CloudProtect: Managing Data Privacy in Cloud Applications

Mamadou H. Diallo[1], Bijit Hore[1], Ee-Chien Chang[2], Sharad Mehrotra[1], Nalini Venkatasubramanian[1]
[1]*Donald Bren School of Information and Computer Sciences, University of California, Irvine*
[2]*School of Computing, National University of Singapore*
{*mamadoud, bhore, sharad, nalini*}*@ics.uci.edu, changec@comp.nus.edu.sg*

***Abstract*—This paper describes the *CloudProtect* middleware that empowers users to encrypt sensitive data stored within various cloud applications. However, most web applications require data in plaintext for implementing the various functionalities and in general, do not support encrypted data management. Therefore, *CloudProtect* strives to carry out the data transformations (encryption/decryption) in a manner that is transparent to the application, i.e., preserves all functionalities of the application, including those that require data to be in plaintext. Additionally, *CloudProtect* allows users flexibility in trading off performance for security in order to let them optimally balance their privacy needs and usage-experience.**

***Keywords*-privacy; security; encryption; cloud computing**

I. Introduction

A variety of end-user applications that let users manage their emails, documents, healthcare data etc., are available today over the cloud. The application-as-a-service model offers numerous advantages to end-users – e.g., 24/7 availability, accessibility from anywhere/anytime, lower cost etc. As a result, large volumes of personal and organizational data is migrating to the cloud at an amazing pace. However, one of the primary limitations of these cloud applications is that they do not support direct mechanisms for end-users to control security of their data at the service-provider. Existing practice requires users to implicitly trust the service providers and their security policies about how data is stored and accessed. Many high profile breaches over the years have established that such implicit trust is often misplaced [18]. Reasons being that, service providers are often targets of persistent hacker attacks. Furthermore, the security mechanisms followed by them remain vulnerable to insider attacks. Many surveys [19] have revealed that concerns of security and privacy are the primary barriers to adoption of cloud computing by governmental, healthcare, and financial institutions. Even if the service providers themselves were fully trusted, since the cloud applications authenticate end-users through login credentials, the data stored within the applications are vulnerable to miscreants who steal such credentials and masquerade as data owners. For example, millions of users fall victim to phishing and other password harvesting attacks each year, leading to loss of sensitive and/or confidential information in many cases [17].

One approach to alleviating security concerns is to build an end-to-end tamper-proof cloud application execution environment. Such an effort is truly monumental (if not impossible) given that such a system has to be resilient to all types of attacks. In this paper, we adopt a different strategy for protecting sensitive user data stored within cloud applications that we believe is more practical for the forseeable future. In the envisioned approach, instead of solely depending upon the security installation of the cloud service provider, end-users are empowered to implement data confidentiality directly by enabling them to encrypt their data within applications. In effect, the cloud application may not get to see the semantic content of most of the sensitive data or, will at most learn minimally about it. From the end-user's perspective, such an approach offers reduced risk of confidentiality breach since data is better protected against external hacker and phishing attacks, as well as, internal attacks. From the service-providers' perspective, it significantly reduces their risks by making data protection a joint responsibility between the users and the service providers.

While empowering end-users with mechanisms to control data access directly is attractive, it raises two major challenges in our view.

- Since applications expect the data to be in plain text form, how will the encrypted data be **stored** at the service provider **without changing the implementation** of the applications?
- Many functionalities supported by application require data to be in plain text form (e.g., language translation function available in Google Doc). How can **such functionalities continue to be made available to the end-user, even though data is encrypted**?

The first challenge can be potentially overcome using a variety of *format preserving encryption techniques* [1] developed in the literature for exactly such a purpose - viz., supporting encryption in legacy applications/systems. In any case, we envision that the bulk of sensitive user data will be contained within documents or textual fields. Most web applications support a somewhat flexible interface for textual fields and allow sufficiently large document sizes, which can help in storing encrypted representation of data. For instance,

[1]Such techniques typically generate a pseudorandom permutation using a key which is used to encode a data value into a different representation from the same domain [2].

in case of file management applications, storing an encrypted file, which might be longer, instead of the file in plain text is not a problem since most such applications do not impose a very tight length restriction on the size of file stored.

Our focus in this paper is on the second challenge – To support application functionalities, while protecting sensitive data. Our solution approach is based on the following observations:

- Encrypted server-side storage does not jeopardize functionalities built using client-side script.
- Techniques for sharing and many types of search over data can continue to be used even if data is stored in the encrypted form. (Many techniques proposed for search over encrypted data [1], [9], [5], [8] can be adopted to support search over encrypted representations without requiring changes to the application code.)
- Other functionalities at the server side (E.g., those implemented using server-side scripts such as PHP) require data to be in the plain text form[2]. However, to minimize exposure risk, the data can be stored in encrypted form until it needs to be accessed for a specific functionality/operation.

In this paper, we propose an extendible middleware called *CloudProtect*, the goal of which is to enable user-driven application level data to be stored on the service provider side in encrypted form. *CloudProtect* empowers the end users to add an extra layer of protection to their own data. *CloudProtect* transforms users' request to operate on the encrypted domain if it is possible – e.g., as in the case for search, creation of new objects, etc. However, if some operation or function execution requires access to the data in plaintext, *CloudProtect* implements a protocol to expose sensitive data for a limited duration so that the operation can be carried out on the server. Such a protocol, when invoked, can be expensive since it requires an additional round-trip with the service provider to request the encrypted data, decrypt it, and store the data back in the clear-text form. To limit such overheads, *CloudProtect*, maintains a policy that dictates which data is stored in plaintext and which is stored encrypted at the server. Such a policy, learnt from user interaction with the application, supports a tradeoff between costs/overhead incurred and the amount of (duration for which) sensitive data is exposed in plain text to the server. The policies learnt by *CloudProtect* are based on user-specified parameters that capture the degree of tolerance a user has to increased overheads as well as to potential information breach and are, furthermore, subject to human-override. Additionally, *CloudProtect* facilitates key management and secure sharing of encrypted data.

[2]While there has been significant advances in computing over encrypted representation (e.g., recent work on homomorphic encryption [7]), such techniques are still too inefficient to offer practical solutions for general computations. Furthermore, incorporating such techniques requires server-side changes, which is the very aspect we try to avoid in our approach.

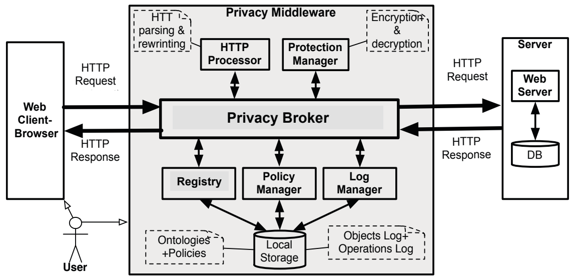

Figure 1. CloudProtect Architecture

The above mentioned features of *CloudProtect* are detailed in the rest of the paper as follows: In Section II we give an overview of the *CloudProtect* middleware architecture. In Section III we describe the exception-based mechanism for confidentiality policy management and performance-security tradeoff. In Section IV we present some evaluation results, mainly relating to tradeoff characteristics exhibited by *CloudProtect*. Finally, we discuss some related work and conclude in Sections V and VI respectively.

II. CloudProtect - An Operational View

We implemented *CloudProtect* as a Java HTTP proxy[3] server and adapted it to work with Google Calendar and Google Docs. The proxy server can play the role of an intermediary for requests from clients seeking resources from servers. It can modify both HTTP requests from clients and HTTP responses from servers, therefore making it very suitable for the implementation of our privacy middleware. The implementation is based on Sahi [20], an open source, automation and testing tool for web applications, with the facility to record and playback scripts.

The overall architecture of *CloudProtect*, shown in Figure 1. It is composed of the following modules: *HTTP Parser*, *Registry*, *Policy Manager*, *Protection Manager*, *Sharing Manager*, *Log Manager*, and *Tradeoff Analyzer*. Due to the restriction on space, we do not include the details here, and can be found in the Master's thesis [6]. Instead, in this section, we describe some of key operational features of *CloudProtect*.

Each application that is supported in *CloudProtect* needs to be registered one time, during which, the *data model* and the *function-map* are created and stored in the *Registry* database. We take an object oriented approach to modeling the user data in an application. The data object is defined by a set of attributes and methods (functionalities). Each data item is an instance of this object. For instance, a text document stored in Google Docs would be modeled as an instance of the *google_docs_data_item* class. Its title, date of creation, length etc. represent specific values the attributes

[3]A web proxy can intercept all communications in the form of HTTP messages between web clients running on a browser and web servers.

take. Operations like *translation*, *sharing* etc. are the methods associated with this class. Data representation within applications is determined by the current *data confidentiality policy* which is described next.

Adversarial model & data confidentiality policy: We assume the popular *curious adversary* model of an attacker, where the adverary is some server-side entity who is curious to glean sensitive information from the user data, but does not actively modify the data or hamper function execution.

The data confidentiality policy (or *privacy policy* for short) consists of a set of confidentiality rules which determine which attributes of the data are encrypted and which are not. We model the application data as set of objects (O). The fields (attributes) of an object is denoted by $o.A_1, \ldots, o.A_k$ where $o \in O$. The system always starts in the "maximal protection" mode which consists of one confidentiality rule for each attribute A_i of the object, denoted: $True \rightarrow Represent(A_i) = SP$.

Rule exceptions: However, each of these confidentiality rules can be relaxed over time by adding *exceptions*. They state the conditions under which a rule need not be adhered to. Exceptions are denoted as follows: $C \rightarrow Represent(A_i) \preceq level$, where the condition C denotes an atomic predicate. For instance, an atomic predicate could be "*document-title contains 'Alice'*". The variable *level* can be any element from the set {`SP`, `WP`, `NP`}. Here `SP` stands for "strong protection", referring to a standard randomized encryption algorithm (e.g., AES). `WP` stands for "weak protection" and refers to deterministic encryption algorithms that allow search directly on encrypted data. `NP` denotes "no protection" which means data is in plain text form. $Represent(A_i)$ indicates the protection to be applied on $o.A_i$ and also takes values from the same set. The predicate C typically contains a relational operator such as (=, $>$, $<$, $\leq$, $\geq$) for numeric data, and substring match for strings. Consider the rule "*True $\rightarrow$ Represent(doc.body) = SP*" and the exception "*doc.title contains 'Alice' $\rightarrow$ Represent(doc.body) $\preceq$ WP*". The exception states that if the document-title contains the word 'Alice', then the body of the document should be either deterministically encrypted (WP) or left un-encrypted (NP). Typically, many exceptions can be added to each rule, in a phase referred to as *rule relaxation* which will be discussed in greater detail in the next section.

Mandatory rules: *CloudProtect* allows an user to specify *mandatory confidentiality rules*, which must be satisfied at all times. For instance a data item should remain encrypted on the server at all times if specified in a mandatory rule, even if encryption leads to a degradation in performance.

Data Encryption: *CloudProtect*'s protection mechanism consists of a library of type-specific transformation techniques. It combines all the applicable rules to derive the appropriate protection level for a data item. We use a tagging technique similar to that used in XML, to embed the transformed data in the HTTP requests. This tagging technique is very useful for segmenting the transformed data embedded in the server-response.

Function model: The various features and functionalities of the application are captured in the function model. A function can be classified as one of *CRUD* (*create*, *read*, *update*, *delete*), *search*, *sharing* or *other* functions. Every invocation of a function (other than *create*) consists of three steps - The first is to specify a subset of all currently instantiated objects S; the second is computing the function against each one of the selected objects and third is displaying the final output to the user. The first step is either specified using a property such as, "retrieve all files with the word *account* in them" or specified explicitly, such as, "share all files in directory named *songs* with Bob". We call this the *selection function*. When a function is invoked, *CloudProtect* determines whether it can be directly evaluated against the encrypted data on the server or not? (This classification is also stored ahead of time in the function-map within the *Registry* module). Luckily, all the CRUD functions, as well as sharing and searching can be executed against encrypted data in most cases. We describe the implementation of search and sharing functions in detail below. For other functions some important implementation decisions need to be made based on the number of objects satisfying the selection criteria (step 1) as well as their representations on the server (i.e., plaintext or encrypted). *CloudProtect* stores a summarized information about the data representation on the server based on the current confidentiality policy of the user. Using this information, it decides if the current function invocation is going to access only unencrypted data, encrypted data or both kinds of data on the server. (Note, this summary information may not allow the middleware to make the determination with perfect accuracy, in which case, it decides that the function potentially needs to be evaluated against both kinds of data). If it is determined that the function needs to be evaluated only against plaintext data, then, the middleware simply sends the original request without any changes to the server. Alternatively, if the function needs to be evaluated against data items that are currently encrypted, then *CloudProtect* raises an *exception* and halts the execution to explicitly seek permission for decrypting the data. It may also warn the user about the potential delays that may be incurred in case a large number of items may have to be decrypted before the function can be executed. The user has the ability to abort the request at this point if he so desires.

Exception handling: Exception handling consists of two steps - (i) *CloudProtect* logs the information about the function and its parameters that led to the exception being raised; (ii) It generates a request to get the encrypted parameters of the function (causing the exception), decrypts and writes them back on the server and, then sends the original http request to the server. The exception based approach employed in *CloudProtect* allows all features of the application to be preserved. However, it adds an overhead in terms of either performance or interruption (to seek user authorization) or both. This overhead can be significant depending upon the size of the data involved. After decrypting a data object for function evaluation, *CloudProtect* can re-encrypt it as soon as the task has been accomplished. This maintains the consistency of the data representation with the specified confidentiality policy. Alternatively, *CloudProtect* can leave the data in plain text if it is accessed often and *relax* the corresponding confidentiality rule in the policy. Such relaxation is at the center of the "rebalancing" feature of *CloudProtect* which is described in greater detail in the next section.

Search: Enabling efficient search over strongly encrypted data is a difficult problem and none of the approaches proposed in literature (e.g., [15], [4]) scale well with size. The main reason being that, requirements of semantic security does not allow the encrypted data to be indexed. Deterministic encryption techniques [1], while not as secure as their non-deterministic counterparts, admit indexing and can support fast searches (in time logarithmic in size of the data set). *CloudProtect* gives users the option to deterministically encrypt the attribute values if they wants it to be searchable. For many applications deterministic encryption may provide sufficient security. In addition, highly confidential data items/values can be non-deterministically encrypted at user's discretion. There are challenges that still need to be overcome before enabling efficient search over data within such applications as illustrated below.

Consider the case of search over a document repository like Google Docs. At any given time, a document might belong to the class of encrypted documents (D_E) or to those in plaintext (D_P). *CloudProtect* stores a summary information about the sets D_E and D_P using a *Bloom Filter* [3] data structure. So, when the user issues a search on keyword k, the system quickly checks whether $k \in D_E$ and $k \in D_P$? If it is in neither, then, it does not issue the search command to the server at all. Else, if it is in D_E and not in D_P, it generates the *trapdoor* corresponding to k and issues it to the server as the search parameter. Alternately, if it determines that k is in D_P and not in D_E, then it simply issues k as the search parameter. Finally, if it determines that k is present in both the sets, then it issues both the queries. The final results are simply concatenated to one another and shown to the user [4]. We note, especially in the last case, there is an unwanted disclosure that happens due to two queries being issued at the same time, i.e., the server gets to know the trapdoor corresponding to the keyword. As its side-effect, the server also gets to know which documents have the keyword k in common. In fact, how to prevent such exposure from query patterns in searchable encryption schemes (both deterministic and non-deterministic ones) is still an open problem. Currently, this cannot be prevented and we feel that such disclosure maybe acceptable in many situations. A practical solution that avoids such disclosure is implementable when the set D_E is small - D_E can be cached on the client machine for the active duration and all search queries against it could be performed at the client itself.

[4]It is possible to try and merge the returned lists of the two queries in the middleware using some ranking function before displaying it to the user. However, at this point, we do not implement the merging in the system to save on post-processing time.

Sharing & collaboration: Sharing of static data with other users of *CloudProtect* can be done quite efficiently. Each object is associated with an encryption key and each user has a unique ID within *CloudProtect*. *CloudProtect* enables each user to generate a public/private key pair. When one user (of *CloudProtect*) shares an object with another user, the encryption key is shared through email message. To make the email a secure channel for key distribution, the sender encrypts the message using the receiver's public key. In the case where the recipient does not use *CloudProtect*, the owner needs to decrypt the object before sharing it. Real time collaboration (as say, supported in Google Docs) is substantially more challenging to implement if one does not want to give the server access to the plaintext data. However, this is not a problem if trusting the server for the short time is acceptable, specifically, while the data is being actively modified by the owner and/or collaborators. It only requires to be encrypted when it is not being actively modified, and that is quite straight forward to achieve by sharing a common *group key* between the authorized collaborators.

Rebalancing: After using the *CloudProtect* for a period of time, it is possible that the user experiences a high rate of interrupts and or degradation in performance due to large number of objects that have to be dynamically decrypted to support function request by users. When the conditions specified by the user are met, the rebalancing phase is invoked (or it may be explicitly initiated by the user at any time). The goal of the rebalancing is to relax the set of privacy policies so as to improve user's experience, based on the logged user workload (history of user- executed operations), and the current state of the DB (information about the current representations of objects a the cloud side).

Next, we describe the policy rebalancing feature of the *CloudProtect* system which is key to its adaptability across different applications.

III. Balancing Privacy, Usability, and Efficiency

CloudProtect tries help a user find the right balance between his/her privacy requirements and user-experience. The basic premise is that user-experience degrades if there are too many interruptions and excessive delay in function execution. The system default is to strongly encrypt all attributes of an object. As the user profile builds up, the system relaxes the default policy (of maximal encryption) by adding exceptions. The key step in this process is the mechanism that, based on the logged activities, determines the right set of exceptions to add. The exceptions are automatically generated using the function parameters. For example, if the function is to translate a document with ID d1, then an exception that would remove the encryption for this document can be "*doc.id = 'd1'* $\rightarrow$ *Represent(doc.body)* $\preceq$ *NP*". In this section, we describe a formulation capturing the tradeoff between privacy, usability, and efficiency, and describe the algorithmic approach taken by *CloudProtect* to achieve the desired degree of tradeoff.

A. Privacy Policy Relaxation/Specialization

The central mechanism used in the rebalancing procedure is that of *policy relaxation*. Let us reconsider the example from previous section, with the rule "*true* $\rightarrow$ *Represent(doc.body) = SP*". Let the following two exceptions be added to this rule: *doc.title contains 'Alice'* $\rightarrow$ *Represent(doc.body)* $\preceq$ *WP* and *doc.body contains 'Bob'* $\rightarrow$ *Represent(doc.body) = NP*. Then, documents that contain 'Alice' in the title but do not have 'Bob' in the body, can remain either in WP or NP state. If the document has the word 'Bob' in the body of the text, they should remain in plaintext. In case of two or more "competing" exceptions to a rule, the exception that leads to the greatest relaxation of the rule will be implemented. For example, in a situation where the first exception has a "=" sign and the document-title contains 'Alice' and document-body contains 'Bob', the system would implement the second exception.

B. Minimizing Interruptions and Costs

Given a privacy policy P, the set of objects O, and an operation w, let $\mathit{cost}(w, P, O)$ denote the execution cost for operation w. This is estimated by a combination of communication and computation overheads involved. We define the cost overhead incurred in executing w as follows:

$$\mathit{cost\text{-}oh}(w, P, O) = \frac{\mathit{cost}(w, P, O) - \mathit{cost}(w, \perp, O)}{\mathit{cost}(w, \top, O) - \mathit{cost}(w, \perp, O)} \quad (1)$$

where $\mathit{cost}(w, \perp, O)$ corresponds to the lowest execution cost (floor) when none of the attributes of any object in O are encrypted, and $\mathit{cost}(w, \top, O)$ corresponds to the highest execution cost (ceiling) wherein all attributes of every object in O are non-deterministically encrypted.

Similarly, we define $\mathit{int}(w, P, O)$ as the interruption, which takes the value of 1 if an interruption is initiated when w is executed, and 0 otherwise. Since lowest interruption ($\mathit{int}(w, \perp, O)$) is always 0 and the highest ($\mathit{int}(w, \top, O)$) is always 1, the interruption overhead can be reduced to just $\mathit{int}(w, P, O)$.

$$\mathit{int\text{-}oh}(w, P, O) = \mathit{int}(w, P, O) \quad (2)$$

Ideally, given O and P, we would like to find a "minimally" relaxed policy P^* s.t. the expected interruption overhead $\mathit{int\text{-}oh}(W, P, O)$ and the expected cost overhead $\mathit{cost\text{-}oh}(W, P, O)$ are below acceptable (user-specified) thresholds.

As the workload distribution is difficult to model in general, we treat W, the set of operations executed within a previous time window, as the sample *workload*. We estimate the average cost overhead and average interruption overhead based on W as follows:

$$\mathit{int\text{-}oh}_{avg}(W, P, O) = \frac{1}{|W|} \sum_{w \in W} \mathit{int}(w, P, O) \quad (3)$$

$$\mathit{cost\text{-}oh}_{avg}(W, P, O) = \frac{1}{|W|} \sum_{w \in W} \mathit{cost\text{-}oh}(w, P, O) \quad (4)$$

We use the "reduction of objects' protection level" as the metric to quantify the extent of policy relaxation. Specifically, if an object's attribute $o.A$ is deemed to be protected at level s_1 under the original policy P, and is lowered to level s_2 under the relaxed policy P^*, then we denote the protection reduction on $o.A$ by the number $level(s_1) - level(s_2)$, where $s_1, s_2 \in \{0(\mathtt{NP}), 1(\mathtt{WP}), 2(\mathtt{SP})\}$. (Since P^* is a relaxed policy, we always have $level(s_1) \geq level(s_2)$). For a set of objects O, we define the protection reduction to be the sum total of reductions across every attribute of all objects in O. Let us denote this quantity as $\mathit{PrivRed}(P, P^*, O, W)$.

Now, the Minimum Relaxation problem can be formally defined as follows.

Definition: *Minimum Relaxation*. Given a set of objects O, workload (sequence of operations) W, a set of encryption rules P, and parameters α and β, the *Minimum Relaxation* problem is the following minimization problem.

$$\mathit{MinRelax}(W, P, O) = \mathit{ArgMin}_{P^*}(\mathit{PrivRed}(P, P^*, O, W))$$

such that

$$\mathit{int\text{-}oh}_{avg}(W, P^*, O) \leq \alpha \quad (5)$$

$$\mathit{cost\text{-}oh}_{avg}(W, P^*, O) \leq \beta \quad (6)$$

We call α the interruption threshold and β the cost threshold.

C. Solution for the $MinRelax$ problem

$MinRelax$ can be formulated as a selection problem[5] where the goal is to select the rules and their relaxations that reduce average cost and interruption overheads to below the specified thresholds, while paying "minimum price" in terms of reduction in the sum total protection of objects. In fact, we can prove that the problem is NP-hard by reduction from the *Budgeted Maximum Coverage (BMC)* problem [11]. The reduction is omitted due to lack of space.

Greedy algorithm: Given the complexity of the problem, *CloudProtect* uses a greedy heuristic based approach. The average interruption overhead ($\mathit{int\text{-}oh}_{avg}$) is easy to compute from equation 2. To estimate the average cost-overhead ($\mathit{cost\text{-}oh}_{avg}$) we need to estimate the quantity $\mathit{cost}(w, P, O)$, $\mathit{cost}(w, \perp, O)$ and $\mathit{cost}(w, \top, O)$ for every operation w in the workload. Each w is associated with a computation and

[5]The problem can be visualized better using as a particular graph-based model, but it is omitted here due to lack of space. The interested reader can look up the details in the MS thesis [6].

communication costs. We simply measure the time it takes to complete the operation w against a set of objects O and a policy P. This includes the communication and computation costs incurred for decryption of encrypted objects retrieved to the middleware before displaying the final result to the user.

Relaxing policy by adding exceptions: The $MinRelax$ algorithm (Figure 2) starts with the policy P, workload W and parameters α and β. Then, it tries to bring down the average interruption and cost overheads for W to below the specified thresholds, by greedily choosing an operation w_{best} in iteration that maximizes the *benefit* per unit of *privacy cost*. We define *benefit(w)* as the reduction in the value of the sum (*cost-oh*$_{avg}$ + *int-oh*$_{avg}$) when the operation w is "eased". That is, add appropriate exceptions to confidentiality rules such that, executing w no longer generates an interrupt or needs any decryption operations in the middleware. At the end of each iteration, w_{best} is removed from W and the appropriate set of exceptions are generated using the parameters of w_{best}, which are then added to the rules which affect the objects associated with w_{best}. For example, say, Alice is the user of Google Docs who triggers the $MinRelax$ algorithm. If the best operation to ease from her workload in the first iteration is the "share document d with Bob", then the following exception will be generated: *doc.recipient = 'Bob'* $\rightarrow$ *Represent(doc.body) = NP*. The final relaxed policy that is returned, is the one computed in the last iteration. *MinRelax* has a worst case time complexity of $O(|W|^3)$, but on an average terminates in time $O(|W|^2)$. It requires $O(|P|+|W|)$ space, where $|P|$ denotes the number of distinct rules in the policy P.

Tightening policy by dropping exceptions: The *CloudProtect* system can also reconfigure to a higher confidentiality setting by dropping some of the exceptions added to a rule. A simple modification of the algorithm shown in figure 2 is used to achieve this. Given the current policy P and the set of exceptions E, the algorithm greedily determines a maximal set of exceptions to drop while still keeping the number of interruptions and costs below their respective thresholds.

Example: Let us suppose that Google Docs contains three strongly protected documents with same privacy level (*level = 2*), but different sizes: *(d1, 100KB)*, *(d2, 200KB)*, *(d3, 300KB)*. Let the following 5 functions comprise the workload: *share(d1, Bob)*, *share((d1, d3), Alice)*, *translate(d3, French)*, *search('Report')* (i.e., search all documents with keyword 'Report'), and *publish(d3)*. Each function generates one interruption and incurs an execution cost, which for simplicity, we will take to be proportional to the total size of the documents involved. Also, let us consider only the interruption threshold, and let it be set to 3. Table I summarizes the information recorded by *CloudProtect*. Table II shows a partial list of all the possible selections of functions that bring the total interruptions below the threshold. Observe

```
MinRelax ($P, W, O, \alpha, \beta$) {
    $W^* = W$; $P^* = P$;
    $int\_oh_{avg}$ = $getIntOverhead(P, W, O)$;
    $cost\_oh_{avg}$ = $getCostOverhead(P, W, O)$;
    while ($int\_oh_{avg} > \alpha$ & $cost\_oh_{avg} > \beta$ & $W^* \neq \emptyset$)
        $CR_{best}$ = 0; /*best benefit per unit privacy cost*/
        $T_{oh} = int\_oh_{avg} + cost\_oh_{avg}$; /*total overhead*/
        for each ($w \in W^*$)
            $W^* = W^* \setminus \{w\}$;
            $O_w = getObjectsAccessed(w, O)$;
            $E = generateExceptions(P^*, w, O)$ ;
            $P^{*'} = addExceptions(P^*, E)$;
            $PC_w = getPrivacyReduction(P^*, P^{*'}, O)$;
            $int\_oh' = getIntOverhead(P^{*'}, W, O)$;
            $cost\_oh' = getCostOverhead(P^{*'}, W, O)$;
            $T'_{oh} = int\_oh' + cost\_oh'$;
            $B_w = T_{oh} - T'_{oh}$; /*benefit of removing w*/
            $CR_w = \frac{B_w}{PC_w}$; /*benefit per unit privacy cost*/
            if ($CR_{best} < CR_w$)
                $w_{best} = w$;
                $CR_{best} = CR_w$;
            endif
            $W^* = W^* \cup \{w\}$; /*put back w*/
            $P^* = removeExceptions(P^{*'}, E)$;
        end for
        $W^* = W^* \setminus \{w_{best}\}$;
        $E_{best} = generateExceptions(w_{best}, O)$;
        $P^* = addExceptions(P^*, E_{best})$; /*modify policy*/
        $int\_oh_{avg} = getIntOverhead(P^*, W, O)$;
        $cost\_oh_{avg} = getCostOverhead(P^*, W, O)$;
    end while
    return $P^*$; }
```

Figure 2. Policy Relaxation Algorithm by Adding Exceptions

Table I
OPERATIONS LOG

Functions	Objects	Params	IntOh	CostOh	PLevel
share	d1	Bob	1	100	2
share	d1, d3	Alice	1	400	4
translate	d3	French	1	300	2
search	d1, d2, d3	Report	1	600	6
publish	d3	Web	1	300	2

that, selecting any of the functions *share(d1, d3)*, *translate(d3)*, *search(Report)*, or *publish(Web)* would satisfies the interruption threshold constraint, but the cost reductions (CostRed) and the privacy reductions (PrivRed) are different. For instance, *search(Report)* reduces the interruptions by 5 and costs by 600, but sacrifices the privacy level by 6 (all the privacy). On the other hand, *translate(d3)* reduces the interruptions by 2 and costs by 300, but gives up the privacy level only by 2. This example illustrates the tradeoffs that can be played between the interruptions (usability), costs (efficiency), and privacy levels (privacy).

IV. CLOUDPROTECT EVALUATION

We evaluated *CloudProtect* from two perspectives - (i) Performance; (ii) Analysis of the tradeoff between privacy, usability, and efficiency using the *MinRelax* algorithm as the policy relaxation mechanism. However, in this paper

Table II
OPERATIONS SELECTION

Selection	IntRed	CostRed	PrivRed	Exceptions
share(d1)	1	100	2	1
share(d1, d2)	3	400	4	1
translate(d3)	2	300	2	1
search('Report')	5	600	6	1
publish('Web')	2	300	2	1

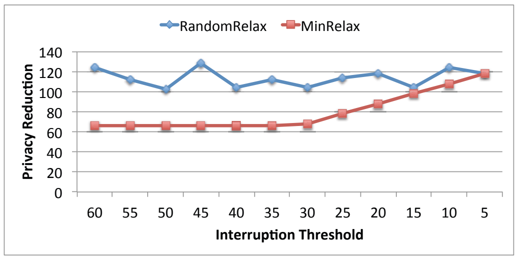

Figure 3. Varying interruption threshold

we only present the results of the tradeoff evaluation. The performance results can be found in the MS thesis [6].

Tradeoff Analysis: The main experiment was to analyze the performance of the rebalancing algorithm, *MinRelax*. We compared $MinRelax$ with two other algorithms, a basic algorithm that selects the operations to remove randomly (*RandomRelax*), and an optimal algorithm that uses a brute-force search mechanism to decide which operations to remove (*OptimalRelax*). We use the number of exception rules, the number of interruptions, the execution cost, and the privacy cost as the main metrics. Since the *OptimalRelax* is an NP-Hard search problem and is only feasible for small input instances, we do not plot its values. We considered Google Docs for these experiments and used the following 5 functions to randomly generate the workload consisting of 64 operations: create, upload, search, share, and translate documents.

Varying interruption threshold: In the first experiment, we note the privacy-reduction as the "number of interruptions" threshold is varied (decreased). In the plot of Figure 3, we see that *MinRelax* handily beats the *RandomRelax* algorithm all through. Interestingly, while *RandomRelax* may lead to greater privacy loss for higher thresholds than for lower ones, the privacy loss in *MinRelax* increases monotonically which decreasing value for the threshold.

Varying execution cost threshold: In the second experiment, we used only the execution cost threshold as the control variable. The execution cost is the time taken (in seconds) to execute the whole workload . The results are shown in Figure 4. However, in contrast to the previous plot, the privacy cost incurred seems to rise monotonically for both *MinRelax* and *RandomRelax* in this case. However, as in the previous case, our algorithm outperforms *RandomRelax* for all values of the threshold.

Varying Interruption and Cost Threshold: In the third experiment, we again compare the performance of *MinRelax*

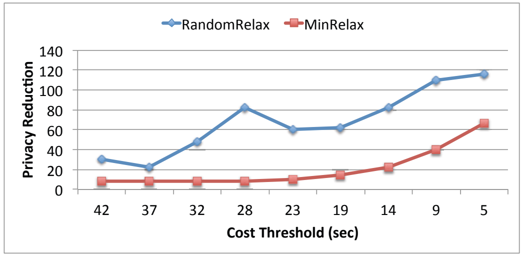

Figure 4. Varying execution cost threshold

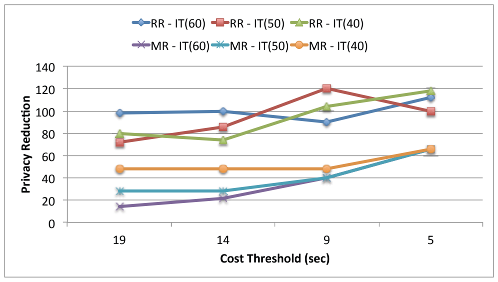

Figure 5. Varying interruption & execution cost thresholds

and *RandomRelax* with respect to the privacy reduction. However, this time we used both, the number of interruptions and the execution cost thresholds, and varied both. We fixed the interruption cost thresholds to 60, 50 and 40 for both the algorithms. The results of the experiment are plotted in Figure 5. We see that the *MinRelax* algorithm performs better than *RandomRelax* for all combinations of the two cost thresholds.

Delay: In this experiment, we compare the execution time of the three algorithms, *MinRelax*, *RandomRelax* and *OptimalRelax*. Since the computation overhead of *OptimalRelax* increases exponentially with the size of the workload, we scale the size only up to 15. As in the second experiment, we used the cost threshold (total time to execute the 15 operations) as the stopping criterion and measured the time (in seconds) it took for the algorithms to terminate (note, in each iteration of the algorithm, all operations that remain in W^* are executed, and the corresponding time delay is measured). The results of the experiment are shown in Figures 6. As can be seen, the difference in execution time is substantial between *MinRelax* and *OptimalRelax* as the cost threshold decreases. The latter quickly becomes infeasible for larger workloads, while the other two scale linearly.

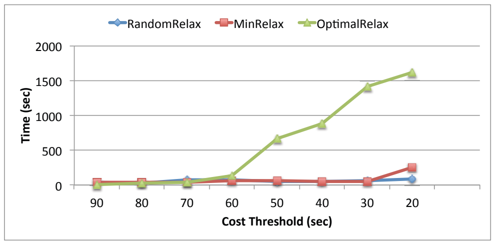

Figure 6. Running time of algorithms

V. Related Work

CloudProtect is related to recently proposed system *Silverline* [13] that also aimed to support encrypting user data in cloud applications [6]. *Silverline* introduced the concept of *functional encryption* wherein the system identifies all data items that can be encrypted in a manner that is transparent to the application. It uses dynamic program analysis to identify functionally encryptable data in a web-based application and propose a encryption key assignment and sharing solution that enables people access parts of the data which they have authorization for. However, *Silverline* takes a rather conservative approach by encrypting only data that is never accessed in plain text. *Silverline* assumes that any data that is accessed by a function initiated by the user cannot ever be encrypted. In contrast to the approach taken by *CloudProtect*, they do not consider any user-level adaptations.

User adaptation has also been considered in context of browser cookie management by the Doppelganger system [14]. Since users do not like to be constantly interrupted with questions or alerts, Doppelganger facilitates easy creation and enforcement of fine-grained, privacy-preserving browser cookie policies.It makes automated determination of cookies' value and uses client-side parallelism to explore alternate policies in the background, trying to find those which result in a positive tradeoff between privacy loss and functionality gain.

Another application that protects user data on the cloud, is *flyByNight* [12]. Similar to *CloudProtect*'s goal of making non-essential user data (from application functionality standpoint) opaque to the cloud application, *flyByNight* tries to make user communications opaque to social networking websites such as FaceBook. The proposed architecture is based on the social network infrastructure, and enables encryption of messages such that they remain hidden from the FaceBook server. Unlike *CloudProtect* however, the system is explicitly designed for Facebook.

Secure data outsourcing [15], [4], [9], [10], [16] has been a very active area of research for a long time. However, the approaches therein, have only considered the "cooperative server" model, i.e., where one has the freedom to design the server side and implement specialized data storage and search mechanisms. In contrast, *CloudProtect* considers the problem of data confidentiality in web applications where server-side code is not modifiable, i.e., the server is "non-cooperative". We refer the reader to more detailed survey of this area for further information [8].

VI. Conclusion and Future Work

Ensuring privacy and confidentiality of data stored within web applications is challenging. The difficulty stems from the fact that service providers do not provide any support for protecting users data. In this study, we propose a policy-based privacy middleware called *CloudProtect* for supplementing existing web applications with a mechanism for protecting users data, with minimal disruption of user-experience. We take a"pragmatic" approach to data confidentiality, where the data may be partially/temporarily exposed as long as it does not violate the user-specified privacy policy. Such an approach, which represents a significant departure from prior work, has numerous advantages. In addition to improved efficiency, limited disclosure allows for "value-added" services that are key benefits of today's personal data outsourcing services.

We implemented a prototype of the privacy middleware with two applications, namely, Google Calendar and Google Docs. The middleware supports most of the functionalities provided by these applications including upload, search, and sharing operations. We performed a number of experiments to analyze the tradeoff between privacy, usability, and efficiency.

Currently, the privacy middleware is implemented as a desktop application, which the user has to install on a local machine. While this implementation is simple to use, it is not very suitable for mobility. A web-based implementation of the middleware will address this issue, and we intend to investigate the feasibility of such an implementation.

[6]*Silverline* and CloudProtect were both conceptualized developed independently, though the fundamental motivation of both is similar.

References

[1] M. Bellare, A. Boldyreva, and A. O'Neill. Deterministic and efficiently searchable encryption. In *CRYPTO*, pages 535–552, 2007.

[2] M. Bellare, T. Ristenpart, P. Rogaway, and T. Stegers. Format-preserving encryption. In *Selected Areas in Cryptography*, pages 295–312, 2009.

[3] B.H. Bloom, Space/time trade-offs in hash coding with allowable errors, Communications of the ACM 13 (7): 422426.

[4] D. Boneh, G. D. Crescenzo, R. Ostrovsky, and G. Persiano. Public key encryption with keyword search. In *EUROCRYPT*, pages 506–522, 2004.

[5] Y.-C. Chang and M. Mitzenmacher. Privacy preserving keyword searches on remote encrypted data. In *ACNS*, pages 442–455, 2005.

[6] M.H. Diallo CloudProtect: A middleware for managing privacy in web and cloud data services. M.S. Thesis, UCI, 2012.

[7] C. Gentry Fully homomorphic encryption using ideal lattices. STOC 2009.

[8] H. Hacigümüs, B. Hore, B. R. Iyer, and S. Mehrotra. Search on encrypted data. In *Secure Data Management in Decentralized Systems*, 2007.

[9] H. Hacigümüs, B. R. Iyer, C. Li, and S. Mehrotra. Executing sql over encrypted data in the database-service-provider model. In *SIGMOD*, 2002.

[10] B. Hore, S. Mehrotra, and G. Tsudik. A privacy-preserving index for range queries. In *VLDB*, 2004.

[11] S. Khuller, A. Moss, and J. Naor. The budgeted maximum coverage problem. *Inf. Process. Lett.*, 70(1):39–45, 1999.

[12] M. M. Lucas and N. Borisov. Flybynight: mitigating the privacy risks of social networking. In *WPES*, 2008.

[13] K. P. N. Puttaswamy, C. Kruegel, and B. Y. Zhao. Silverline: Toward data confidentiality in third-party clouds. In *ACM SOCC*, 2011.

[14] U. Shankar and C. Karlof. Doppelganger: Better browser privacy without the bother. In *ACM CCS*, 2006.

[15] D. X. Song, D. Wagner, and A. Perrig. Practical techniques for searches on encrypted data. In *IEEE Symposium on Security and Privacy*, 2000.

[16] H. Wang and L. V. S. Lakshmanan. Efficient secure query evaluation over encrypted xml databases. In *VLDB*, pages 127–138, 2006.

[17] D. D. Zero Day Ryan Naraine. How many people fall victim to phishing attacks?, 2009.

[18] http://www.securityweek.com/survey-more-attacks-coming-outsiders-insider-attacks-more-costly

[19] http://gcn.com/microsites/2011/cloud-computing-download/cloud-computing-application-development.aspx

[20] Sahi: A web automation and test tool. http://sahi.co.in/w/.