My Private Google Calendar

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Technical Report Nr. 759

Systems Group, Department of Computer Science, ETH Zurich

My Private Google Calendar

by

Tahmineh Sanamrad, ETH Zurich
Daniel Widmer, ETH Zurich
Donald Kossmann, ETH Zurich

Apr. 26, 2012

inf | Informatik
Computer Science

Abstract

Everybody loves Google Apps. Google provides highly available web applications that help you communicate, organize and collaborate from anywhere using different interfaces in the most user friendly and efficient way, without being worried about any IT issues. However, some people still hesitate using Google services because of privacy and trust issues. In this paper, we identify privacy issues in Google Web Applications as a particularly vital problem and propose a solution. In our solution a transparent encryption layer is put between the user and the cloud service provider on a site trusted by the user. This layer accesses the request and response messages passed between the two parties in a fine-grained manner. It applies modern cryptography techniques to encrypt the data without sacrificing functionality and portability of the cloud service. This way the trust of the end user can be reobtained and he or she will be encouraged to further enjoy using web applications such as Google Apps without having to worry about privacy issues.

1 Introduction

Trust and privacy play a crucial role in todays applications, as more and more people and companies decide to outsource their data and IT services. There are plenty of cloud data services and web applications that help to organize and store data for free or at a low cost. Still, for some people high availability, up to date features, light-weight interfaces, backup, low maintenance costs and portability on the latest mobile devices seem all to be overshadowed by privacy doubts and suspicions [18, 12]. The goal of this paper is to present a case study on how to enforce privacy on top of a popular service, i.e., Google Calendar.

This paper makes several contributions. The first contribution is the proxy architecture that suggests having a security middleware as a transparent encryption layer on a site trusted by the client. Although using a security middleware is not a new idea, and has been exploited in several previous works such as [14, 19], the idea of having a proxy server in the security middleware that inspects and accesses the http message body, selectively encrypts/decrypts its content in a fine-grained manner, thereby preserving all the functionalities and the original APIs, is a novel approach. The main advantage of this architecture is the transparency of the complicated encryption mechanism and the key management towards the end user. Also, given the variety of mobile devices that embed various cloud data services, our approach assures a device-independent installation and thereby portability is guaranteed.

The second contribution of this paper is a novel encryption scheme. Since the user expects the functionality of Google calendar with additional security on top, a strong, displayable, searchable and shareable encryption schema needs to be devised. In order to fulfill the vital calendar requirements, first of all, an acceptable security level needs to be provided; to achieve this a probabilistic encryption scheme seems to be a wise choice. On the other hand, since searchability is also an indispensable feature of a web calendar, a deterministic encryption scheme appears to be the way to go. To meet both of these requirements, we are using a hybrid encryption scheme. First, we are using an AES-128bit in Cipher Block Chaining mode to ensure a CPA-secure encryption, then we combine it with a one-way collision-resistant hash function namely SHA-2. This combination not only guarantees a CPA-secure encryption scheme, but also allows us to search for events even in the shared calendars (more on this in section 3).

The third contribution of this paper is the key management and salt distribution among the users. The main problem to be solved here is the calendar sharing issues between users in the same or different proxies. Not only the users need to be able to view the calendar entries of other users(from the same or different proxy), but they also should be able to search for entries across all the shared and non-shared calendars while keeping the privacy of each single user intact. There are multiple ways to distribute and manage keys. Also to be safe against rainbow attacks, an additional salt needs to be added to the plaintext, so both keys and salts need to be managed and distributed in a secure and reasonable way (more on this in 3.4).

The fourth contribution is the composition and orchestration of different components on the middleware

to rebuild the functionalities of our Google Calendar case study, while preserving privacy of the users. Along the way, many interesting problems have been addressed and new techniques have been devised. For example, one of the essential features of a web calendar is the ability to send and receive invitations. In our case, the event should be encrypted and the invitees should be hidden from the cloud service provider. To achieve this, an SMTP server component has been added to the proxy middleware, so that invitation messages get decrypted before being forwarded to the invitees. This way we assure both the privacy of the event content and the invitees.

Organization. The remainder of this paper is organized as follows: In section 2 the architecture of our approach is being discussed and compared against other possible approaches. In section 3 the security challenges will be discussed, including the encryption scheme and the key management issues. In section 4 the orchestration of the components to provide the required functionalities are being demonstrated. Afterwards in section 5 implementation will be briefly discussed along with the set up and configuration efforts that need to be done. Finally, in section 6, benchmark results will be shown to give the reader an approximation of his or her expectations from the system in terms of response time after adding the proposed encryption layer in between. In section 7, we will discuss the related work and eventually conclude this paper in section 8, while adding the future work prospects.

2 Proxy Architecture

In this section we will first explore different possible approaches to provide security for the end user given our case study, Google calendar. The main idea is to replace the plain text content entered into the Google calendar's web interface with a ciphertext before submitting the request to Google. In order to accomplish this, there are two possible approaches as shown in figure 1:

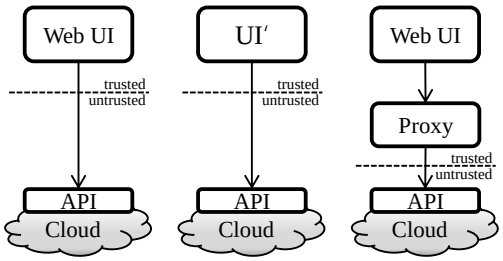
1. Having a rich client that takes care of the replacement
2. Adding a level of indirection between the client and the service provider

In the next subsection we are comparing these two approaches and eventually justifying our chosen approach.

2.1 Rich Client vs. Proxy Middleware

As shown in figure 1(a), currently the users directly use the API of the cloud service provider and submit their requests in plaintext. Although an SSL connection is established between the user and the cloud service provider, the data stored and processed on the cloud side is in all plaintext.
To have the data encrypted before submission, there are two possible solutions. The first, as shown in 1(b) is to implement a new user interface on the client side, and add an additional encryption method that takes care of the security before submitting any request into the cloud. In our case study, Google actually provides a neat *Calendar Data API*[3] that enables the users to access its calendar methods, such as create, edit, and invitations, while giving them a lot of flexibility to design their own desired user interfaces. A considerable amount of documentation has been written and dedicated to support developers who use this API. Its client library supports multiple common programming languages such as .NET, Java, PHP and Python[3]. On the other hand, given the variety of the devices and hardware architectures, implementing such a user interface for each and every device out there is a cumbersome task with a high development cost. Additionally, the current user interfaces already provided by the cloud service providers are efficient and friendly enough. Another disadvantage is not only the high development cost but also the end user has to undergo a lot of installation and set up efforts on every single device she or he has.

The second approach that is proposed in this paper and is shown in 1(c) adds a level of indirection between the user and the cloud service provider. This level of indirection resides on the user's trusted site and is a proxy server that acts as an intermediary for requests from clients seeking resources from the cloud service provider. The first advantage of our approach is that the client accesses the proxy server using the same API as the one provided by the cloud service provider; this assures the transparency of the seurity mechanism to the end user, so the user should not even be aware of the encryption process going on behind the scenes. The second advantage is the very low installation cost for the user; all that the user needs to do is to route the relevant traffic through the proxy middleware, and the rest is taken care of by the middleware. The third advantage is the portability of our approach. This way all the user's devices can keep their current well provided and maintained interface of the cloud service provider and only route their traffic through the proxy middleware.



(a) Traditional (b) Rich Client (c) Proxy MW

Figure 1: Comparing different approaches

In the next section, the core components of the middleware proxy and their interactions in order to make our system work and achieve our desired goals will be introduced.

2.2 Make it work!

As already discussed in the previous subsection, there are several advantages associated with the proxy architecture. Having a security middleware has previously discussed in the database encryption research area in [14], or in some works related to the Internet data storage security such as [19], but the proxy middleware gives us an additional ability to further inspect the traffic and provide selective fine-grained encryption on the most sensitive fields. This way the amount of encryption is configurable to adjust the non-functional requirements of the user, such as responsiveness along with privacy while preserving the full functionality of the service. The first step to build such a proxy middleware is to identify its crucial components as follows:

- *Proxy server*. As shown in figure 2, the client connects to the proxy server on the middleware, requesting some service from the cloud service provider. The proxy server evaluates the request according to its filtering rules. If the request is validated by the filter, the proxy provides the resource by connecting to the cloud service provider and requesting the service on behalf of the client. Basically, the proxy server interferes in all the traffic passing between end user and the cloud service provider. The proxy server's interference is through an ICAP server.
- *ICAP Server*. ICAP stands for Internet Content Adaptation Protocol which is used to extend transparent proxy servers, as it is shown in figure 2. This lightweight http-like protocol is specified in RFC 3507[15]. This component adapts the content of the http messages by performing the particular value added service (content encryption/decryption) for the associated client request/response.

- *Database*. The database keeps track of the registered users' information such as their ac

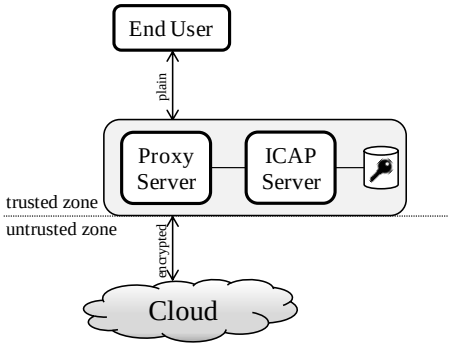


Figure 2: Our Architecture

Having the data encrypted in a fine-grained manner, we need to make sure that the chosen encryption schema is not only secure enough but also the functionalities of the cloud service are preserved. Thus, the second contribution of this paper is the unique encryption schema that has been composed to fulfill the functional and non-functional requirement of our case study, i.e. Google Calendar. The encryption schema is a hybrid scheme composed out of the most secure and advanced available encryption schemes. There has been several problems being addressed by our encryption schema as follows:lfill the functional and non-functional requirement of our case study, i.e. Google Calendar. The encryption schema is a hybrid scheme composed out of the most secure and advanced available encryption schemes. There has been several problems being addressed by our encryption schema as follows:

- CPA-Secure Encryption
- Search and Inforamtion Retrieval
- Calendar sharing and key distribution

Those items listed above are all functional and non-functional requirements that were challenging to achieve given our case study, which is Google Calendar, and our goal which is to provide privacy on a fine-grained level while preserving all the functionalities. In the next section we will describe our proposed encryption scheme that solves the above mentioned issues in a novel way in more depth.

3 Security

One of the main contributions of this paper is the functionality preserving encryption scheme. In this section first we introduce the basic encryption scheme used, and analyze its security.

3.1 Encryption Scheme

As described in the previous section, the main idea is to read the plain text data out of the user's request and replace it with a ciphertext before it hits the cloud service provider. Choosing a proper encryption schema was quite challenging since not only data should be displayable but it also has to be searchable. For this matter we have selected a combination of symmetric key encryption and hashing. The reason to choose a symmetric encryption is first because our case study has a single point of encryption/decryption

which is in the proxy middleware, and the second reason is that asymmetric encryption is computationally hundred to thousand times more expensive.
Among the symmetric-key algorithms, we have chosen *Advanced Encryption Standard(AES)* that is one of the most recommended[20, 13] encryption standards that gets a lot of attention from the security community in terms of analysis and cryptanalysis. To be more concrete, we have chosen AES with a key length of 128-bits[1]. Additionally to introduce more randomness we make use of a probabilistic encryption schema by having AES operating in *Cipher Block Chaining(CBC)* mode. As also illustrated in figure 3, a random *Initialization Vector(IV)* is first chosen; then, each of the remaining ciphertext blocks is generated by applying the pseudorandom permutation to the XOR of the current plaintext block and the previous ciphertext block[20]. Since we want to have a stateless encryption schema, we are encrypting the initialization vector with another key in *Electronic Codebook(ECB)* mode which is deterministic, and then concatenate it to the ciphertext.
In the remainder of this subsection we will analyze the security of our encryption scheme so far with the

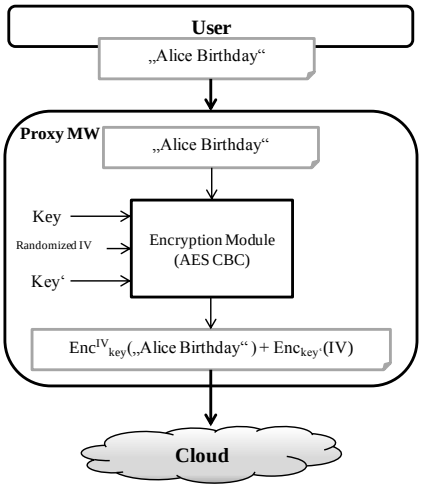


Figure 3: The basic encryption scheme

possible attack scenarios.

Security Analysis

The most basic attack scenario is the *Ciphertext-only attack*, where the attacker is given only a series of outputs $enc_k(plain)$ for some inputs $plain$ unknown to the attacker[20]. This attack scenario is easily launchable in our case study under the assumption that attacker has access to the data stored on the public cloud or he is listening to our communication channel. It has been stated in [20] that any secure encryption scheme must have a key space that is not vulnerable to exhaustive search; Thus, AES-128 bit is a safe option of course under the condition of having a strong key chosen for encryption.
The next and stronger attack scenario is the *Known-plaintext attack(KPA)*, where the attacker is given pairs of inputs and outputs $(plain_i, enc_k(plain_i))$. KPA is one of the most probable attack scenarios in our case study, under the assumption of having some background knowledge about the user. In this case attacker can verify his guesses about some of the events in the user's calendar and discover the key. Again

[1]Of course having a 256 bit key is much stronger but also computationally more expensive, thus we can ignore the additional security gain

AES[2] was explicitly designed to address concerns regarding the short key length and block length of the previously widely used encryption scheme called *Data Encryption Standard(DES)* to be secure against KPA[20].
The last and strongest attack scenario regarding our case study is the *Chosen-plaintext attack*, where the attacker is given $(plain_i, enc_k(plain_i))$ for a series of inputs $plain_i$ that are chosen by the attacker[20]. This scenario requires an active attacker that sends plaintext and receives ciphertext. This attack is also probable, when an attacker who has access to the ciphertext also registers to use the proxy service. According to the literature[20], no deterministic encryption scheme can be secure against chosen-plaintext attacks. Rather, any CPA-secure encryption scheme must be probabilistic. Thus, we are using a probabilistic mode of operation of AES, which is the Cipher Block Chaining mode.

3.2 Search on Encrypted Data

For many people the main reason to use a computer based calendar is its search functionality. So far, our encryption schema consists of the randomized encryption of the entry and the randomized element being encrypted deterministically. Assume now we have a search query like *"birthday alice"*. With the current encryption schema it is impossible to find the event and retrieve it because:

$$enc_k(\text{“}Alice\ Birthday''\text{)} \neq enc_k(\text{“}birthday\ alice''\text{)}$$
$$enc_k(\text{“}Alice\ Birthday''\text{)} \neq enc_k(\text{“}Alice\ birthday''\text{)}$$
$$enc_k(\text{“}Alice\ Birthday''\text{)} \neq enc_k(\text{“}alice\ birthday''\text{)}$$

Thus, our approach should not only to encrypt the whole entry exactly as the user has entered it in terms of upper or lowercase and whitespaces to display it correctly to the user, but also it should tokenize and normalize the user's entry and store them separately for the *search* to work. As we know, from an information retrieval aspect, whatever we do with a document in terms of tokenization, normalization, stop word elimination and stemming, we need to apply also to the query[21]. Using a probabilistic encryption scheme will not allow us to reconstruct what we have stored in an encrypted fashion in the public cloud. Therefore, to make the calendar entries searchable, we store the hash of the tokens along with their probabilistic encryption in the public cloud as shown in figure 4.
Toward a secure hash function. Most cryptographic hash functions take a string of any length and produce a fixed-length hash value. We have chosen SHA-2[11][3] The final output of our suggested encryption schema is demonstrated in figure 4. For example, an event entitled *"Alice Birthday"* is being processed through two separate modules, one is responsible for the CPA-secure encryption and the other is responsible for the information retrieval.
One of the main advantages of our approach is the transparency of the security process to the end user. Thus, we can strengthen or weaken our encryption scheme on the middleware, yet it appears effortless to the end user. In the next subsection we analyze the security implications of our enhanced encryption schema storing additionally SHA-2 hash of the tokens in the public cloud.

Security Analysis

Since we have additionally hash the tokens to be stored in the public cloud. It is important to analyze the security of the chosen hash function. First of all a strong hash function needs to meet the following essential security requirements:
Preimage Resistance. Given a hash *h* it should be difficult to find any message *m* such that *h=hash(m)*.

[2] AES can be considered the replacement of DES which was broken in the latest challenge in 22 hours in a known-plaintext attack attempt[20]

[3] SHA is short for Secure Hash Algorithm and is designed by National Security Agency(NSA).

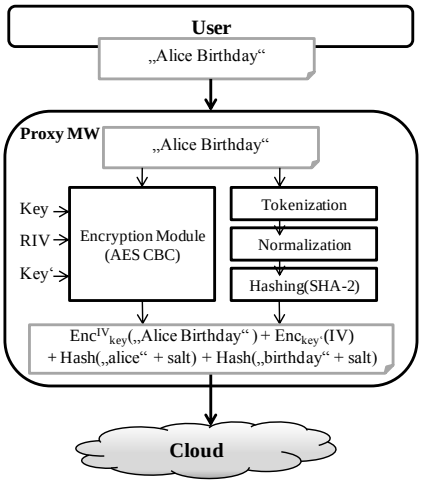


Figure 4: The complete encryption scheme

This concept is related to the one-way function property. Functions that lack this property are vulnerable to preimage attack.
Collision Resistance. A function *h* is collision resistant if it is infeasible for any probabilistic polynomial-time algorithm to find a collision in *h*.
A non-reversible, collision-resistant hash function is still prone to some potential attacks as will be described in the following: *Rainbow Table*. Rainbow tables are precomputed tables for reversing cryptographic hash functions[22]. Thus, we include a large *salt* value to prevent these kind of attacks by ensuring that each token is hashed uniquely. In order to succeed, an attacker needs to precompute tables for each possible salt value; hence, the salt value must be large enough otherwise an attacker can make a table for each salt value.
Correlation Attack. In our case study the data sets are the events in a calendar. The main problem here is that a lot of statistics and correlations can be extracted given that SHA-2 is deterministic. so the attacker can build a correlation graph for the identical tokens and due to the repetitive nature of the events in the calendar and their distribution, some tokens can be easily guessed. For example, birthdays are yearly events, so an adversary can easily assume that a hashed token that is yearly repeating corresponds to the plaintext *“birthday”*. Another example is when the adversary knows that the user is a professor and a conference is taking place at certain days that has been marked on the calendar, he can guess the plaintexts. Therefore, in order to be secure against this kind of attack we suggest the following solution:
Bucketization. This method suggests using a hash function with collisions. This way an adversary cannot be able to ever verify his guess easily, since the token correlation between events do not leak information anymore. Afterwards, the encrypted event will be decrypted on the middleware and through post-processing efforts the additional false positives are eliminated. This approach has been suggested and implemented in several searchable database encryption literatures[14, 16].

3.3 SSL Interception

As we know, most web applications including Google’s are accessed via HTTPS, a protocol that establishes a secure connection between the browser and Google’s servers. In other words, Google uses SSL(Secure Socket Layer) protocol which encrypts the segments of network connections above the

Transport Layer, using asymmetric cryptography for privacy and a keyed message authentication code for message reliability. Coming back to our approach, normally a proxy server should(would) not be able to read the content of the message as an intermediary between the Google and end user. Nevertheless, some proxy servers offer options to decrypt SSL traffic and allow transparent SSL traffic redirection; thus, instead of having an encrypted SSL tunnel between the end user's browser and Google's server, as shown in figure 3.3, our proxy server terminates the Google's SSL traffic at the proxy level. Sequentially, the ICAP server extracts the content to be encrypted and reconstructs the HTTP message on its way to the Google server. The adapted content is then sent to the end-user, while presenting a forged SSL certificate to the users browser. In other words, our middleware basically performs something similar to a Man in the middle attack, but the big difference is that the user agrees to give our middleware the permission to access and replace its contents. An illustration of this flow is shown in figure 4.1.

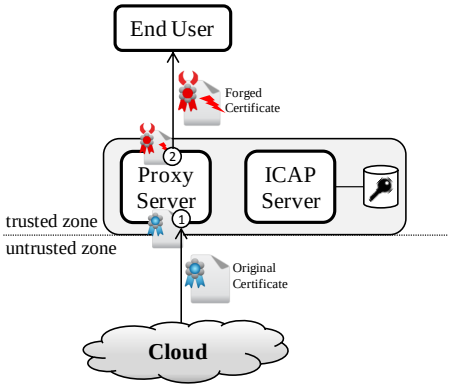


Figure 5: SSL Interception: At point 1: the original Google certificate gets terminated by the proxy and at point 2: a forged certificate which the user already trusted travels back to the end user.

We have different options to intercept the SSL connection. One possible way is that the the end user needs to confirm a certificate exception on his browser for the first time visiting the google.com/calendar. Indeed, this is the option we have chosen, since it has minimal configuration effort for both the end user and proxy middleware. Another approach is that the user installs our root CA Certificate in the Trusted root Certificate Authority list of his browser. This might be difficult for home users, but this is actually what IT departments of companies do all the time. Our certificate can then be signed by verisign. However, no reputable Certificate Authority will sell us a certificate for the google.com domain. Another option is that we offer an additional web interface, similar to http://www.onlineanonymizer.com/ [8] which allows the user to access the calendar through our website. This website could have a legitimate SSL certificate (which is signed by a trusted party such as VeriSign or similar). There might be a legal issue with this second approach: We are basically serving Google's content through our own website. However, one could argue that this is the exact same thing as onlineanonymizer.com is doing and therefore it would all be fine.

3.4 Key Management

As explained in previous section, in order to encrypt an event we need to have a key and an initialization vector. The IV is generated by a pseudorandom function in order to be CPA-secure[20]; the IV is then concatenated and stored with the encrypted event in the public cloud. However, the key is kept on the trusted middleware! Here, another challenging issue emerges that deals with the key management, particularly since in our case study the calendars can be shared among multiple users.

The data model of the Google Calendar combined with our proxy middleware is shown in figure3.4. According to this schema, a Proxy has several users who subscribed to it for the security service, a user

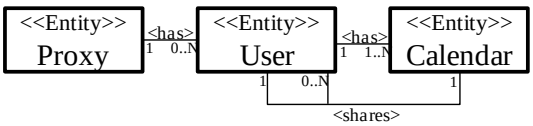


Figure 6: Data Model

can have several calendars and a calendar can be *shared* among several users. Thus, we have again several options for the number and the way we distribute keys among proxies, users and calendars.
One key per proxy. All the events of each proxy user are encrypted using one key. This indeed leads to a trust issue between the proxy users. Since they have the same key, a chosen-plaintext attack scenario can easily be launched to discover the key. However, using the AES in CBC mode makes it impossible for the adversary to succeed. Another disadvantage is sharing a calendar; if user A shares a calendar with somebody using the same proxy, everything will be fine, but as soon as the user A shares it with user B from proxy B, the encrypted events can no more be retrieved unless the user shares his/her calendar key. This way the key management and trust issues will hardly be control *One key per user*. If we would have one key per user, again when a calendar is shared by user A, he has to disseminate his key. This seems to be an open door to all of user A's calendar data to user B, i.e. user B has the key to all shared and unshared calendars of user A.
One key per calendar. Calendar sharing is solved by each calendar having its own key. If user A shares a calendar with user B, he also shares the key of the shared calendar, but user B cannot decrypt the events of the users of proxy A, neither can he decrypt other calendars of user A.

3.5 Salt Distribution

In section 3.2 we have discussed that we need to use a strong salt for our hash function in order to be secure against precomputed *rainbow tables*. In this section we will discuss how the salt should be distributed. In order to search for a query like *"Alice Birthday"*, proxy middleware needs to reconstruct the tokens stored in the public cloud. If the salt is distributed on the proxy, user, or calendar level, provided that a user might have shared calendar from other users in other proxies, and since we do not know in beforehand in which calendar the search result resides, the search functionality will break. This is because the tokens could not properly be reconstructed to produce all the hits. Thus, we will need a salt per event, i.e. same tokens have the same salt, independent of to which calendar, user or proxy they belong. As an example we might use a hash function like the following to produce the salt:

$$Hash(token) = SHA2(token + SHA2(token))$$

It is highly recommended not to openly publish what kind of hash function is used to avoid constructing rainbow tables. One might argue that this is not secure enough, still there are multiple ways as discussed in section 3.2 that can make our approach safer by introducing for example false positives to the result set, and eliminate them on the proxy middleware by some additional post-processing efforts.

4 Functionality

Encrypting the data changes the functionality of the cloud service in multiple ways. Therefore, adjustments were required to be applied to some of the features of our case study example, Google Calendar.

In the remainder of this section, first we will introduce the important features of our case study and then describe the adjustments required to preserve the functionalities. We believe that these adjustments are not only specific to our case study, but they can be applied to a lot of similar scenarios.

4.1 Create, Display and Edit Events

In the case of event creation, display or edit, as shown in figure 4.1, in the first step the user enters plaintext event data and POSTs the request to the proxy server. Each request has a URL and a message body. Depending on the request URL, the corresponding script on the ICAP Server begins to process the request by extracting and encrypting the event data. In the fourth step, the reconstructed message is being forwarded to the cloud service provider by the proxy server. Finally, a response is sent back to the user through the proxy. Depending on the content of the response, it might get decrypted on its way back to the user.

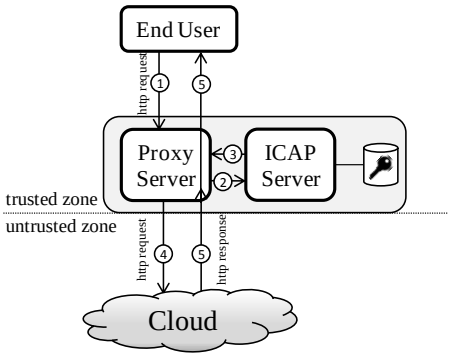


Figure 7: Request and Response Sequences

As discussed earlier, one of the main contributions of this paper is the fine-grained information extraction and encryption. Here, we want to take a closer look on what information is being extracted from an http message body and what processes are being done on the extracted content. In our case study, a *calendar event* has multiple properties such as title, date, time, etc. As described in section 3, there are a few processes performed on the plain text before being sent to the cloud such as tokenization, narmalization, hashing, etc. In table1 you will see what properties go through which transformations and why.

Table 1: Properties of a Calendar Event and the applied Transformations by the proxy middleware ('-' means: not applicable)

Property	Tokenization	Stop Word Elimination	Stemming	Randomized Encryption	SHA2 Hashing
Event Title	✓	✓	✗	✓	✓
Date	✗	-	-	✗	✗
Time	✗	-	-	✗	✗
Repeatability	✗	-	-	✗	✗
Where	✓	✓	✗	✓	✓
Calendar	✗	✗	✗	✗	✗
Description	✓	✓	✗	✓	✓
Guests	✗	-	-	✓	✗

Table Summary. Beginning with the tokenization process, its main use is to support search, therefore the fields that search is performed upon such as *Title*, *Where* and *Description* are being tokenized, their stop words are being eliminated, hashed and stored in the public cloud. Currently *stemming* is not provided by Google calendar search; we can however, provide a better searchability to our users by supporting also stemming. *Date* and *Time* are not being encrypted due to the complexity they will introduce; what we have observed while encrypting *Date* and *Time*, was that events are prefetched based on a specific time-frame by the web interface; Hence, encrypting them strongly interferes with the prefetching mechanisms of the web application on the client side. Generally, the properties that are kept in plain are the ones whose additional security gain is considered negligible, at least for now.
Confidentiality of the past! As we have progressed with the encryption, an important question showed up. What do we do about the events in the past? One solution could be to encrypt them too. Nevertheless, the public cloud already got hold of them, so encrypting them does not improve privacy. As a result, we have left them unencrypted and consequently unretrievable by our search mechanism. There is also a security aspect to it; the hash of the past events can easily be verified since the public cloud has(or had) their corresponding plain text. The future hashes with a high probability can be repetitive events from the past, thus a lot of new hashes will be disclosed immediately.

4.2 Search and Information Retrieval

Preserving the search functionality was particularly challenging, since the data needs to be retrieved and displayed back to the user as he had entered it, yet it should also be searchable. Therefore, we need to store the whole event title, word by word(i.e. no stop word elimination), in the correct case(i.e. no normalization) and with all the whitespaces(i.e. no whitespace elimination). Nevertheless, encrypting the whole phrase will not allow us to search for a single word, hence we are using another method along with the randomized encryption to make the events also searchable. Therefore, for each event we tokenize, eliminate stop words, normalize tokens, and hash them before sending them to be stored in the public cloud. As typical to information retrieval, "Whatever you do with the document, you should do with the Query"[21]. Consequently, a user-entered query in the search box of the calendar should be tokenized, stop-word eliminated, normalized, and hashed as the original event, since he might submit a query with an arbitrary mix of tokens in lower or upper case using, but still the matching event should be retrievable. The security drawbacks are discussed and avoided partly in section3.2. Stemming can also be added as one of the steps, before hashing, but as mentioned earlier since the calendar search is not currently supporting it, we are omitting it too.

4.3 Calendar Sharing

This feature has been discussed in detail in section3.4 and section3.5. Again we will summarize here the most interesting challenges and our contributions towards a solution. Google Calendar gives its users the ability to share their calendars among each other. The shared calendar has two specific and very important features:

1. Calendar events should be visible to all sharing parties
2. Calendar events should be searchable for all sharing parties

One might argue that events should be also editable for the selected parties, but this will eventually be achieved as soon as the solution to the first important feature is found. In table 2 we will take a look at the possible combinations.

Table Summary. In this table we see, that in the event of sharing a calendar, there are two different type of parties that could be involved. The first is a calendar that has been shared between people using

Table 2: Calendar Event Property Adjusments

Sharing Party Mix	Calendar Display	Calendar Search
Same Proxy	possible, using same key on the proxy level	possible, using a proxy-bound salt
Different Proxy	possible, using same key on the calendar level	possible using a token-bound salt

the same proxy. In this case in order to display events, it suffices to have one key for the whole proxy. This is however not the approach we have chosen, since as soon as we add people from other proxies they can no more see the contents and thus we need to find ways to distribute our keys around proxies. More details on this in section 3.4. The same case applies for the hashes of the tokens and their corresponding salt value3.5. The interesting case is when one of the parties does not use a proxy, then he is not able to view or search the shared encrypted calendar.

4.4 Invitations

One of the significant features of every calendar application is the ability to send invitations to guests. As part of our Google Calendar case study we have observed the flow of an invitation message being constructed and sent. As shown in figure4.4, the end user adds guests to the invitee list, giving in their email addresses where the invitation message should be sent to. Once the user enters the invitees to the event and adds(or updates) the event. This request is being processed by Google and additionally for each invitee Google creates a unique hyperlink namely a token allowing them to slightly modify the original event by accepting or declining it. Having the message body constructed, the invitation messages are distributed to the invitees. As soon as the invitees receive the message and answer it, the original event will be updated with the number of people attending it.

Proxy Invitation Model. Using a proxy middleware in between, will of course impose a slight change

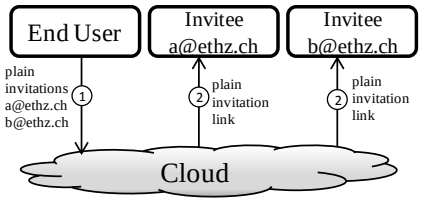


Figure 8: Invitation without Middleware Proxy

in the initial flow of the invitation procedure. The proxy invitation model is one of the contributions of this paper. As shown in figure 4.4, in this model the end user as usual initiates the request by adding guests to an event. In the first step, the plain invitation message is directed to the proxy, where the icap server encrypts the event properties and additionally changes the invitees' mail addresses to some "fake" email addresses. As expected, the encrypted message goes to the cloud service provider where additional tokens are added and the invitations are distributed to their "fake" recipients. The "fake" recipients is actually our proxy middleware that is equipped additionally with an SMTP server. As the SMTP server receives the encrypted messages from Google, it begins decrypting the event and replacing the recipients with their true receivers; Afterwards, it distributes the plain invitation messages to the original invitees.

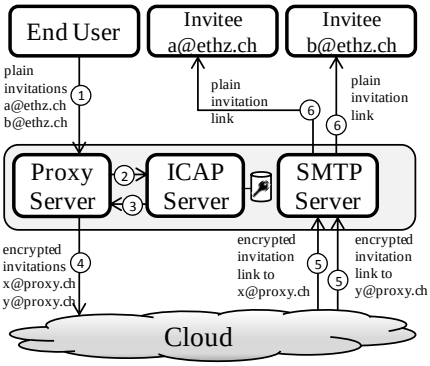


Figure 9: Invitation with Middleware Proxy

4.5 Import and Export

Another important feature of a calendar is to import and export calendars as files. In our case study, Google Calendar can import event information in iCal or CSV format, and it can be published or exported in XML, iCal, and HTML formats[4]. In order to import a calendar, the file is first opened by the proxy middleware, the content that needs to be replaced will be encrypted and replaced and then the file is sent over to Google for the final storage. In case of an export, the user makes a request to have his or her calendar exported in a particular format. The Google basically gives the link to the exported file requested by user. Here is where the middleware proxy comes in and before the user can view the file, it takes the file, replaces all its encrypted content by plain text, reconstructs the whole file and forwards it to the user.

4.6 Tasks

As part of our case study, we are also encrypting the Google tasks, since it is also integrated with the Google Calendar. Encrypting tasks are straight forward using the same fine-grained encryption mechanism described in the earlier sections.

5 Implementation

One of the advantages of the proxy middleware is its low implementation cost. As already discussed in previous sections, in order to make it work, the middleware proxy should consist of a proxy server, an ICAP server and an SMTP server. In the remainder of this section we will shortly describe the critical configurations we have made and modules we have used to accomplish the fine-grained encryption/decryption task while preserving full functionality of our case study, i.e. Google Calendar.

5.1 Proxy Middleware

Proxy Server. In our system, we have used squid 3.1[10] as the proxy server. Squid is a free, cross platform and widely used proxy server. Squid has ICAP support, which allows our scripts to perform content manipulation on the traffic captured by the proxy server. Additionally, squid has a feature called *Squid-in-the-middle SSL Bump*[10], that enables ICAP inspection of SSL traffic. This feature is one of the key requirements in our case study, since Google Calendar traffic is mostly encrypted.

ICAP Server. The ICAP server is responsible for content inspection and manipulation. We have chosen Greasyspoon[6] as our ICAP server.
SMTP Server. In order to have the invitation functionality we need also a mail server. We have chosen Postfix for this task. Postfix implements support for the Sendmail version 8 Milter (mail filter) protocol. This protocol is used by applications that run outside the MTA to inspect SMTP events (CONNECT, DISCONNECT), SMTP commands (HELO, MAIL FROM, etc.) as well as mail content (headers and body). All this happens before mail is queued[7]. This way the invitation emails can be intercepted by the proxy middleware and decrypted before being forwarded to the actual invitees.

5.2 End User

One of the main contributions and goals of this paper was to provide the end user with a privacy solution that requires little configuration effort from the end user. We have accomplished this fact by having a proxy middleware to take care of all the content extraction and encryption/decryption leaving the user only with one task, to route his or her cloud service traffic through the proxy.
To route a specific traffic through proxy, the proxy's IP address needs to be added to the browser's settings. A far better and more flexible option would be to use browser extensions to automatically switch between the direct connection and the proxy depending on the traffic requested by the user. For example, a well-known plug-in is FoxyProxy[2] which is a set of proxy management tools for Mozilla Firefox, Internet Explorer and Chrome.

6 Performance Evaluations

Though adding security on top of a cloud service will affect the performance, the added value of privacy is worth it. In the remainder of this section we will present the response time difference perceived on the user's side, as he redirects the requests through the middleware compared to the traditional approach that is directly submitting the requests to the cloud service provider. The Benchmarks were particularly challenging to conduct because the API was the Google Calendar's web interface. Thus, we had to access the browser's API and automate the benchmark process; In order to achieve the benchmark automation on the browser we have used Selenium, i.e. a suite of tools to automate web browsers across many platforms[9]. However, this tool itself appears to introduce an inherent delay for both benchmarking with and without the proxy middleware that could be considered a response time upper bound for our system. Another challenge was the network connections, link occupancy and the load on the Google's servers that could have potentially caused some of the deviations we are showing later in our graphs, therefore producing repeatable results was not straightforward.
The client machine that has been used to run experiments is a lenovo X61s, having an Intel core 2 duo cpu 1.60GHz with a RAM of 4GB on a 64-bit windows 7 operating system using Internet Explorer 9.0.3 as the emulated browser. The proxy middleware is implemented on a cluster machine having a single AMD opteron 2.4GHz processor and 2GB of RAM running Ubuntu 10.04 64-bit.
In this section, we first identify the interesting factors to vary and measure the response time by varying those. The factors we have chosen are as following:

- Functionalities: create, render, search
- Request Length: Event title length, Search query length
- Number of Events

As also mentioned above the most interesting metric to measure considering our case study is response time. Response time indicates the latency introduced by our proxy middleware as experienced by the

end user. In other words, response time is considered the time difference between the timestamp that the request is issued by the user until the response is received by the user again.
Encryption response time(Create). First of all we are showing the response time of an Event creation once with and once without the interference of the proxy middleware. We variate the request length on the x-axis, i.e. the event title to see how encryption is affecting the response time as the request length grows. To vary the length of the request, we vary the number of words in the event title. In this experiment, we isolate the length of a single word(token) to be the average word size in English, which is 5[1], then we increment the number of words in a title by one to see the effect of encryption and hashing while creating an event. As shown in figure10, as expected by increasing the number of words in an event title, the response time increases but the factor is very small, so the trend is close to constant. Also as expected there is a noticeable difference between both approaches. However, as also described earlier the response time numbers are rather an upper bound and the user experience stays almost the same. Decryption response time(Render). The second benchmark we have conducted is considering

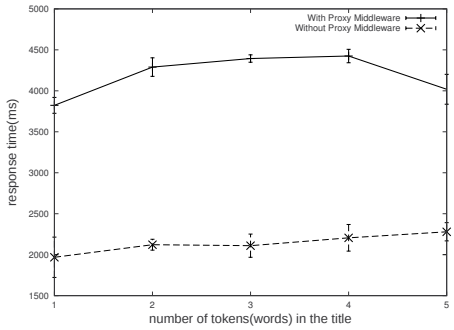


Figure 10: Event Creation overhead varying the number of words

the rendering of a calendar interface and the number of events that needs to be decrypted and displayed properly on a *month view* interface. In figure 11 we see that increasing the number of encrypted events that need to be rendered and shown on a calendar interface will affect the response time. However, we see that using the middleware proxy in some cases produces a small overhead compare to the direct connection but it will not change anything in what end user experiences. This is in fact related to the background prefetching calls done by JavaScript of the Google Calendar web interface.

Hashing and Decryption response time(Search). The goal of this benchmark is to show how fast a

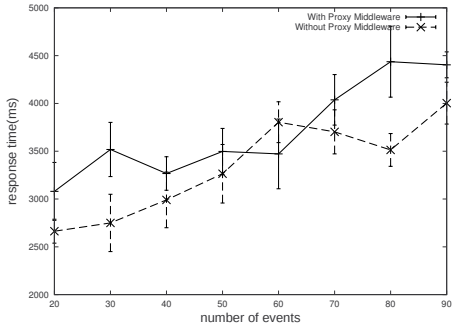


Figure 11: Response Time of Rendering varying the number of events to be displayed

search query is being replaced with its tokens being hashed and how fast the result set is being decrypted

and shown back to the user. Therefore, this benchmark targets the efficiency of the query tokenization, normalization and hashing module plus the decryption and post-filtering module. Take note that different mechanisms and algorithms are used for hashing(SHA2) and decryption(AES CBC). Before we analyze the benchmark results of figure 12, we introduce the type of events stored in the public cloud. These events follow an incremental pattern:

$$A$$
$$A\ B$$
$$A\ B\ C$$
$$A\ B\ C\ D$$
$$A\ B\ C\ D\ E$$

Therefore, the queries are constructed accordingly, i.e. the first query will be *"A"* that has a minimum hashing overhead but has a lot of hits in the public cloud that need to decrypted. The last query will have 5 tokens, which introduces the maximum hashing overhead and a minimal decryption overhead since only one entry needs to be decrypted.
As shown in figure 12 the response time is measured while variating the length of the query. As the length of the query increases (consisting of more tokens) more hashing operations needs to be done, so we can see that the response time goes up but the overhead compared to the approach without middleware is in average less than 500 milliseconds(half a second), which can be considered negligible

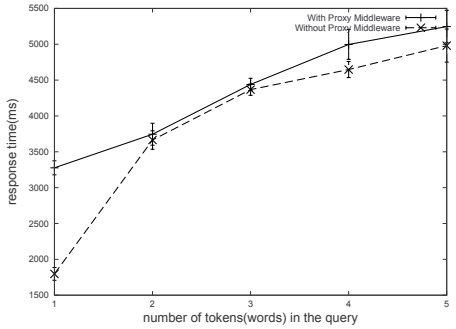


Figure 12: Response Time of Search varying the Query Length

7 Related Work

The only comparable product[17] that guarantees encryption of the Google calendar data is from IBM developersWorks and it is basically an alteration of Google Calendar Quick Add Firefox extension[23]. The development of this extension requires the system to have Perl, GnuPG and Firefox. In order to make use of this extension the user should be familiar with software development in order to diagnose problems or configuration issues specific to their setup[17]. The core idea is that in the encryption stage, the currently entered event text will be stored on disk, an external program will encrypt it, and then the encrypted file will be read in and sent to Google's servers. On decryption, each event is written to disk, unencrypted, and the plain text read back in to be displayed to the user[17]. This approach has almost no portability and makes Google calendar impossible to use on any other device unless all the required programs are installed and the files storing the encrypted events are properly synchronized. Whereas in

our approach, the encryption is completely transparent to the user.
Another related work is iDataGuard[19] which is an interoperable security middleware for untrusted Internet data storage. Their main goal is to adapt to heterogeneity of interfaces of Internet data providers and enforce security constraints. They also allow search on encrypted data using a special indexing technique. Their case study was Amazon S3 and GMail. However as we can see their granularity is at the file level whereas we provide fine-grained encryption to preserve privacy of more complicated web application than just pure data storage.
Recently the concept of "Security as a Service" is emerging. A lot of security vendors such as McAfee and M86 are now leveraging cloud based models to deliver security solutions. It has been specially successful because like any other service, security can be outsourced and taken care of by the professional security providers. These vendors mostly provide their customers with spam and virus protection services. Their architecture has similarities to our approach, since they have also a centralized solution which is a private cloud that inspects the web and email traffic of the users and applies security procedures when necessary. Nevertheless, this model is not concerning the privacy of users fine-grained data in the public clouds, but rather investigates the traffic received by the user to be free of threats.

8 Conclusion

This paper proposes a novel and general approach to solve privacy issues in web applications by having a transparent encryption layer. The goal is to preserve the advantages of cloud-based web services (i.e., low cost, no administration, great user experience) without sacrificing privacy, performance, and functionality. The paper showed how this goal could be achieved for the Google Calendar service. A proxy architecture was devised and a number of new techniques were implemented in order to preserve the Google Calendar functionality on encrypted data. In particular, a new encryption scheme was presented that allows to search on the encrypted data. Furthermore, new techniques were devised to support the other features of Google Calendar in a transparent way. Performance experiments showed that the latency impact is tolerable.
There are a number of avenues for future research. First, we would like to port our approach to mobile devices. Using the current architecture, i.e. decoupling the encryption from the end user, we can support different devices by applying a minimal setting on these devices to route their calendar traffic through our proxy and the rest is handled by our system. If these devices use the web interface to access the calendar then our current solution will work. However, most of these cloud service providers have embedded application for the mobile devices such as smart phones and tablets out there. They use another protocol than http. For example, in our case study, Google uses GSync[5]; this service uses push technology to synchronize the entries of the user on multiple devices. Thus an extension to the proxy middleware might be required to be able to intercept also between these kind of protocols.
Second, we would like to apply our approach to other Web Applications such as Gmail, Google Contacts, Google Docs, and the services provided by other providers (e.g., Microsoft, Yahoo, Amazon, etc.). There are a number of new technical challenges that need to be addressed to support all the features of these services, but the general approach and the proxy architecture should still be applicable.

References

[1] English language. http://en.wikipedia.org/wiki/English_language.

[2] Foxyproxy: Proxy management tool. http://getfoxyproxy.org/.

[3] Google calendar api. http://code.google.com/apis/calendar/data/2.0/developers_guide.html.

[4] Google calendar: Import and export. http://www.google.com/support/calendar/.

[5] Google sync. http://www.google.com/mobile/.

[6] Icap server. http://greasyspoon.sourceforge.net.

[7] Milter: Mail filter protocol. http://www.postfix.org/MILTER_README.html.

[8] Online anonymizer. http://www.onlineanonymizer.com/.

[9] Seleniumhq: Web application testing system. http://seleniumhq.org/.

[10] Squid proxy server. http://www.squid-cache.org.

[11] K. Aoki et al. Preimages for step-reduced sha-2. In *In Proc. of ASIACRYPT*, 2009.

[12] R. Chow et al. Controlling data in the cloud: outsourcing computation without outsourcing control. In *In Proc. of CCSW*, 2009.

[13] J. Daemen and V. Rijmen. *The Design of Rijndael*. Springer-Verlag New York, Inc., 2002.

[14] E. Damiani et al. Balancing confidentiality and efficiency in untrusted relational dbmss. In *In Proc. of CCS*, 2003.

[15] Elson and Cerpa. Rfc 3507: Internet content adaptation protocol (icap).

[16] H. Hacigumus et al. Providing database as a service. In *In Proc. of ICDE*, 2002.

[17] N. Harrington. Integrate encryption into google calendar with firefox extensions. http://www.ibm.com/developerworks/web/library/wa-googlecal/, 2008.

[18] G. Iachello and J. Hong. End-user privacy in human-computer interaction. *Found. Trends Hum.-Comput. Interact.*, 2007.

[19] R. Jammalamadaka et al. idataguard: an interoperable security middleware for untrusted internet data storage. In *Proceedings of the ACM/IFIP/USENIX Middleware '08 Conference Companion*.

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

[21] C. D. Manning. *Introduction to Information Retrieval*. Cambridge University Press, 2008.

[22] P. Oechslin et al. Making a faster cryptanalytic time-memory trade-off. In *In Proc. of CRYPTO*, 2003.

[23] E. Torres. Google calendar quick add: Firefox add-ons. https://addons.mozilla.org/de/firefox/addon/google-calendar-quick-add, 2007.