Silverline: Toward Data Confidentiality in Storage-Intensive Cloud Applications

Krishna P. N. Puttaswamy
Computer Science Dept.
UC Santa Barbara
Santa Barbara, CA 93106
krishnap@cs.ucsb.edu

Christopher Kruegel
Computer Science Dept.
UC Santa Barbara
Santa Barbara, CA 93106
chris@cs.ucsb.edu

Ben Y. Zhao
Computer Science Dept.
UC Santa Barbara
Santa Barbara, CA 93106
ravenben@cs.ucsb.edu

ABSTRACT

By offering high availability and elastic access to resources, third-party cloud infrastructures such as Amazon EC2 are revolutionizing the way today's businesses operate. Unfortunately, taking advantage of their benefits requires businesses to accept a number of serious risks to data security. Factors such as software bugs, operator errors and external attacks can all compromise the confidentiality of sensitive application data on external clouds, by making them vulnerable to unauthorized access by malicious parties.

In this paper, we study and seek to improve the confidentiality of application data stored on third-party computing clouds. We propose to identify and encrypt all *functionally encryptable* data, sensitive data that can be encrypted without limiting the functionality of the application on the cloud. Such data would be stored on the cloud only in an encrypted form, accessible only to users with the correct keys, thus protecting its confidentiality against unintentional errors and attacks alike. We describe *Silverline*, a set of tools that automatically 1) identify all functionally encryptable data in a cloud application, 2) assign encryption keys to specific data subsets to minimize key management complexity while ensuring robustness to key compromise, and 3) provide transparent data access at the user device while preventing key compromise even from malicious clouds. Through experiments with real applications, we find that many web applications are dominated by *storage and data sharing* components that do not require interpreting raw data. Thus, Silverline can protect the vast majority of data on these applications, simplify key management, and protect against key compromise. Together, our techniques provide a substantial first step towards simplifying the complex process of incorporating data confidentiality into these storage-intensive cloud applications.

Categories and Subject Descriptors

K.6.5 [**Security and Protection**]: Unauthorized Access

General Terms

Security, Design, Experimentation

SOCC'11, October 27–28, 2011, Cascais, Portugal.

Keywords

Cloud computing, Data confidentiality, Program analysis

1. INTRODUCTION

Third-party computing clouds, such as Amazon's EC2 and Microsoft's Azure, provide support for computation, data management in database instances, and Internet services. By allowing organizations to efficiently outsource computation and data management, they greatly simplify the deployment and management of Internet applications. Examples of success stories on EC2 include Nimbus Health [4], which manages and distributes patient medical records, and ShareThis [5], a social content-sharing network that has shared 430 million items across 30,000 websites.

Unfortunately, these game-changing advantages come with a significant risk to data confidentiality. Using a multi-tenant model, clouds co-locate applications from multiple organizations on a single managed infrastructure. This means application data is vulnerable not only to operator errors and software bugs in the cloud, but also to attacks from other organizations. With unencrypted data exposed on disk, in memory, or on the network, it is not surprising that organizations cite data confidentiality as their biggest concern in adopting cloud computing [17, 33, 48]. In fact, researchers recently showed that attackers could effectively target and observe information from specific cloud instances on third party clouds [40]. As a result, many recommend that cloud providers should never be given access to unencrypted data [6, 41].

Organizations can achieve strong data confidentiality by encrypting data before it reaches the cloud, but naively encrypting data severely restricts how data can be used. The cloud cannot perform computation on any data it cannot access in plaintext. For applications that want more than just pure storage, *e.g.*, web services that serve dynamic content, this is a significant hurdle. There are efforts to perform specific operations on encrypted data such as searches [1, 11, 12, 15, 25, 30, 43, 44]. A recent proposal of a fully homomorphic cryptosystem [24] even supports arbitrary computations on encrypted data. However, these techniques are either too costly or only support very limited functionality. Thus, users that need real application support from today's clouds must choose between the benefits of clouds and strong confidentiality of their data.

In this paper, we take a first step towards improving data confidentiality in cloud applications, and propose a new approach to balance confidentiality and computation on the cloud. Our key observation is this: in applications that can benefit the most from the cloud model, the majority of their computations handle data in an opaque way, *i.e.* without interpretation. For example, a `SELECT` query looking for all records matching userID `'Bob'` does not need to interpret the actual string, and would succeed if the string were encrypted, as long as the value in the query matched the encrypted string. We refer to data that is never interpreted (a.k.a. used in a computation) by the application as *functionally encryptable*, *i.e.* encrypting them does not limit the application's functionality. Consider for example, ShareThis [5], which uses Amazon's SimpleDB for attribute search and list management. Its documentation states "aggregators sum instances of each event type by publisher and update SimpleDB on day boundaries." With ShareThis, users search for events by matching specific attributes, but the cloud does not interpret the value of the attributes and simply treats them as opaque data. Similarly, the report generator functionality only computes count of events of a particular type but ignores the actual value of those "opaque" types. Similar operations are common in other applications like social networks or shopping carts.

Leveraging the observation that certain data is never interpreted by the cloud, our key step is to split the entire application data into two subsets: functionally encryptable data, and data that must remain in plaintext to support computations on the cloud. As we later show, a large majority of data in many of today's applications is functionally encryptable. As shown in Figure 1, such data would be encrypted by users before uploading it to the cloud, and it would be decrypted by users after receiving from the cloud. While this idea sounds conceptually simple, realizing it requires us to address three significant challenges: 1) identifying functionally encryptable data in cloud applications, 2) assigning (symmetric) encryption keys to data while minimizing key management complexity and risks due to key compromise, and 3) providing secure and transparent data access at the user device.

Identifying functionally encryptable data. The first challenge is to identify data that can be functionally encrypted without breaking application functionality. To this end, we present an automated technique that marks data objects using tags and tracks their usage and dependencies through dynamic program analysis. We identify functionally encryptable data by discarding all data that is involved in any computations on the cloud. Naturally, the size of this subset of data depends on the type of service. For example, for programs that compute values based on all data objects, our techniques will not find any data suitable for encryption. In practice, however, results show that for many applications, including social networks and message boards, a large fraction of the data can be encrypted.

Encryption key assignment. Once we identify the data to be encrypted, we must choose how many keys to use for encryption, and the granularity of encryption. In the simplest case, we can encrypt all such data using a single key, and share the key with all users of the service. Unfortunately, this has the problem that a malicious or compromised cloud could obtain access to the encryption key, *e.g.* by posing as a legitimate user, or by compromising or colluding with an existing user. In these cases, confidentiality of the entire dataset would be compromised. In the other extreme, we could encrypt each data object with a different key. This increases robustness to key compromise, but drastically increases key management complexity.

Our goal is to automatically infer the right granularity for data encryption that provides the best tradeoff between robustness and management complexity. To this end, we partition the data into subsets, where each data subset is accessed by the same group of users. We then encrypt each data subset using a different key, and distribute keys to groups of users that should have access (based on the desired access control policies). Thus, a malicious or buggy cloud that compromises a key can only access the data that is encrypted by that key, minimizing its negative impact. We introduce a dynamic access analysis technique that identifies user groups who can access different objects in the data set. In addition, we describe a key management system that leverages this information to assign to each user all keys that she would need to properly access her data. Since key assignment is based on user access patterns, we can obtain an assignment that uses a minimal number of encryption keys necessary to "cover" all data subsets with distinct access groups, while minimizing damage from key compromise. Key management is handled by the organization[1]. We also develop mechanisms that we need to manage keys when users or objects are dynamically added to or removed from the application or service.

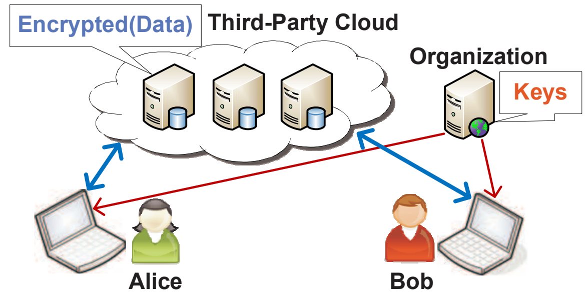

Figure 1: A depiction of our approach. The cloud stores encrypted data, and the organization stores decryption keys. The clients fetch the two and decrypt the data locally to obtain the application's service.

Secure and transparent user data access. Client (edge) devices, *e.g.* browsers, are given decryption keys by the organization to provide users with transparent data access. Of course, these devices (and users) must protect these keys from compromise. For example, an untrusted (or compromised) cloud can serve customized attack code to obtain encryption keys and decrypted data. To ward off these attacks, we propose a client-side component (which runs in the users' browsers) that allows users to access cloud services transparently, while preventing key compromise (even from a malicious cloud). Our solution works by leveraging already available features in modern web browsers such as same-origin policies and support for HTML5 *postMessage* calls. As a result, our solution works without any browser modifications, and can be easily deployed today.

Prototype and evaluation. We implemented our techniques as part of Silverline, a prototype of software tools designed to simplify the process of securely transitioning applications into the cloud. Our prototype takes as input an application and its data (stored in a database). First, it automatically identifies data that is functionally encryptable. Then, it partitions this data into subsets that are accessible to different sets of users (groups). We assign each group a different key, and all users obtain a key for each group that they belong to. This allows the application to be run on the cloud, while all data not used for computation is encrypted. Since popular EC2 applications like ShareThis are all proprietary, we apply our system to several popular open-source applications, and show that our system can partition data and assign keys to maximize data protection with a minimal number of keys. In addition, we find that a large majority of data can be encrypted on each of our tested applications.

In summary, the main contributions of this paper are:

- We introduce a novel approach to provide data confidentiality on the cloud while maintaining the functionality of cloud

[1] In this paper, we use "organization" to refer to the entity that wants to securely deploy its application and data on the cloud.

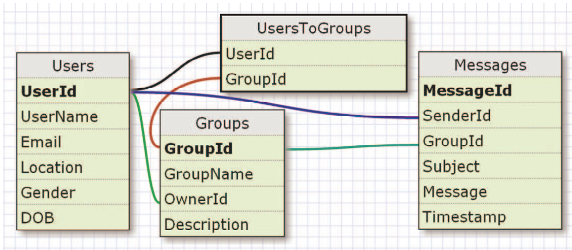

Figure 2: An example message board application's DB schema.

applications. Our approach works by automatically identifying subsets of an application's data that are not directly used in computation, and exposing them to the cloud only in encrypted form.

- We present a technique to partition encrypted data into parts that are accessed by different sets of users (groups). Intelligent key assignment limits the damage possible from a given key compromise, and strikes a good tradeoff between robustness and key management complexity.

- We present a technique that enables clients to store and use their keys safely while preventing cloud-based service from stealing the keys. Our solution works today on unmodified web browsers.

- We describe Silverline, a prototype toolset that implements our ideas, and discuss the results of applying Silverline to three real-world applications.

2. OVERVIEW OF SILVERLINE

Our overarching goal is to improve the confidentiality of application data stored on the cloud. We assume that the third-party computing cloud provides service availability according to service level agreements, but is otherwise untrusted. More specifically, we assume cloud servers may be compromised or may maliciously collude with attackers to compromise data confidentiality.

Our solution to improving data confidentiality on the cloud calls for end-to-end encryption of data by its owner (the organization) and its consumers (the users). In this paper, we concern ourselves with the data persistently stored in the databases. Our techniques apply to both traditional relational databases on the cloud, and to databases specifically designed for the cloud [3, 34, 14]. Access to encrypted data is granted through selective distribution of encryption keys, but only to users that have legitimate access to the data. We use *symmetric keys* to encrypt the data – symmetric keys are highly efficient and provide confidentiality with low computational overhead.

2.1 An Illustrative Application

We illustrate our approach using an online message board application, where users use topic-based forums for exchanging messages and discussions. We show a sample database schema for this application in Figure 2, and we will use this example throughout the paper to discuss our approach. The schema consists of a `Users` table to store user profile information such as name, userid or email, a `Groups` table to store information about discussion groups, a `UsersToGroups` table that maps users to the groups they are members of, and a `Messages` table that contains individual messages sent to the groups.

Today, an organization would deploy the above message board on the cloud by directly running it on the cloud infrastructure. Data would be stored in plaintext in a database, and queries from the users would be executed directly on this database. In this simple approach, user data confidentiality can be compromised in several ways. The cloud operators have access to cloud hardware and the application data. A bug in the software managing the cloud may reveal user data to attackers. Finally, the multi-tenant nature of the cloud brings a unique challenge: a compromised application running in the cloud can "infer" data that belongs to the users of other applications running on the same hardware [40]. Recent survey papers [13, 35] describe these and other threats in more detail.

2.2 Proposed Approach

In Silverline, we improve data confidentiality by encrypting as much of the application data as possible on the cloud (without breaking the application's functionality). This enables organizations to use existing clouds and protect their data and the data of their users. The key ideas of our approach are shown in Figure 1.

Storing and querying data on the cloud. In Silverline, data in the database (running on the cloud) is encrypted, but keys are not revealed to the cloud. The keys are stored by the organization that "outsources" its application and user data to the cloud. To fetch data from the cloud, the user first contacts the organization to get the appropriate key(s), and then sends the query to the cloud to fetch the data. The input parameters to the query are also sent in encrypted form. The cloud executes the query using this encrypted input and then sends back the results, also in encrypted form. Then, the user's device decrypts the data and displays it.

For example, consider the query: `SELECT * FROM Users WHERE UserName = 'Bob'`. Here, Bob queries the cloud for his detailed profile information. In current systems, the username would be in plaintext. In Silverline, the username is encrypted, using a symmetric key that is known only to the organization and Bob. Bob obtained this key when he registered with the organization. Thus, the query uses $E(\texttt{Bob}, K_{Bob})$ as the input parameter for the field `UserName`. The query (the SQL code itself) does not need to be modified. The results returned from the cloud are also encrypted with the symmetric key K_{Bob}, which Bob decrypts upon receipt.

Similarly, if some data is to be known to a group of users, then all the users in the group share the same key. For example, all the members of the group `Literature` would obtain the key K_{Lit} when they join the group. A query to fetch the messages sent to this group (`SELECT * FROM Messages WHERE GroupId='Literature'`) would use the encrypted value $E(\texttt{Literature}, K_{Lit})$ as the input parameter for `GroupId`. If a member wants to post a message to the group, the message would be encrypted before sending it to the cloud, using the group's key. The cloud would then store the encrypted blob of text in the database (instead of the plaintext message itself). Once the keys are received, the clients cache them to reduce future key requests to the organization, and thus, reduce the load induced on the organization.

Unfortunately, our approach might not be able to encrypt all application data. For example, the message board might want to display the average age of the users in the system. To compute this, the application must access the date-of-birth (DOB) field in the database, calculate the age of each user by subtracting the DOB with today's date, and then perform a summation to calculate the average[2]. Since this involves computation, Silverline identifies this field as *not* functionally encryptable and leaves it in plain text.

[2]Such computations could be performed on encrypted data using homomorphic encryption schemes [24]. Unfortunately, they incur very high overhead, which we want to avoid.

Storing and managing keys in the organization. In our approach, the role of the organization is to store all keys securely, and to provide users with only the keys that they should have access to. We take a fine-grained encryption approach to provide strong confidentiality guarantees.

A database consists of tables; tables consist of rows; and rows consist of cells. In Silverline, *different parts of a single table (even individual cells) may be encrypted with different keys*. For example, in the `Users` table above, consider a situation where all users can see all users' UserId and UserName in the system. However, only the user herself should see her email address, DOB, location, and gender. In this scenario, the ideal key assignment would be to encrypt all UserId and UserName cells in the table with one symmetric key, and give that key to all the users. But the cells corresponding to email, location, and gender of a user, say Bob, must be encrypted with a key (K_{Bob}) that is accessible only to Bob.

The organization is responsible for securing the creation of user accounts. For instance, a university deploying a message board for its employees is in charge of ensuring that each application account is actually owned by a different employee. This is important to prevent the cloud from gaining access to all keys by creating many users in the application's database to perform a Sybil attack [20].

The organization is also responsible for using Silverline to determine the key assignment, store these keys, and to provide access to keys to users, as they need them. Of course, all keys must be stored by the organization in a secure fashion. Since symmetric keys are small in size, and they can be cached on users' machines, the load on the organization is quite low, and so is the hardware cost. Effectively, the organizations can use external clouds, preserve the confidentiality of their data, and only incur a small cost in in-house hardware.

Data access on user devices. In our model, users store and retrieve (encrypted) data on the cloud, and they obtain their keys from the organization. Data is encrypted and decrypted locally. Therefore, protecting decrypted data and user keys is critical. Desktop applications can protect keys locally using standard techniques. For example, by storing and isolating the keys on disk with permissions given only to the user that represents the organization. However, we need an approach to provide similar isolation properties in web applications, where data, code and keys are combined in the same browser. To accomplish this, Silverline provides a solution that works without browser modifications. To leverage this, applications on user devices must request keys on behalf of the user, decrypt data from the cloud before displaying it to the user, and encrypt any user data before sending it to the cloud.

Application modifications and responsibilities. The goal of our work is to enable organizations to easily migrate existing Internet applications to a more secure model, where the majority of application data is protected from vulnerabilities in the cloud using end-to-end encryption.

To leverage this model, the organization's developers need to make three minor changes to the application before deploying it on the cloud. First, the developers need to add routines to encrypt data before uploading it to the cloud and routines to decrypt data before the clients can consume it. But there is *no need* to make any changes to the application logic. Second, the developers need to make minor changes to the database schema. The Silverline tools inform developers which fields can be encrypted on the cloud without affecting the application. Then, these encrypted fields should be modified to an appropriate type in the database, *e.g.* an `int` now becomes a `blob` of text (to store the encrypted integer value). These database schema changes can be completely automated. Third, the application needs to be run in a modified runtime on the cloud, which implements our techniques, rather than in the regular runtime. This modified runtime performs simple, light-weight tracing of data between the DB and the application.

Finally, after running this application on the cloud, the organization needs to perform key management, while the client devices perform data encryption and decryption.

Outlook. The key questions to answer when implementing our approach are the following: 1) Which portion of the data can be encrypted without breaking the application's functionality, 2) which keys are used to encrypt what portions of the data and how are they managed, and 3) how is encrypted data managed at the end users' devices. The answers to these questions are discussed in more detail in Section 3.

2.3 Confidentiality vs. Key Management

Before discussing the design of our system, we use this section to introduce and define some terminology: A user has access to a set of database cells, and hence, is given a set of keys that decrypt these cells. We describe the tradeoffs involved in assigning these keys to the cells starting with some basic definitions.

Our main goal is to maintain the confidentiality of the database on the cloud. This is achieved as long as the confidentiality of each cell is protected. A cell's confidentiality is defined as:

DEFINITION 1. *The* confidentiality *of a cell is maintained when no user that does not have access to the cell is able to decrypt it.*

We use the notion of the *scope of a key* to quantify the confidentiality properties in Silverline.

DEFINITION 2. *The scope of a key is the number of cells in the database that the key can decrypt.*

A user may receive multiple keys to decrypt all her cells. Then her scope is the sum of all her keys.

DEFINITION 3. *The scope of a user is the union of the scopes of all her keys.*

To reduce the management overhead on the organization and the users, the number of keys given to each user should be minimized. The obvious solution is to give no key to any user. However, this is not valid because it does not provide any functionality to the user. Of course, the application's *functionality must be preserved* after applying our mechanisms. That is, there is a tradeoff where the organization aims to distribute as few keys as possible, without denying any user access to data that this user is entitled to.

DEFINITION 4. *A user is said to have minimal keys, when reducing or increasing her keys any further leads either to breaking the application's functionality or to a loss of data (cell) confidentiality.*

The end points in the spectrum of choices to tradeoff between confidentiality and key management overhead do not meet our requirements. A key with absolute scope on the entire database violates confidentiality (as a user with that key can decrypt any cell in the DB). On the other hand, a key per cell (with a scope of one) leads to high key management overhead. For a given database, the best tradeoff is the one where each user has minimal keys according to Definition 4. If each user has minimal key assignment, then the key assignment for the entire database is said to be *optimal*. Silverline aims to achieve this *optimal key assignment*.

Finally, the cloud's scope must be zero. If the cloud colludes with a small set of users, then its scope is the union of the scope of all users it colludes with. As long as the organization secures the account creation process, the cloud cannot gain access to the entire database by performing a large-scale Sybil attack.

2.4 Integrity of the Data on the Cloud

In this paper, we focus on data confidentiality but set aside issues of verifying the integrity of data and computation on the cloud. This is because clients can use several existing techniques to verify that the cloud is not tampering with their data: First, clients can add an HMAC (hash-based message authentication code) [10] to the encrypted blob that is stored on the cloud. This can later be used by the receivers to verify the blob's integrity.

Second, the clients can use proof of storage [7, 29] techniques to verify that the cloud is not performing denial-of-service attacks and that the cloud is actually keeping all the data the clients stored. Even dynamic data [22] can be verified with this approach. These techniques work with very low computational overhead, and by transferring little data between the clients and the server.

Finally, techniques to protect against consistency attacks by untrusted cloud servers [31] can further help the clients to verify the consistency of the data obtained from the cloud – in particular, that the data received is the most up-to-date version, and that it is the one seen by other clients concurrently querying the cloud. These techniques need to exchange small amount of information, such as the root of a Merkle hash tree [32], with the users, which can be done in our model via the organization.

3. SYSTEM DESIGN

Silverline includes three techniques to help automate the transition to a more confidential application model. 1) *Encrypted data tracking* identifies functionally encryptable data. 2) *Database labeling and key assignment* partitions functionally encryptable data into different groups and assigns encryption keys. This also includes mechanisms to handle dynamic database updates appropriately. Finally, 3) *client-side key management* protects keys from compromised clouds. We describe these techniques in detail next.

3.1 Encrypted Data Tracking

Silverline uses a combination of information tagging and dynamic analysis to determine the types of data (*i.e.* the database fields) that are functionally encryptable. We apply our techniques by modifying the application runtime environment (for our evaluation, the PHP interpreter) to tag information associated with different database fields, and propagate them throughout the application logic. By training Silverline with a representative set of application queries, we expose the computational requirements of the application and determine whether each database field is functionally encryptable or not. We show a simple example in Figure 3, and describe details of our approach below.

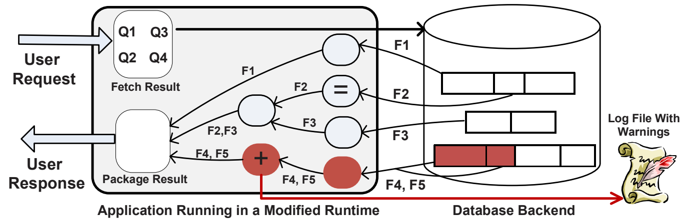

Figure 3: Encrypted data tracking: We train Silverline with a set of user inputs to the application that generate queries to the database back-end. The database responds with data, and the modified application runtime tags each data with a unique field number. The application runtime then propagates tags through computational dependencies, and logs warnings whenever a tagged piece of data is involved in a computation.

Dynamic program analysis. To find functionally encryptable data, one can perform static or dynamic program analysis. In both cases, the goal is to find database fields that are used by the application in computations, such as string operators, numerical operators, and comparators. To this end, one needs to track the use of results from the database and analyze their usage.

In this paper, we use a *dynamic* approach, based on a set of training queries that exercise the application. Given a set of training queries that are representative of application-to-database queries, we modify the interface between the database and the application runtime to automatically extract meta-information as data is returned from the database. Note that these modifications are application-independent and only need to be performed once for a particular programming environment (such as PHP, Python, or Java). We use this to build a table that maps the signatures of specific queries to fields accessed in the DB.

Data tagging and propagation. At a high level, Silverline tags all data entering the application from the database. It then tracks this data while it is used by the application, until a data object is involved in a computation or returned to the user without accessing its values. Data is sent from the database to the application in response to application queries. As each piece of data is retrieved from the database, it is tagged with a *field number* that corresponds to the field read. Field numbers are positive integers that uniquely identify a field in the DB.

As operations are performed on data, the modified application runtime or interpreter propagates the tags as follows. An assignment (data move) operation propagates the union of all tags of the right-hand side (RHS) operand to the left-hand side (LHS) operand. Any previous tags for the LHS are overwritten. For arithmetic, string, logical, or comparison operations, tags are propagated in the same way. However, in addition, if any of the operands on the RHS are tagged, then a warning event is generated for each tagged operand. This event includes field numbers of all tagged operands and their source code location.

After all queries in the training set have been executed, Silverline collects the logs containing all warnings generated in the application. We aggregate all warnings to produce a unique list of field numbers that tagged non-encryptable data. Using the previously-produced table (which maps field numbers to field details in the DB), we produce a list of all database fields whose values must be exposed in plaintext for the application to function properly. These fields are not functionally encryptable. All other fields are.

Modifying application runtime. We demonstrate our techniques on PHP applications, by modifying the PHP interpreter and the PHP-MySQL interface to support data tagging and propagation. We store the tags by extending the *_zval_struct* data structure that is at the base of all data types in the PHP interpreter. This ensures that tags propagate correctly for all data types and persist as long as an object remains.

An alternative approach. A static analysis approach is an alternative to our dynamic approach. It requires additional developer effort, which we aim to avoid. In particular, the developers need to annotate the queries in the source code with the fields accessed. The static approach might be more appropriate when a training set is not easily available. The tag propagation policies, however, remain the same in both approaches.

3.2 Database Labeling and Key Assignment

We now explain how Silverline addresses the challenge of assigning encryption keys to sets of data objects with the aim of producing a minimal key assignment for each user. To do so, we need to automatically determine the appropriate scope for different keys.

Again, we solve the problem by relying on a (relatively complete) training set of application requests. We assume that we have access to a snapshot of the application database, either taken from a running instance of the application, or produced by a sequence of recorded or synthetic user requests. We use the training set and the snapshot to generate a workload of database queries, allowing us to infer user access patterns and to perform optimal key assignments.

3.2.1 Labeling Algorithm

Given a sufficiently detailed set of requests, we can identify all database cells accessible to each user. By modifying the interface between the application runtime and the database, we can use a "database labeling" technique to capture and store these patterns. These labels are then used to produce a minimal key assignment. Figure 4 depicts labeling with an example.

In Silverline, the modified application runtime accesses application userIDs, and associates all queries to the database with the ID of the user whose request generated that query. This allows Silverline to assign to each cell in the database a *label*. A label is a set of all userIDs (users) who have access to that cell. For a cell c_i, its label can be written as $L_{c_i} = \{o_1, o_2, ..., o_j\}$, where o_j is the ID of a user that can access c. By definition, a user who runs a query has access to all cells returned as the result of that query. Therefore, we can build up a label for each cell in the database by running our training set of application requests. As each user runs a query that accesses a cell, her userID is appended to the cell's label if it is not already there. For example, if the query "`SELECT UserId FROM Users where Gender=0`" is executed by two users Bob and $Admin$, Silverline will label the `UserId` cells of all male users `(Gender=0)` in the table with label $\{o_{Bob}, o_{Admin}\}$.

Our approach uses a training set of either logged or synthetic user inputs (SELECT statements) to drive the database cell labeling process. For extremely large databases with complex schemas, it can be difficult for a training set to cover the bulk of the user-cell combinations possible in the application. In this case, we propose to augment an existing training set with additional synthetic requests using an approach similar to protocol input fuzzing [49], dynamic input generation for testing Web applications [47, 39] and dynamic input generation for high-coverage tests in database applications [21, 45]. For example, we can add queries to the query above with `Gender` as input parameter for all values of `Gender`, *e.g.* $\{0, 1\}$. For fields with a large number of potential values, *e.g.* a long type, we can use sampling guided by the application developers. To provide comprehensive coverage, we can continue updating cell labels until the query has been executed for all (or significant sample) of parameter values and user accounts.

Of course, even after using the aforementioned techniques, it is possible that our training data is incomplete. In this case, users are not provided keys to cells that they have access to. While this does not interfere with the confidentiality of data, it might deny legitimate users access. We handle omissions due to incomplete training in the same way as dynamic updates to the database (in both cases, some new information is added or discovered). The mechanism to handle this is described in Section 3.3.

Handling access hierarchies. Finally, most applications control data access using different hierarchies of users, *e.g.* the `admin` user versus regular users. Silverline mechanisms support this naturally because they infer a user's access privileges based on actual queries, rather than user names. For example, regular users can run the query `SELECT * FROM users WHERE UserId='xxx'` for their own userID, `admin` can run the query `SELECT * FROM users` to get data on all users. When these queries run during the training phase, Silverline naturally adds `admin` to the labels of all the cells in the users table. This easily extends to a complex hierarchy of users with different access privileges.

3.2.2 Key Assignment

Once the labeling step has finished, all cells will have labels that represent users who can access them. Our key assignment process uses this information to assign keys to groups of database cells that have common access patterns. Keys are then distributed to users based on their accessibility to groups of cells. The goal is to produce a minimal number of keys in the application while guaranteeing that each user can 1) decrypt all the cells she owns, but 2) cannot decrypt any cell that she does not own.

The key assignment is a simple process. We want an assignment that satisfies the constraint that each user's keys provide her with access to all cells she has access to (based on our training data), but no more. We also want to use a minimal number of total keys. We compute the initial key assignment by examining all cell labels in the entire database. We group all cells together that share the same label, and assign these cells a single, unique key. This divides all cells into a number of groups, each defined by a common label and a common key. Cells that share a common label are accessed by the same set of users, and thus, share the same encryption key.

There is an additional constraint to consider. Cells in columns that queries use to perform *join* on tables need to be either unencrypted, or encrypted using a single key. This is necessary to allow users to join tables without decrypting the involved table columns. This means that giving a user access to a single cell in the column is the same as giving her access to all cells in the column. We believe keeping these join columns unencrypted is generally reasonable, since joins are mostly performed on columns representing IDs of entities, and would not expose real valuable data.

Once assignment finishes, we create an encryption key for each cell group, encrypt the cells, and then distribute the key to all users identified in the group label. This ensures that each user has all the keys necessary to access all cells she should have access to.

Key assignment properties. As defined in Section 2.3, optimal key assignment for a database is the one that assigns the minimal number of keys to each user, such that the keys for this user 1) decrypt all her cells, and 2) do not decrypt any cell that she does not have access to. For a given database, our key assignment achieves these optimality and confidentiality properties. This follows from the three key steps that we perform in our assignment algorithm: 1) cells with same labels are assigned the same key, 2) cells with different labels are assigned different keys, and 3) a key is given to a user only when this user (her ID) is included in the corresponding label.

Key minimality: From the key assignment algorithm, it follows that the total number of keys assigned to encrypt the entire database is equal to the number of unique labels in the database. This is indeed the optimal number of keys: if there is a key assignment that uses fewer keys than the total number of unique labels, this assignment can only occur if *two different labels* are given the same key, which our algorithm never does. Hence, our algorithm achieves key minimality.

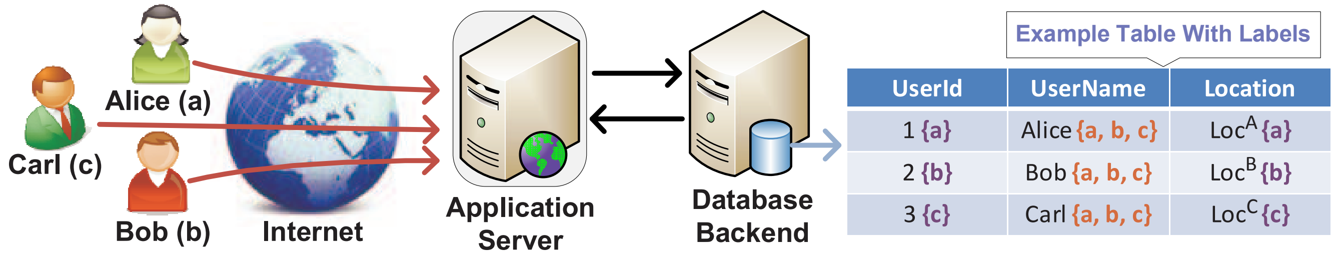

Figure 4: Database labeling in action. The application executes in a modified runtime that implements database labeling. The results produced after performing database labeling on the `Users` table for two queries: `SELECT UserName from Users` and `SELECT Location from Users WHERE UserId='id'` are shown. The first query returns all users' names, while the second returns only the querying user's location.

Cell confidentiality: A cell's confidentiality is violated only if the key to decrypt this cell is given to a user that does not belong to the label of the cell. However, our key assignment algorithm is based on the labels acquired during the training phase. Since a user without access to a cell is not included as a part of this cell's label, Silverline preserves confidentiality of all cells in the database.

3.3 Incompleteness and Database Dynamics

So far, we have described our mechanisms assuming that data tracking identifies all encryptable data, and that the database is static. However, neither of these hold in practice. Data tracking might miss computations on certain fields and, as a result, incorrectly tag these fields as functionally encryptable. Similarly, databases change due to a number of reasons. For example, new users join the system and are given access to existing data, and existing users leave the system. In addition, our training set of queries may not trigger all code paths in the application, thus, possibly omitting some users from labels of database cells they should have access to.

A core part of our approach is to accept that the results of the initial training process can be incomplete or outdated. We introduce an *online monitoring component* to deal with these two problems.

Online monitoring component on the cloud. This component runs inside the modified runtime deployed on the cloud. It has two purposes: First, the monitoring component performs a light-weight data tracking to identify any encrypted data used in computations, and, if so, alerts the organization. The organization can then use the key to decrypt the data and properly turn the field on the cloud into plaintext. The data tracking is similar to the approach discussed in Section 3.1 (and hence, we omit the details here).

The second purpose of the monitoring component is to determine when there are changes to the database cell labels that impact key assignment. For example, a query is executed where a user accesses cells for which she does not have the proper keys yet. Another example is a user that leaves a group (and hence, her access needs to be reduced). If such a change to a label is detected, the organization is notified. The organization then updates the key assignment based on the label changes. The monitoring component detects label changes by maintaining a "shadow" copy of the database to store the labeling results. That is, the label for a cell in the original database is stored in the corresponding cell of the shadow database (but the shadow does not store any other, actual data). Due to the fact that the number of unique labels is *significantly less* than the number of cells, the size of the shadow database can be optimized to be much less than the size of the original database (e.g., by indexing the shadow cell content into another table). Using this shadow database, the monitoring component just needs to generate events whenever a label in the shadow database changes after a query is run (without knowing the query details). When the organization receives a notification, it updates the key assignment and takes appropriate actions, as described next.

Changes due to database dynamics. The four possible events due to database dynamics that impact key assignment are listed in Table 1. Adding a user to a label and deleting a user from a label are the two fundamental events. These occur when a user accesses her data for the first time, or when a user leaves (or is removed from) the system. These two events, in turn, lead to two more events – merging of two labels and splitting of a label into two. A merge happens when the labels of two sets of cells become equal, and a split happens when only a subset of an original set of cells becomes accessible to a new user.

In Table 1, we present the ideal actions required to properly deal with the changes while maintaining key minimality. Adding a user to a label, for example, means that this user should get access to the label's key. Fortunately, providing a key to a user is a low overhead event. Removing a user from a label requires that the data corresponding to this changed label be re-keyed, *i.e.* decrypt the data using the old key and re-encrypt using a new key. This is necessary to prevent the revoked user from accessing future updates to data under this label. Data re-keying is undesirable, because it exposes data as plaintext, and thus, must be performed on the organization's own in-house computing resources rather than on the cloud. As a result, such an event incurs a non-trivial overhead.

Fortunately, this problem of key revocation for old members of the group is well addressed in the literature, originally in the context of content distribution from untrusted servers [23, 8]. Provably secure lazy *key regression* solutions have be proposed in the past. We reuse these techniques in Silverline to handle high-overhead events. More precisely, we handle a delete by assigning a new key to the label from which a user was removed. This new key is given to all the current members of the label, and is used to encrypt all data subsequently generated or updated under this label. However, this regressed key is special in the sense that it can be used by the current members to derive all previous keys used to encrypt content in this group [23, 8] and hence decrypt all content under this label with only this single key. Intuitively, the keys generated by this key regression approach are linked in a manner similar to a reverse hash chain, where given the current key, all previous keys can be derived, but not the other way around. This key regression approach has been shown to scale well in real applications with highly dynamic group membership with very low overhead [23].

We handle label merges by revealing the keys of both the merged labels to the members of both the merged label groups. We handle label split events by assigning a new regressed key to each of the

Change event	Add a user	Delete a user	Merge two labels	Split a label into two
Ideal action	Grant access to key (low)	Re-key data (high)	Re-key data (high)	Re-key data (high)
Proposed action	Grant access to key (low)	Assign a new key (low)	Reveal both keys (low)	Assign new keys (low)

Table 1: Possible label changes due to dynamism in database content, ideal actions required to deal with these changes and their corresponding overhead (in bracket), and our proposed approach to address dynamism and its overhead.

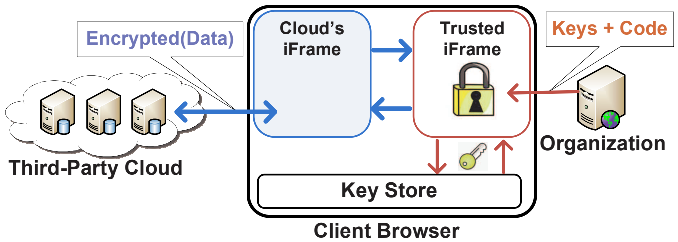

Figure 5: Our design of safe data processing in the user's browser.

new labels generated from the split. The new user in a split label will be given access to the key to the old label before the split to ensure she has access to the data under that label. Using this lazy approach to reduce the overhead from dynamic group membership slightly relaxes key minimality. For instance, there are two keys (instead of one) associated with a label after a split or merge event. However, the application can reach the minimal assignment state by periodically re-keying the data.

Note that we consider the events generated by the monitoring component on the cloud to be untrusted. To prevent the cloud from giving (possibly malicious) users access to data arbitrarily, the organization checks the label changes to ensure that the users' scopes are constrained and that no single user gets access to a large amount of data. Only then the organization reacts to the label change events and updates key assignments. Thus, the organization can keep tight control over the scope of the users to maintain confidentiality.

Delivering new keys to users. A final question is how to seamlessly provide users with access to new keys. Of course, this should not cause any service disruption. When an authorized user accesses data, but she is missing the necessary decryption keys, this user would not be able to process or display the data. In this case, the user can simply query the organization for the missing keys (using key indices described in Section 3.4) and then continue to use the application. The new keys can be obtained with one RTT latency and cached for future. Thus, the user only incurs a one-time delay of a single round trip. Similarly, when a user writes (encrypted) data to the cloud, the cloud can warn the user when she is using older keys. Then, the user can obtain the new keys from the organization and upload newly encrypted data.

3.4 Safe Key Management on User Devices

To provide users transparent access to their data, the organization must distribute decryption keys to users' edge devices. As a result, Silverline must ensure that a compromised cloud cannot steal keys or decrypted data from user devices. In particular, when the client accesses the application on the cloud and downloads encrypted data with a web browser, a malicious cloud could inject client-side code (a piece of JavaScript, for instance) into the output. This client-side code is then executed on the user's device, which stores the decryption keys. Clearly, we need to ensure that this client-side code cannot access or leak keys, and that the decryption can be done in a secure fashion before the data is presented to the user.

Secure data access on user devices. The key insight behind our approach is to isolate (prevent) the untrusted code from the cloud from accessing sensitive data (such as keys or decrypted data) on user devices. Only the code from the organization is allowed to access such data. We accomplish this by leveraging functionality that is already present in modern Web browsers. In particular, we make use of the Same Origin Policy (SOP) and HTML5. As a result, our solution works in current browsers without modification.

We leverage iFrames to isolate and restrict access to sensitive data in the Web browser. The idea is to use two iFrames in designing web applications hosted on the cloud. One frame belongs to the cloud, and one belongs to the organization. The keys are stored in the user's web browser (as cookies, or on disk with HTML5) under the same *origin* (the source site details) as the organization. As a result, the browser's SOPs prevent the untrusted cloud frame from accessing keys that belong to the organization, due to different origins. Keys are only accessible by the organization's frame, protecting them from a potentially compromised cloud.

Once keys are isolated, the next step is to isolate the data decryption process, so that unencrypted data does not leak to the cloud. In our solution, the untrusted cloud's frame downloads the encrypted data from the cloud, then sends this (encrypted) data to the trusted organization's frame via a HTML5 `postMessage` call. The organization's frame receives the encrypted data, decrypts it locally, and renders or processes the data based on user requirements. Any data sent back to the cloud is first encrypted with appropriate keys inside the organization's frame, and then sent back to the cloud's frame, which posts the message to the cloud. Because the frames cannot directly access each other's data inside the browser, decrypted data is never accessed by the cloud's frame. Our solution is depicted in Figure 5.

Trusting the browser-side code. The final detail is to determine how the code is sent safely to user devices. In our implementation, the organization hosts the entire code that runs in the trusted frame and sends it to the user, which is then cached in her browser. Then, the cloud's frame only needs to download encrypted data from the cloud, and then upload encrypted data generated by the user to the cloud. Since the code is generally small is size, and is cached on the client, the load incurred on the organization in hosting the code is also small. While we chose this approach for its simplicity, an alternative approach based on code verification, similar to BEEP [28], is also possible.

We implemented a prototype application to validate this design, as shown in Figure 5. We hosted data on one server (acting as the cloud), code on another server (acting as the organization) and ran the application on a separate user machine. Our prototype runs successfully *without any browser modification* on Internet Explorer 8, Firefox 3.5.8, Google Chrome 5.0.3, and Safari 4.0.5.

Key indexing to guide data access. To enable user devices to decrypt data received from the cloud, each piece of encrypted data must have an accompanying piece of metadata that indicates the key necessary for decryption. Thus, we assign indices (random numbers) to each key generated at the organization during the database labeling phase. The index of a key is essentially its name, and it is distributed with the key and all data encrypted with that key. The cloud sending encrypted data to the user also sends all necessary key indices, thus allowing the trusted user frame to use the proper key for decryption.

4. LIMITATIONS OF SILVERLINE

Not all data on the cloud is encrypted. While we would like to encrypt the entire database's content on the cloud, in this work, we focus on encrypting functionally encryptable data. We recognize this limitation and are designing techniques to cover more data as part of our ongoing work.

Cloud can learn some metadata. To be able to run queries on encrypted data, we have to ensure that a given value that is encrypted with a given key always yields the same ciphertext; that is, there is no randomization (or salting) used. For instance, while using key K, Alice always encrypts her user name $alice$ to $E_K(alice)$. In our system, this is necessary for the cloud to run queries and to select all data that matches $E_K(alice)$. Randomizing the ciphertext with each encryption would prevent the cloud from running such queries.

The downside to this is that the cloud can learn some metadata about the data stored on it. For example, if two users $alice$ and bob send each other messages, the cloud would know the number of messages sent between two users $E(alice)$ and $E(bob)$. While this alone is not sufficient to break either user's privacy, if the cloud were to combine this with some outside data, it might be able to determine the number of messages exchanged between $alice$ and bob.

Executing inequality comparisons on encrypted cells. Once the cells are encrypted, queries such as `SELECT * FROM Messages WHERE MessageId > 10` no longer work, as inequality comparisons over encrypted data fail. We leave resolving such issues to future work.

Attacks on community data. Data encrypted with a single key (to protect from the cloud) that is shared with all the registered users in an application (called community data hereafter) are vulnerable to a variety of attacks by the cloud. The cloud can mount a *known-plaintext attack* or a *distribution-based attack*. Consider a community field with a fixed set of values, such as `Gender`. In a known-plaintext attack, the cloud can join the system as two users (or collude with two users), one with each gender. Based on the encrypted value learned, the cloud now knows the actual gender of all other users in the database. In the distribution attack, the cloud can use some external information to learn the gender of all users in the system. For example, if the cloud knows that there are more male Star Trek fans, then it can easily guess the gender of all the users in the Star Trek message board on the cloud using the distribution of encrypted values. Note, however, that such attacks work *only against community data*. Data encrypted with user-specific keys is still secure.

5. EVALUATION

We now evaluate the efficacy of Silverline techniques on existing, real-world applications. Our evaluation is geared towards answering *two key questions*: 1) How much of the data in today's applications can be encrypted without breaking any functionality? and 2) Does our labeling identify all the different types of data sharing between users and assign the right keys to the right users?

5.1 Setup and Implementation

Evaluation setup. We applied our techniques to three different, real-world PHP applications hosted on *sourceforge.net*. We chose these applications because they represent a good mix of features commonly found in real applications, which lead to several interesting data sharing characteristics. The details of the applications used in our evaluation are presented in Table 2. Each of these applications has tens of thousands of lines of code, and all contain a significant number of database queries.

Implementing encrypted data tracking. Our modification to the PHP interpreter and the PHP-MySQL interface were based on the code for phptaint [46]. We modified this code to incorporate our tag propagation policies as described in Section 3.1. Our implementation logs a warning every time a tagged data item is used in a computation. We ran each application in our modified interpreter, exercising different paths of the program via "normal" user interactions. Then, we analyzed the contents of the log to identify those cells that cannot be encrypted. Note that we do not consider using data in display (output) functions, such as `echo` and `print`, as computation. Data in such functions can be sent encrypted to the user, where it can be transparently decrypted and displayed.

Implementing database labeling and key assignments. All the applications that we used for our evaluation use MySQL as their back-end database. We implemented labeling in a MySQL-proxy between the database and the PHP runtime. For each of these applications, we used the following setup. We 1) create a database with the exact same schema used in the application, 2) insert sample data into the database to create a training database for labeling, 3) identify all SELECT queries in the application that read data from the database, 4) perform database labeling on SELECT queries in the applications, and finally 5) analyze the labels attached to the cells to verify the data classification and key assignment performed by our techniques.

5.2 Application Descriptions

AstroSpaces: A social networking service. AstroSpaces is a social networking application that provides the following features to users: 1) create user profiles, 2) add users to their friend list, 3) send private messages to friends, 4) create blog posts, 5) write comments to friends on their profiles, and 6) create content on their own profiles. These features are built on 7 database tables and a total of 51 SELECT queries.

UseBB: A full-featured message board. UseBB is a popular bulletin board service that provides many advanced features to users, including the ability to 1) create accounts, 2) create and moderate groups, 3) join groups, and 4) post new topic messages or reply to existing topics. UseBB administrators have access to advanced features such as banning users (by email or username or IP address), banning keywords and configuring replacement words, sending mass emails, editing/deleting users, and many other options to configure user forums. These features are implemented using 12 tables and a total of 114 SELECT queries.

Comendar: A community calendar. Comendar is a community calendar service that provides users with the ability to: 1) create user accounts, 2) create groups (for communities), 3) join com-

[3]http://sourceforge.net/projects/astrospaces/
[4]http://sourceforge.net/projects/usebb/
[5]http://sourceforge.net/projects/comendar/

Application	Purpose	Lang.	LOC	Queries	Total Downloads
AstroSpaces [3]	Social Networking	PHP	14790	51	8320
UseBB [4]	Complex Message Board	PHP	21264	114	75066
Comendar [5]	Community Calendar	PHP	23627	42	5123

Table 2: Details of the applications used in our evaluation. We only list the # of SELECT queries in the application in this table. We retrieved the total # of downloads of the applications from sourceforge as of November 19th 2010.

munities (or groups) of interest, 4) create new personal and community events, 5) view personal and community events, 6) setup reminders to be sent via email (for both personal and group events), and 7) set display and privacy preferences. This application provides the services of an online calendar service – but for both personal and community uses. There are a total of 13 tables in the database and 42 SELECT queries in the application.

5.3 Amount of Functionally-Encryptable Data

First, we evaluate the amount of functionally encryptable data in the applications. We consider all database fields that store user data (only excluding the auto-increment IDs used to identify entities in the tables) as sensitive. These ID fields are typically integers that do not reveal any information about a user. Hence, they can remain in plaintext. To understand the fraction of sensitive fields that can be encrypted, we use our modified PHP interpreter and track the usage of sensitive data. By analyzing the warnings produced by our tracking system, we could understand which fields were used in computations and why. Table 3 summarizes the results, which we discuss below.

AstroSpaces social networking service. Out of the 24 user data fields (those that did not store UserId, GroupId, or any other IDs), we find that only seven were used in computations, including: Username (to search based on partial names), read/unread status of messages (to display unread messages in bold), accepted/unaccepted status of friendship requests (to display friend request status in categories), theme and style chosen by the user (again, for display), activation status of the account (to decide if users are allowed to login or not) that users are required to set by confirming account creation, and finally, the user's email (to send emails, search by email for existing accounts during account creation, and send password reminders).

Interestingly, most of these fields store information not directly related to the user. On the other hand, personal data such as the user's first name, her last name, the messages exchanged between friends, the user's address, the phone number, blog posts, and wall posts are never interpreted or used in any computation, only read and sent to users. Thus, these fields are all functionally encryptable and protected by Silverline.

UseBB message board. As Table 3 shows, out of a total 81 user data fields in the UseBB database, only 14 fields are used in computations. These 14 user data fields are the following: The names of the users, title and content of their posts (to enable searching by keywords, and replace banned keywords), emails (to send emails and password reminders), the level of the user (guest, standard user, or admin; to decide what operations they can perform), activation status of user accounts (for login purposes), and the user's privacy and display preferences.

Nearly half of the functionality that requires interpretation of data is related to content formatting. This functionality can be easily moved to client-side scripting code, thus removing those computation dependencies and making the data fields they touch functionally encryptable. Several remaining fields store information

Application	# of Database Fields			
	Total	User Data	Encryptable	Non-Encryptable
AstroSpaces	37	24	17 (71%)	7 (29%)
UseBB	106	81	67 (83%)	14 (17%)
Comendar	105	57	41 (72%)	16 (28%)

Table 3: Encrypted data tracking results. We show a) the # of fields in total, b) the # of sensitive fields storing user data, and the # of sensitive fields that c) are functionally encryptable and d) are not functionally encryptable.

that is not related to personal user data (*e.g.* user's level, and activation status of the accounts). This leaves us with only the fields used for keyword search (user names, title, and content of the posts). They are personal, used in computation, and should preferably remain encrypted on the cloud. Fortunately, work on keyword search on encrypted data [43, 44] can help in encrypting these fields also.

Comendar community calendar. Comendar performs more computations than the two previous applications. Out of a total of 57 sensitive fields, 16 were used in computations. These are: a user's email, magic string (for password reminders and account activation), the account activation status, user's gender and level, group and event security settings (public or private), event titles and contents (for keyword search), start and end date for reminders, reminder and event repetition interval, and event attendance status (yes, no, or maybe).

Similar to the two previous applications, half of the computations (8 out of 16) were performed on fields that were used only to format the data displayed to the user. For example, user's gender is used to decide if "he" or "she" should be displayed. A majority of the fields that are involved in computations on the cloud, such as start and end date of reminders, reminder and event interval, etc. can likely remain in unencrypted form. Only the events' titles and descriptions, which are used in search operations, should preferably be stored in encrypted form.

Summary. For the three applications that we examined, we found that the *majority* of fields that store personal information *are never used in any computation*. These fields include address, phone number(s), messages exchanged between users, and other personal details. Many fields used in computation store information about users that are unlikely to be sensitive. Only a handful of fields stored sensitive information and were used in computation (mostly for keyword searches), which the organization could still encrypt with specialized encryption schemes [44]. In short, an organization can encrypt most sensitive fields with efficient symmetric keys and efficiently obtain confidentiality when running applications on today's clouds.

5.4 Evaluating the Key Inference Techniques

Now, we evaluate if our labeling and key assignment techniques correctly identify different groups of users that have access to different cells in the database, and if they assign appropriate, shared keys to each group.

AstroSpaces social network. This application involves a significant amount of pair-wise user interactions, as can be expected from a social network. More precisely, most queries were involved in creating the friendship graph and exchanging messages between friends.

There are basically three types of data in AstroSpaces: 1) data that is publicly visible to all users (Blogs, Username, UserId, profile content), 2) data that is viewed only by a pair of users, and 3) data that is viewed only by the owner (details about the user, such as gender, email, and last login time). We first create a database with 50 users, then make each user connect with a random number of randomly chosen friend users. After that, we make users interact with their friends by sending private messages and by writing comments on profiles. We make this interaction realistic by biasing the frequency of interactions towards a handful of "close" friends. Finally, users create blogs and embellish their profile pages.

Then, we run the queries in the application on this sample database, and analyze the labels acquired by the cells. A total of 51 labels, and hence, keys, are assigned to the `Users` table. Out of these, 50 user-specific keys are assigned to the 50 users (one key each) to encrypt all columns, with the exception of Username and UserId. All publicly accessible columns are encrypted with just one key, which is given to all users. The data in the `Private Messages` table is read only by the receiver of messages, and never read by the sender. Hence, Silverline reuses the user-specific keys assigned to the `Users` table to encrypt this table as well. In particular, a message sent to a user A is encrypted with the key of user A. The data in the `Friendship` table, on the other hand, is accessed by the users on both ends of friendship edges. As a result, the same label (key) is assigned to all cells accessed by a particular pair of users. In our database, there were 588 distinct pairs, and hence, 588 keys were created. Finally, the content in the rest of the tables is public. For this, the key associated with public data (known to all users) is reused to encrypt this content.

In summary, our labeling technique successfully identified the three different groups of data in this application, as well as the users that belong to these groups. Our system assigned a total 639 keys to protect our AstroSpaces database.

UseBB message board. There are four types of data in UseBB: data that is 1) visible to the entire world (public), 2) visible to all registered UseBB users (community), 3) visible to a single user, and 4) visible only to the admin. There is no data accessible to a specific subset (or group) of users in UseBB, and most of the data belongs to the first two types. Similar to other message boards, data generated by users in UseBB is organized in different categories. Each category has multiple forums. Each forum, in turn, has multiple topics on which users discuss by sending posts. Topics are akin to a new mail thread, and each post is akin to a response to this mail thread. In UseBB, all posts in all forums and categories are public. Even several details of the members that made the posts are public. However, information such as statistics about members' activities and the full list of members is community data. Some information, such as a user's preferences (email is public or not, theme, etc.) is accessible only to a particular user (and the admin). Finally, data such as the banned users, words, and IP addresses are accessible only to the admin.

We create a sample database with 50 users, five categories, five topics, and 20 forums. We then make random users send posts to different topics. Finally, we use Silverline to examine the SQL queries and perform key inference. Our system correctly classified the data into the four types mentioned above, and identified the fields that belonged to each type. The key assignment is simple, due to the lack of complex groupings of users. A total of 53 keys are assigned – 50 user-specific keys (one per user), one key for public data, one key for community data, and finally, one key for the admin data.

Comendar community calendar. There are four types of data in Comendar: 1) data visible to the entire world (public), 2) data visible to all registered users in the Comendar application (community), 3) data visible to all users in a group (group), and 4) data visible only to the user that created it (personal data).

Comendar is interesting because some queries were dynamically generated. More precisely, the application dynamically constructs selection conditions used to query tables. As a result, although the number of queries in the source code is 42, over several runs, we identified 49 different queries. Since our technique only depends on the name of the user running a query, Silverline handled these dynamic queries easily.

We run Silverline on a sample database with 50 users and 10 groups, and assign a random number of randomly chosen users to each group. Each user creates one event for each of the different access types (public, community, group, and personal). We then assign group events to randomly chosen groups. Users then create reminders for their own events and for community events. Finally, we run the application so that Silverline could analyze the `SELECT` queries.

Silverline correctly classified all four types of data. More precisely, our system assigned a total of 61 keys to these four types. 50 out of 61 keys were used to encrypt user-specific data (personal events, personal reminders, event attendance status, etc.). Since there were 10 groups, our technique was expected to assign 10 keys to protect the groups' data. Interestingly, however, only 9 keys were created. Closer examination revealed that one group contained only one user. As a result, our algorithm correctly re-used that user's personal key for this group's data. Moreover, one key was assigned to encrypt the community data, and finally, one key was assigned to encrypt the public data.

Summary. The evaluation shows that our labeling technique successfully identifies different types of sharing behaviors in production applications, and classifies the data into groups. The technique also identifies all users that have access to these groups. Putting the evaluation results together, we learn that many of today's applications can easily benefit from Silverline.

6. RELATED WORK

Encrypted databases. Encrypted databases [19, 26] offer database-as-a-service [26], where databases run on an untrusted third-party and operate on encrypted data. They aim to offload most of the query execution from clients to the third-party, by inserting additional columns in the encrypted database to provide hints for query execution. Our work differs significantly in the threat models we consider. Encrypted databases consider a single server and a single client (the organization hosting the DB), whereas we assume many clients (other than the organization) in our model. As a result, their approach of using a single key for encryption is not sufficient for our model, which supports mutually distrusting users.

Systems running on encrypted data. Persona [9] is a social network where the server never sees any data in plaintext. Persona uses attribute-based encryption to allow fine-grained sharing of encrypted information with friends. However, this approach requires applications to be rewritten to support encryption natively. In contrast, Silverline focuses on using automated tools to simplify the transition of legacy applications to a secure cloud platform.

Supporting security and privacy in clouds. Work on accountable clouds [27] proposed an approach for users of third-party clouds to verify that the cloud is operating "correctly" on their data. Similarly, a recent paper [42] aimed to build trusted clouds that protect user data against attacks from compromised cloud administrator accounts using TPMs. While these approaches are based on modifying the cloud infrastructure to enforce security and privacy policies, we aim to work on unmodified clouds.

Taint tracking for security and software debugging. Taint tracking has been used in a variety of contexts, such as detecting software vulnerabilities [38], malware analysis [51], debugging applications [18], and securing web applications [50]. More broadly, information flow control has been used in the development of programming languages [37, 36], secure operating systems [53] and applications [52] to prevent data from reaching untrusted entities. Our work differs from these projects in the way we use data tagging and information labeling. In particular, our focus lies in identifying functionally encryptable data.

7. CONCLUSIONS AND FUTURE WORK

Data confidentiality is one of the key concerns that prevent organizations from widely adopting third-party computing clouds. In this paper, we describe Silverline, a set of techniques that promote data confidentiality on the cloud using end-to-end data encryption. Encrypted data on the cloud prevents privacy leakage to compromised or malicious clouds, while users can easily access data by decrypting data locally with keys from a trusted organization. Using dynamic program analysis techniques, Silverline automatically identifies functionally encryptable application data, data that can be safely encrypted without negatively affecting application functionality. By modifying the application runtime, *e.g.* the PHP interpreter, we show how Silverline can determine an optimal assignment of encryption keys that minimizes key management overhead and impact of key compromise. We demonstrate the viability of our approach by applying our techniques to several production applications with a mix of commonly used features. Our experiences show that applications running on the cloud *can* protect their data from security breaches or compromises in the cloud.

While our work provides a significant first step towards full data confidentiality in the cloud, a number of challenges remain. We target two specific areas as topics of ongoing work.

Learning high-level intuitions for data classification. While our database labeling currently classifies the cells in the database that can be encrypted together, it does not tell the developers about the reasons why such a classification happened. An intuitive reasoning for such a classification is more helpful for the developers in later implementing encryption and decryption functionality in the applications. We believe applying associative rule mining [2] techniques can help us derive these intuitions.

Automatic partitioning of the applications. We are planning on extending Silverline to automatically partition applications and move sensitive data (and its computation) to client devices, similar to Swift [16]. Swift only supports partitioning of static data in applications, but we plan to extend it to partitioning database content using the labeling information dynamically learned by Silverline.

Acknowledgments

We thank the anonymous reviewers for their comments to help improve the paper. This material is based in part upon work supported by the National Science Foundation under grants IIS-0916307, IIS-847925, and CNS-0845559, and by the U.S. Army Research Laboratory and the U.S. Army Research Office under MURI grant No. W911NF-09-1-0553.

8. REFERENCES

[1] ABDALLA, M., ET AL. Searchable encryption revisited: Consistency properties, relation to anonymous IBE, and extensions. *Journal of Cryptology* (2008).

[2] AGRAWAL, R., IMIELIŃSKI, T., AND SWAMI, A. Mining association rules between sets of items in large databases. *Proc. of ACM SIGMOD* (1993).

[3] AMAZON. Amazon SimpleDB.

[4] AMAZON. Nimbus health. `http://aws.amazon.com/solutions/case-studies/nimbus-health/`.

[5] AMAZON. Sharethis. `http://aws.amazon.com/solutions/case-studies/sharethis/`.

[6] AMAZON. Thread: Does amazon ec2 meet pci compliance guidelines? `http://developer.amazonwebservices.com/connect/message.jspa?messageID=139547`.

[7] ATENIESE, G., ET AL. Provable data possession at untrusted stores. In *Proc. of ACM Conference on Computer and Communications Security (CCS)* (2007).

[8] BACKES, M., CACHIN, C., AND OPREA, A. Secure key-updating for lazy revocation. In *Proc. of ESORICS* (2006).

[9] BADEN, R., BENDER, A., SPRING, N., BHATTACHARJEE, B., AND STARIN, D. Persona: An online social network with user defined privacy. In *Proc. of SIGCOMM* (2009).

[10] BELLARE, M., CANETTI, R., AND KRAWCZYK, H. Keying hash functions for message authentication. In *Proc. of Annual International Conference on Cryptology (CRYPTO)* (1996).

[11] BONEH, D., DI CRESCENZO, G., OSTROVSKY, R., AND PERSIANO, G. Public key encryption with keyword search. In *Advances in Cryptology-Eurocrypt 2004* (2004).

[12] BONEH, D., AND WATERS, B. Conjunctive, subset, and range queries on encrypted data.

[13] CACHIN, C., KEIDAR, I., AND SHRAER, A. Trusting the Cloud. *ACM SIGACT News 40*, 2 (2009).

[14] CHANG, F., ET AL. Bigtable: A distributed storage system for structured data. In *Proc. of Operating Systems Design and Implementation (OSDI)* (2006).

[15] CHANG, Y., AND MITZENMACHER, M. Privacy preserving keyword searches on remote encrypted data. In *Applied Cryptography and Network Security* (2005), Springer.

[16] CHONG, S., LIU, J., MYERS, A. C., QI, X., VIKRAM, K., ZHENG, L., AND ZHENG, X. Secure web applications via automatic partitioning. In *Proc. of ACM Symposium on Operating Systems Principles (SOSP)* (Oct. 2007).

[17] CIRCLEID. Survey: Cloud computing 'no hype', but fear of security and control slowing adoption. `http://www.circleid.com/posts/20090226_cloud_computing_hype_security/`.

[18] CLAUSE, J., AND ORSO, A. Penumbra: automatically identifying failure-relevant inputs using dynamic tainting. In *Proc. of ISSTA* (2009).

[19] DAMIANI, E., VIMERCATI, S., JAJODIA, S., PARABOSCHI, S., AND SAMARATI, P. Balancing confidentiality and efficiency in untrusted relational DBMSs. In *Proc. of ACM Conference on Computer and Communications Security (CCS)* (2003).

[20] DOUCEUR, J. R. The Sybil attack. In *Proc. of IPTPS* (March 2002).

[21] EMMI, M., MAJUMDAR, R., AND SEN, K. Dynamic test input generation for database applications. In *Proc. of International Symposium on Software Testing and Analysis (ISSTA)* (2007).

[22] ERWAY, C., KUPCU, A., PAPAMANTHOU, C., AND TAMASSIA, R. Dynamic provable data possession. In *Proc. of ACM Conference on Computer and Communications Security (CCS)* (2009).

[23] FU, K., KAMARA, S., AND KOHNO, T. Key regression: Enabling efficient key distribution for secure distributed storage. In *Proc. of Network and Distributed System Security Symposium (NDSS)* (February 2006).

[24] GENTRY, C. *A fully homomorphic encryption scheme*. PhD thesis, Stanford, 2009. `http://crypto.stanford.edu/craig`.

[25] GOLLE, P., STADDON, J., AND WATERS, B. Secure conjunctive keyword search over encrypted data. In *Applied Cryptography and Network Security* (2004).

[26] HACIGUMUS, H., IYER, B., AND MEHROTRA, S. Providing database as a service. In *Proc. of IEEE International Conference on Data Engineering (ICDE)* (2002).

[27] HAEBERLEN, A. A case for the accountable cloud. In *Proc. of Workshop on Large Scale Distributed Systems and Middleware (LADIS)* (2009).

[28] JIM, T., SWAMY, N., AND HICKS, M. BEEP: Browser-enforced embedded policies. In *Proc. of WWW* (2007).

[29] JUELS, A., AND KALISKI JR, B. PORs: Proofs of retrievability for large files. In *Proc. of ACM Conference on Computer and Communications Security (CCS)* (2007).

[30] KATZ, J., SAHAI, A., AND WATERS, B. Predicate encryption supporting disjunctions, polynomial equations, and inner products. *Proc. of EUROCRYPT 2008*.

[31] LI, J., KROHN, M., MAZIÈRES, D., AND SHASHA, D. Secure untrusted data repository (SUNDR). In *Proc. of Operating Systems Design and Implementation (OSDI)* (2004).

[32] MERKLE, R. A digital signature based on a conventional encryption function. In *Proc. of Annual International Conference on Cryptology (CRYPTO)* (1987).

[33] MESSMER, E. Are security issues delaying adoption of cloud computing? `http://www.networkworld.com/news/2009/042709-burning-security-cloud-computing.html`.

[34] MICROSOFT. Microsoft SQL Azure.

[35] MOLNAR, D., AND SCHECHTER, S. Self Hosting vs. Cloud Hosting: Accounting for the security impact of hosting in the cloud. In *Proceedings of the Ninth Workshop on the Economics of Information Security* (2010).

[36] MYERS, A., AND LISKOV, B. A decentralized model for information flow control. In *Proc. of ACM Symposium on Operating Systems Principles (SOSP)* (1997), ACM.

[37] MYERS, A., ZHENG, L., ZDANCEWIC, S., CHONG, S., AND NYSTROM, N. Jif: Java information flow. *Software release. Located at `http://www.cs.cornell.edu/jif` 2005* (2001).

[38] NEWSOME, J., AND SONG, D. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. In *Proc. of Network and Distributed System Security Symposium (NDSS)* (2005).

[39] RICCA, F., AND TONELLA, P. Analysis and testing of web applications. In *Proc. of the International Conference on Software Engineering (ICSE)* (2001).

[40] RISTENPART, T., TROMER, E., SHACHAM, H., AND SAVAGE, S. Hey, You, Get Off of My Cloud: Exploring Information Leakage in Third-Party Compute Clouds. In *Proc. of ACM Conference on Computer and Communications Security (CCS)* (2009).

[41] ROITER, N. How to secure cloud computing. `http://searchsecurity.techtarget.com/generic/0,295582,sid14_gci1349550,00.html`.

[42] SANTOS, N., GUMMADI, K., AND RODRIGUES, R. Towards trusted cloud computing. *Proc. of HotCloud* (2009).

[43] SHI, E. *Evaluating Predicates over Encrypted Data*. PhD thesis, PhD Thesis, Carnegie Mellon University, 2008.

[44] SONG, D., WAGNER, D., AND PERRIG, A. Practical techniques for searches on encrypted data. In *Proc. of Security and Privacy* (2000).

[45] SUÁREZ-CABAL, M., AND TUYA, J. Using an SQL coverage measurement for testing database applications. *ACM SIGSOFT Software Engineering Notes 29*, 6 (2004), 253–262.

[46] VENEMA, W. Taint support for php. `http://wiki.php.net/rfc/taint`.

[47] WASSERMANN, G., ET AL. Dynamic test input generation for web applications. In *Proc. of International Symposium on Software Testing and Analysis (ISSTA)* (2008).

[48] WESTERVELT, R. Researchers say search, seizure protection may not apply to saas data. `http://searchsecurity.techtarget.com/news/article/0,289142,sid14_gci1363283,00.html`.

[49] WONDRACEK, G., COMPARETTI, P. M., KRUEGEL, C., AND KIRDA, E. Automatic network protocol analysis. In *Proc. of Network and Distributed System Security Symposium (NDSS)* (2008).

[50] XU, W., BHATKAR, E., AND SEKAR, R. Taint-enhanced policy enforcement: A practical approach to defeat a wide range of attacks. In *Proc. of USENIX Security Symposium* (2006).

[51] YIN, H., SONG, D., EGELE, M., KRUEGEL, C., AND KIRDA, E. Panorama: Capturing system-wide information flow for malware detection and analysis. In *Proc. of ACM Conference on Computer and Communications Security (CCS)* (2007).

[52] YIP, A., NARULA, N., KROHN, M., AND MORRIS, R. Privacy-preserving browser-side scripting with BFlow. In *Proc. of EuroSys* (2009).

[53] ZELDOVICH, N., BOYD-WICKIZER, S., KOHLER, E., AND MAZIERES, D. Making information flow explicit in HiStar. In *Proc. of Operating Systems Design and Implementation (OSDI)* (2006).