follows:

- Unauthorized data retention and aggregation performed by service providers and data holders. Note that, in many cases, the privacy policy of online services is unable to protect users from these practices, which constitute the core business of most providers.
- Leak of private data to unauthorized persons or third parties. The leak could be the resulting effect of deliberate commercial activities, managing faults, or even software bugs (e.g. [11]).
- Third party sniffing of browser-server communications.
- The concrete possibility of unwanted data acquisition and collection mandated by governments or other authorities (e.g. dictatorships or otherwise non democratic governments, limitations on civil rights, etc.).

### *Design.*

The above risks are the starting point for the design of content cloaking. Necessarily, the solution should be *client-side*, due to the inherent architecture of web applications:

the storage cloud is user transparent and obviously the provided services can not be modified. With that in mind, the choice of user interface is more or less forced as well. While many web applications provide some Application Programming Interfaces (APIs) to programmatically access and manipulate data, implementing content cloaking on top of them would necessitate a change in the user interface and the inability to get the feature improvements that SaaS softwares frequently release. Hence content cloaking should rely on existing web interfaces.

The usage of some kind of cryptography is the obvious response to the confidentiality request. The standard behavior of web applications is to transfer and store data in clear. Conversely, for CoClo, as much data as possible should be transmitted and stored in encrypted form. The generation of encrypted data on the client implies not only that eavesdropping during client-server communication is prevented, but also that data is actually stored on the server in encrypted form, so that not even the service provider can access it. Referring again to the previously described risks, secure protocols (such as Secure Sockets Layer, SSL) could be used to prevent sniffing. Yet, this solution is only possible if supported by the AJAX application itself and, in any case, it would be unable to reduce other risks. Therefore, a wider use of encryption is necessary: a cryptographic mechanism that works with the *payload* of application messages and that does not breaks client and server compatibility.

Synthesizing the above design principles and considering the characteristics of AJAX-based applications, content cloaking results in an additional thin software layer to be deployed within the browser. This layer plays a man-in-the-middle role between client and server, and needs to understand the application-specific protocol used by the web application. In detail, it has to detect and recognize the protocol messages to implement the confidentially mechanism, and its requirements. From an implementation viewpoint, this can be accomplished in many ways, the most straightforward choice being a browser extension (i.e. a plugin that allows developers to add functionalities to a web browser). Let's consider a realistic scenario in which a user is editing a document in a web office suite. Each modification conveyed by a protocol message gets intercepted by the extension. The payload of the intercepted message is then encrypted by the extension, and delivered to server side as a new message. Conversely, during document visualization, the extension intercepts all messages coming from the server and decrypts them before client side elaboration (usually involving its rendering on screen). The overall result is that users use the web application as before, the data holder is unable to snoop into the stored information and, in case of data leaks, only encrypted content is released into the wild.

### *Cryptography.*

Focusing on cryptography, many solutions can be used but a very simple approach is enough to satisfy the anticipated requirements. In CoClo, a symmetric-key algorithm is used to encrypt and decrypt all the transmitted data and hence server storage. In case of single-user editing of documents, the user defines a new password (i.e. key) each time a new document is created. Later on, the password can be easily changed using a specific feature provided by the browser extension that, in fact, will transparently create a new document, encrypt it with the new key, while deleting the old one. In this single-user case, the key of each document can be maintained in client local storage.

Multi-user collaboration adds some complications: the key has to be shared among all users that are allowed to edit or view the document. Key distribution is a very well known problem in cryptography, in CoClo it can be effectively solved using a public-key cryptography scheme. Similarly to the single-user case, the secret key is defined at document creation time, but now it has to be communicated to all participants. In CoClo, this is done by using the public key of each of them, that is by encrypting the password using the public key of each receiver and storing the result in a specific online document that is no further encrypted. This way, each legitimate receiver will be able to obtain the encrypted password and to decrypt it using her private key. An alternative approach is to require the usage of an offline delivery mechanism (for example emails). Using this scheme, the secret password used for the symmetric encryption algorithm is never transmitted or stored in clear (on the server). In practice, the described procedure has to be repeated each time a participant decides to change the document password. Moreover, it is worth noting that read-only users are unable to change the password, given that this access mechanism provided by the web application remains unchanged. A final design note is about the usage of public key cryptography, for our purposes it is possible to use one of the many available public key infrastructures (such as the web of trust implemented by the OpenPGP key servers).

### *Analysis.*

Content cloaking satisfies all imposed requirements: content is protected from unauthorized accesses, from both the service provider and third parties. The transmission of cleartext content is avoided also when the web application does not support secure transmission protocols, and the data acquisition by governments or external authorities is made much harder. All of this is accomplished while the original user interface of the web application remains unchanged. Yet, server-side services that are based on the document data content are no longer available, since with CoClo the server has access to encrypted data only. In practice, this limitation is much less severe than expected. For example, with current generation online word processor the main missing feature is full-text search through multiple documents. On the other hand, in spreadsheets the situation is a little worse, given that the most part of the computation is server-side. In this case, it is not possible to encrypt numerical values, since computation would become impossible or unreliable, but only the data labels. More precisely, cells appearing as arguments in other cell functions cannot be encrypted while other cells can. On that regard, we observe that a spreadsheet without text labels is hardly intelligible by both humans and machines. An alternative, but more complex and intrusive solution requires to move some part of the computation on the client.

To conclude the analysis, the success of web office suites is also due to their very low software requirements (e.g. a web browser). In CoClo, the software requirements are higher by little: an extension has to be installed in the browser. Given the ease of web browser deployment in modern browsers (a hyperlink to a single file is enough), in our opinion, this requirement is rather reasonable. From the computer security viewpoint, CoClo is unable to prevent

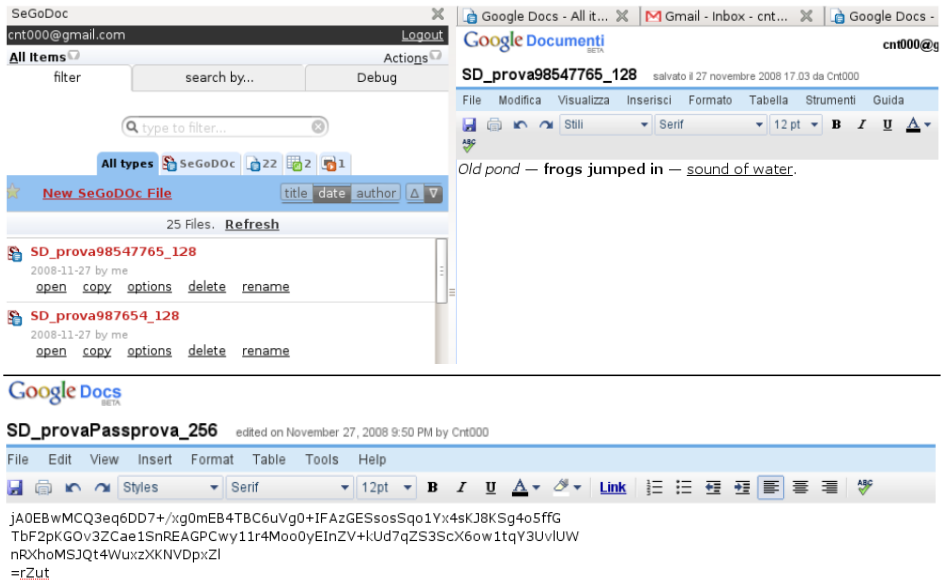


**Figure 1: SeGoDoc user interface and examples**

data leaks due to compromised clients, i.e. clients where local vulnerability or social engineering have been exploited to install back-doors, Trojan horses, or in general to affect the local security of the system (for example a key-logger). In the real world, this security problem can not be ignored but it is neither connected to nor introduced by content cloaking: the security of web clients (of all clients, in case of collaborative editing) is a common assumption of most online applications (e.g. online banking).

## 4. IMPLEMENTATION

A prototype implementation of CoClo has been developed as an extension for the Firefox browser. The prototype is called *SeGoDoc* (for "Secure Google Docs") as it implements content cloaking specifically for Google Docs, and in particular for its word processor. SeGoDoc's main interface is a sidebar, developed as an extension of gDocsBar.[1] The main functionality of gDocsBar, i.e. document browsing, is left unchanged and login to Google Docs can be performed directly from the sidebar. The user interface is shown in the upper part of Figure 1.

Just after login SeGoDoc registers, using the extension API, two observers to monitor HTTP activity. The observers know the Google Docs AJAX protocol and recognize both the request parameters used to send content to the server (`docContents` and `delta`) and the AJAX callbacks that receive new content for the browser. When receiving content, the content is first decrypted and then delivered to the original receiver (the Gecko engine), thus not hindering usual markup rendering. Currently, SeGoDoc only implements AES [14] (in pure JavaScript), and stores some meta information about it (such as key length) in mangled file names to enable content access from different browsers. Passwords are requested when creating or opening a document for the first time, and stored only temporarily in memory. Users missing SeGoDoc, or providing the wrong password, will simply see (ASCII-armored) garbage, as shown in the lower part of Figure 1. While in theory encryption-unaware users can mess up encrypted content, Google Docs versioning effectively protects against similar problems.

Currently public/private secret sharing (discussed in Section 3) is not implemented, but it can be easily achieved by coupling SeGoDoc with third party extensions such as FirePGP.[2] SeGoDoc is free software (GPL license); its source code is at http://segodoc.trac.cs.unibo.it/browser/.

## 5. USE CASES

The presented content cloaking implementation is specific to Google Docs and gives privacy guarantees during common word processing tasks. Content cloaking however is more generic than that: it unleashes potentialities of AJAX-based collaborative applications and can also be applied to other kinds of web applications. The use cases presented here account for both aspects of content cloaking generality.

**Slide delivery**. A frequent need with teaching material is to make it available to students without releasing it publicly on the web (e.g. because the material is meant to become part of a commercial book). While this need is addressed by web presentation tools like the Google Docs presenter, user management quickly becomes cumbersome: students need to be invited to share the document and, year after year, accounts need to be replaced. "Old style" solutions such as password-protected pages are not directly applicable to document authored in, and published by, Google Docs. Content cloaking simplifies the task: the material can be published in encrypted form, while the key needed to decrypt it can be announced at lectures and easily changed year after year.

**Prior art management**. Content cloaking can turn Google Docs word processor in a powerful prior art registration device. Documents explaining prior art can be pub-

[1] http://www.gdocsbar.com

[2] http://getfiregpg.org/

lished as encrypted material; with implementations similar to SeGoDoc titles are already left unencrypted, while other unencrypted parts can be published in related clear text documents. At that point, Google offers timestamped and archivable publishing of the prior art material, also offering indexing of all clear text. When the time comes to reveal the full prior art material, password can be published enabling everyone to access the full content.

**Embargoed bug tracking**. Vendors such as FOSS (Free and Open Source Software) distributions collaborate in dealing with sensible bug reports, such as easily exploitable security flaws. The initial life time of such bugs is "embargoed" and hidden from the public, *de facto* leaving involved teams locked out of their usual bug tracking systems (BTSs). Content cloaking can be applied to BTSs to selectively encrypt parts of bug logs (e.g. proof of concept exploit code, affected versions, etc.). Key management is straightforward and the loop of people having access can easily be enlarged to, e.g. upstream authors. Once the issue is fixed, keys can be released to the public (or bug log decrypted *tout court*).

**Encrypted web-based IM**. Privacy concerns similar to those discussed thus far have historically been perceived by users of IM (Instant Messaging) services which have no protection against service providers. Current solutions rely on payload encryption (e.g. Off-The-Record (OTR) communication [1]) and are usually implemented as plugins for stand-alone IM applications. Such solutions are not applicable in web-based IM like Google Talk, since users cannot install plugins there. Content cloaking can come to the rescue by encrypting messages flowing from the browser to the service and back: OTR can easily be implemented for Google Talk[3] providing compatibility with stand-alone IM applications.

**The dissident wiki**. Content cloaking is not specific to AJAX applications: it can be implemented also for legacy web applications. For instance, specific wikis and their instances can be recognized to have their content encrypted and decrypted on the fly according to the CoClo architecture. Even though that might seem at stake with the open access design principle of wikis [13], specific real-life scenarios already requires similar features, such as a hypothetical "dissident wiki", where the wiki provider cannot be trusted as it can be forced to disclose content by authorities.

## 6. CONCLUSIONS AND FUTURE WORK

*Content cloaking* is a lightweight, client-side, software solution to give confidentiality guarantees in Software as a Service web applications (e.g. Google Docs), where the user does not control the storage of data. Content cloaking relies on on-the-fly encryption of application messages, which are usually exchanged via AJAX. The proposed solution shields from eavesdropping both during transfer and at destination (e.g. at Google's), at a minimum price of some content-based features that should be given up. A proof-of-concept implementation for Google Docs word processor has been presented to assess the feasibility of the approach. The main future development we envisage is the integration of secret sharing schemes with content cloaking. That would enable to create collaborative web environments in which a document is accessible only in presence of a sufficient number of participants, each of them holding only a small slice of the encryption key.

[3]Note that this has nothing to do with Google's "chatting off the record" (http://www.google.com/talk/chathistory.html#offrecord), in spite of the unfortunate name clash.


## Acknowledgements

The authors would like to thank Edoardo Gargano for his work on the proof-of-concept implementation, and Enrico Tassi for his insights on the presented use cases.


## 7. REFERENCES


[1] N. Borisov, I. Goldberg, and E. A. Brewer. Off-the-record communication, or, why not to use PGP. In *ACM WPES*, pages 77–84, 2004.

[2] M. Campbell-Kelly. Historical reflections. the rise, fall, and resurrection of software as a service. *Commun. ACM*, 52(5):28–30, 2009.

[3] S. Chong, K. Vikram, and A. C. Myers. Sif: enforcing confidentiality and integrity in web applications. In *USENIX Security Symposium*, pages 1–16. 2007.

[4] S. Chopra and L. White. Privacy and artificial agents, or, is google reading my email? In *IJCAI*, pages 1245–1250, 2007.

[5] M. Christodorescu. Private use of untrusted web servers via opportunistic encryption. In *W2SP 2008: Web 2.0 Security and Privacy 2008*. IEEE, 2008.

[6] D. Crockford. The application/json media type for JavaScript object notation (JSON). RFC 4627. 2006.

[7] S. Dekeyser and R. Watson. Extending Google Docs to collaborate on research papers. Technical report, University of Southern Queensland, Australia, 2006.

[8] E. Gargano. Privacy e Web 2.0: una soluzione sicura per utilizzare Google Documents e Mozilla Firefox. Master's thesis, University of Bologna, 2007.

[9] J. Garrett. AJAX: A new approach to web applications. http://www.adaptivepath.com/ideas/essays/archives/000385.php, 2005.

[10] S. Ghemawat, H. Gobioff, and S.T. Leung. The Google file system. In *ACM SOSP*, pages 29–43, 2003.

[11] Google Docs blog. On yesterday's email. http://googledocs.blogspot.com/2009/03/on-yesterdays-email.html, Mar. 2009.

[12] R. C. Jammalamadaka, R. Gamboni, S. Mehrotra, K. Seamons, and N. Venkatasubramanian. iDataGuard: an interoperable security middleware for untrusted internet data storage. In *ACM Middleware Conference*, pages 36–41. 2008.

[13] B. Leuf and W. Cunningham. *The Wiki Way: Quick Collaboration on the Web*. Addison-Wesley. 2001.

[14] NIST. Advanced Encryption Standard (AES). Federal Information Processing Standards 197, Nov. 2001.

[15] T. O'Reilly. What Is Web 2.0: Design Patterns and Business Models for the Next Generation of Software. *Communications and Strategies*, 65(1):17–38, 2007.

[16] A. Rezgui, M. Ouzzani, A. Bouguettaya, and B. Medjahed. Preserving privacy in web services. In *ACM WIDM*, pages 56–62. 2002.

[17] O. Tene. What Google knows: Privacy and internet search engines. *Utah Law Review*, 2008. Forthcoming. Available at http://ssrn.com/abstract=1021490.

[18] Writely and Google Inc. Google Docs. http://docs.google.com.