TEAL: Transparent Encryption for the Database Abstraction Layer

Karl Lorey[1], Erik Buchmann[2], and Klemens Böhm[1]

[1] Karlsruhe Institute of Technology, 76131 Karlsruhe, Germany
karl.lorey@student.kit.edu, klemens.boehm@kit.edu
[2] Hochschule für Telekommunikation, 04277 Leipzig, Germany
buchmann@hft-leipzig.de

Abstract. Outsourcing databases has numerous advantages over hosting traditional servers in-house. However, data containing business secrets or personal information cannot be outsourced to external service providers unprotected. A number of formal frameworks for query processing on encrypted data have been proposed to solve this challenge. However, these frameworks leave open a large number of implementation issues, among others the run-time performance of the encryption and the integration into existing structures.
We demonstrate TEAL, our approach for Transparent Encryption in the Abstraction Layer of a database system. TEAL is an implementation of a framework for secure storage and resides inside Abstraction Layer of a database (DBAL). We demonstrate how TEAL integrates transparently into existing applications. Furthermore, we explain query processing on encrypted data by re-using components of the DBAL. Finally, we visualize the impact of the processing steps of TEAL on the run-time performance of the query processing. Our demonstration shows that TEAL allows to integrate secure storage easily and with a reasonable increase of the execution times into object-relational mapping.

1 Introduction

Outsourcing databases has become a matter of course. By hosting data from many customers, specialized service providers can offer cost-efficient database services with defined service levels that adapt to flexible loads. However, data privacy laws [5] and business secrets in the data frequently prohibit outsourcing unencrypted databases to external service providers. Naive encryption solves this problem, but is equivalent to a restriction to plain storage. To tackle this issue, a number of approaches [6, 1, 3, 8] have been proposed to encrypt databases in a way that certain types of queries can be efficiently executed without knowing the encryption key and without disclosing too much private information [4]. Such approaches use a trusted middleware that rewrites each query from the client. The rewritten query can be executed over the encrypted data at the server side. The server answers then with an encrypted result set that might contain more

In: S. España, M. Ivanović, M. Savić (eds.): Proceedings of the CAiSE'16 Forum at the 28th International Conference on Advanced Information Systems Engineering, Ljubljana, Slovenia, 13-17.6.2016, published at http://ceur-ws.org

data than requested. Subsequently, the middleware decrypts this result and does some post-filtering in order to deliver the final query results to the client.

While this process has been formally described, the realization of such a middleware is not obvious: First of all, the encryption must be transparent both to the database service provider and the application developer, to re-use existing services and software components. Second, in the worst case the service provider must transfer the entire database to the middleware at the client for local decryption. Thus, data management at the middleware is an issue. Third, the middleware must be able to pass transactions between client and database provider. Fourth, it is important that the middleware is interoperable with the specific SQL dialects of different database vendors. Finally, the run-time overhead of the encryption is important.

This demo will feature TEAL, our approach for Transparent Encryption in the Abstraction Layer of a database system. TEAL implements a well-researched framework for secure storage [6] in the database abstraction layer (DBAL). This allows us to abstract from vendor-specific extensions of the SQL standard, and to realize fully transparent database encryption without having to interfere with the transactional mechanisms of the database or the application.

Our demonstration provides insight into three aspects: We show that the integration of TEAL into existing applications is feasible with moderate effort. To this end, we have implemented TEAL for PHP applications using Laravel, that is, we realize secure storage for object-relational mapping. We explain in detail how TEAL encrypts data, executes a certain query over encrypted data and obtains the plain-text query result. As part of our demo we show that the run-time overhead induced by the encryption is reasonable.

Paper structure: The next section introduces the foundations of our approach. Section 3 provides an overview of TEAL. Section 4 describes our demo and Section 5 concludes.

2 Fundamentals

This section briefly describes the approach [6] we have used as a basis for our work and explains the database abstraction layer (DBAL).

2.1 SQL over Encrypted Data

We have based TEAL on Hacigümüş' algebraic framework [6], because it allows exact-match access to all columns with a limited number of false positives in the encrypted result set, which is important for object-relational mapping. In contrast, approaches such as [3] require the client to download and decrypt the entire database whenever a sensitive column is queried. Order-preserving encryption [1] is prone to statistical attacks when the adversary can access the query log. Approaches like CryptDB [8] need to extend the entire server infrastructure from the application server over a proxy server to the DBMS.

In the following, we briefly describe [6]. Let $R(A_1, A_2, \ldots, A_n)$ be a plain-text relation R with attributes $A_1, \ldots, A_n$. The framework transforms this relation

into an encrypted relation $R^S(etuple, A_1^S, A_2^S, \ldots, A_n^S)$. *etuple* is an encrypted copy of the original tuple $A_1, \ldots, A_n$, i.e., the middleware can decrypt the original tuple from *etuple*. $A_1^S, A_2^S, \ldots, A_n^S$ are encrypted and binned index values that allow for certain searches. Due to binning, several plain-text values are mapped to the same encrypted index value. Thus, the service provider cannot infer the order and the distribution of the original data. However, queries on R^S tend to contain false positives. Consider the relation *emp* containing all employees:

emp:	**id**	**name**	**salary**	**address**
	1	Alice	1000	N.Y.
	2	Bob	2000	L.A.
	3	Carol	3000	N.Y.

This relation would be encrypted and stored as encrypted table *enc* as follows:

enc:	**etuple**	$\mathbf{I_1}$	$\mathbf{I_2}$	$\mathbf{I_3}$	$\mathbf{I_4}$
	enc(1, Alice, 1000, N.Y.)	e_1	e_5	e_3	e_2
	enc(2, Bob, 2000, L.A.)	e_7	e_5	e_6	e_2
	enc(3, Carol, 3000, N.Y.)	e_7	e_4	e_6	e_2

With $e_1 \cdots e_7$ we refer to encrypted index values. Some plain-text values are mapped to the same encrypted value. For example, Employee Id 1 has been mapped to bin e_1, while the Ids 2 and 3 have been mapped to e_7.

To process queries on encrypted data, the encryption middleware rewrites plain-text queries from the client according to translation rules for relational algebra [6]. For example, `SELECT name, salary FROM emp WHERE id = 2` translates to `SELECT etuple FROM enc WHERE` $\mathtt{I}_1$ `=` e_7. The server executes the rewritten query over encrypted data without knowing the decryption key. It returns an encrypted result set to the middleware. The translation rules ensure that the result set is a superset of the data queried. In our example, the result set is {`enc(2, Bob, 2000, L.A.), enc(3, Carol, 3000, N.Y.)`}. Finally, the middleware decrypts the result set and computes the final query result. The encryption can be parametrized so that for typical queries the effort for decryption and post-processing on the client side is small. However, in extreme cases the server might send the entire encrypted database to the client. While the formal process is well defined (cf. [6]), its practical realization leaves open a number of issues.

Querying encrypted data always reveals some information to the service provider [4]. Refinements exist that mitigate the impact of some issues, e.g., for range queries [7, 2]. TEAL allows to integrate such approaches easily.

2.2 The Database Abstraction Layer

Many Database Abstraction Layers (DBALs) have been proposed so far to overcome the heterogeneity introduced by SQL vendors, to gain independence from SQL dialects and from database vendors. Popular DBALs are object-relational mappers such as hibernate[3], the Java Persistence API[4] or Laravel[5].

[3] http://hibernate.org
[4] http://www.oracle.com/technetwork/java/javaee
[5] http://laravel.com

Other DBALs provide language-level abstraction (JDBC, ODBC) or service-level abstraction (e.g., Apache SPARK[6]). The resulting software is vendor-agnostic, meaning that the underlying DBMS can be changed easily.

A transparent integration into a DBAL allows to migrate to secure storage without having to re-write existing applications. It also makes TEAL independent from specific database vendors, because this is already obtained by DBAL. Furthermore, an integration into DBAL allows a seamless integration into the transaction processing between application and database server. The integration into a DBAL allows to re-use efficient libraries for query processing. Finally, DBALs focus on programming interfaces and prohibit direct access to indexes and other database-specific features. This makes it easier to realize a secure storage approach, because no side-channels have to be considered.

To demonstrate that TEAL provides secure storage which can be integrated into existing applications seamlessly, we have implemented it in PHP using Laravel. PHP and Laravel are typical components of LAMP [7] servers that handle a wide share of today's Web load. Laravel is a DBAL that provides an object-oriented persistence interface for MySQL, Postgres, SQLite, and SQL Server.

3 Transparent Encryption in the DB Abstraction Layer

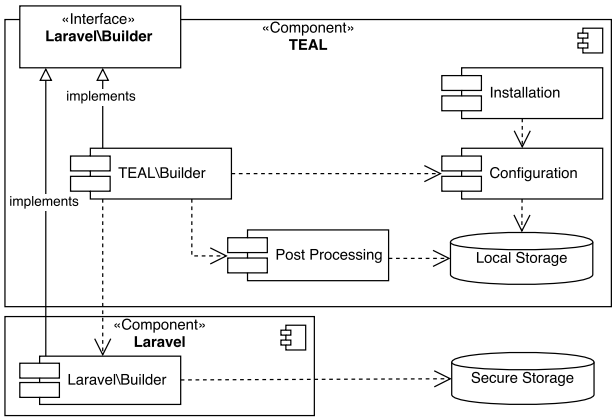

Fig. 1. TEAL software components.

TEAL provides secure storage [6] within a Database Abstraction Layer. It integrates encryption, query rewriting and post-processing of decrypted results into the Laravel object-relational mapper, as shown in Figure 1. 'Laravel\Builder' subsumes Illuminate\Database\Query\Builder and connected Lar-

[6] http://spark.apache.org
[7] LAMP refers to a software package containing Linux, Apache, MySQL and PHP.

avel classes. Dashed lines represent 'uses' relations between components, and continuous lines stand for 'implements' relations.

TEAL consists of four components:

Configuration: This component stores all meta data that is needed for encryption, query rewriting and database access in a local database on the client. This includes the encryption and binning methods used, the encryption key, and the range and number of bins used for the encrypted indexes. It also includes the mapping from plain-text identifiers of databases, tables and columns on the client side to encrypted ones on the server side.

Installation: The installation component comes as an installation package that reconfigures an existing LAMP installation for secure storage. That is, it integrates TEAL in between the PHP application and the Laravel library so that TEAL can encrypt queries and post-process encrypted query results. Furthermore, the installation provides a user interface to the configuration parameters.

Builder: The TEAL\Builder obtains plain-text queries from the PHP application via the Laravel\Builder interface. It translates them to queries over encrypted data, and passes this translation to the native Laravel library. Conversely, the Builder obtains and decrypts the encrypted intermediate results from the server side. It passes them to Post-Processing and transfers the final result set to the PHP application. To this end, the TEAL\Builder interacts with the Laravel\Builder component.

Post-Processing: The translated query yields an encrypted superset of the data sought. The Post-Processing computes the final query result from the decrypted superset. This includes sorting, grouping, and filtering. In the first step, our Post-Processing component evaluates predicates from the `WHERE`-condition of the query. In a second step, we put the resulting data set into a client-side database system[8] and execute the original (unencrypted) query. The client-side database must have the same vendor and schema as the original database. This lets us obtain the final query result efficiently and without having to re-implement database functionality inside the DBAL.

4 Demonstration

Our demonstration database for TEAL will have 100,000 rows, one column of sequential tuple IDs and two columns of uniformly distributed integers. This results in a table size of 4.3 MB and an index size of 2.2 MB for the primary key. The given data distribution allows us to demonstrate queries with a given selectivity. We use AES-256-CBC for encryption. The encrypted indexes are binned with a bucket size of 100, i.e., on average each bucket holds 1000 values. We are executing our demonstration on a machine with PostgreSQL 9.3, Intel Core i7 4770k, 16GB RAM and a Solid State Disk. Our PHP application mimics a typical dynamic web application.

[8] 'Local Storage' in Figure 1

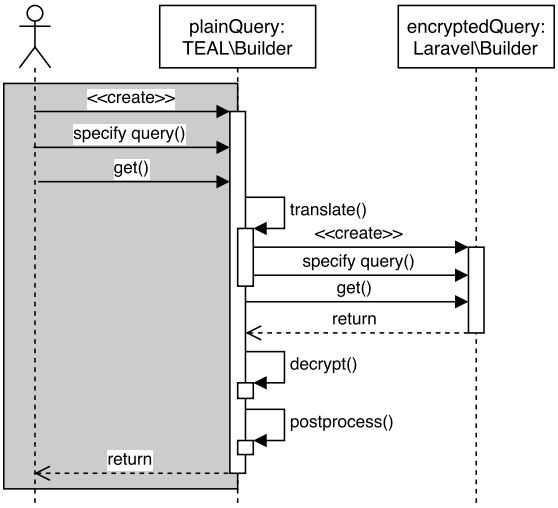

Fig. 2. Sequence diagram of a query execution.

4.1 Transparent Encryption

First we demonstrate how TEAL integrates seamlessly and transparently into existing PHP applications. To this end, the installation component of TEAL comes with a setup wizard. Once TEAL has been installed, the setup wizard can be accessed via a web interface. We will guide the audience through the various steps of the setup. This includes the selection of the tables to be encrypted, the selection of the keys and parameters for the encryption and the binning, and the configuration of a database connection to a third-party provider storing encrypted data. Based on this information, the wizard converts the plain-text database into an encrypted one and transfers it to the database provider.

At the end of this part of the demonstration, a secure storage at the server side exists. It can be used by modifying a line in the configuration file of Laravel, i.e., without having to modify the PHP application. With our demonstration setup, the secure storage contains one table with four columns: the *etuple* column containing the encrypted tuples and the three columns containing encrypted and binned index values. The encrypted table has a size of 11 MB and an index of the same size.

4.2 Querying Encrypted Data

Second, we demonstrate how TEAL rewrites queries for execution on the encrypted storage, and how it computes the result set from the encrypted answer (see Figure 2).

TEAL obtains a plain-text query over the Laravel API, i.e., all method calls to Laravel\Builder are redirected to TEAL\Builder. In detail, Laravel allows the application developer to formulate queries using object-oriented method calls

that correspond to `FROM`, `WHERE` conditions, `JOIN`, `GROUP BY` etc. The Builder constructs an internal query representation from these calls. Figure 2 subsumes these calls under `specify query()`. To execute a query on the database, the application eventually invokes the method `get()`.

Once `get()` is invoked, TEAL translates this query representation into one that can be executed on encrypted data [6], using the original method calls of Laravel. Our demonstration will visualize this transformation process for arbitrary queries. Laravel executes the encrypted query and returns an array of encrypted objects containing *etuples* to TEAL\Builder. After decryption, this array contains more data objects than requested and needs to be filtered and post-processed. Again, we will demonstrate the decryption and post-processing step. Finally, TEAL returns the array with the final query result to the PHP application through the Laravel API.

4.3 Run-Time Performance

The third part of our demo addresses the run-time penalty of TEAL. To this end, we measure the execution time of the query which we have demonstrated.

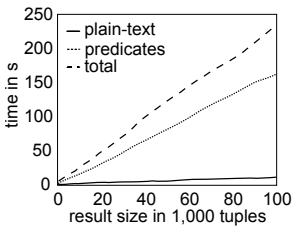

Fig. 3. Run-time penalty of encryption.

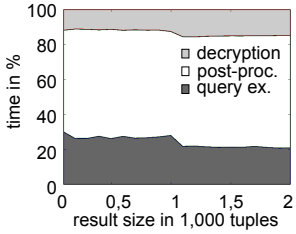

Fig. 4. Run-time of the processing steps.

We show the increase of the run-time due to using secure storage first. In particular, we vary the `WHERE` condition of the query to obtain query results of varying size (cf. Figure 3). First we determine the execution time of a plain-text query processing without using TEAL (continuous line in Figure 3). Second, we measure the run-time of a query processing with TEAL by filtering for the `WHERE` predicates, i.e., without fetching the filtered query result into a client-side database for further post-processing such as grouping and sorting (dotted line in the figure). Third, we measure the total run-times using TEAL (dashed line).

Finally, we visualize the proportions of the various processing steps of TEAL on the total run-time of a query execution (see Figure 4). Again, we vary the result size by modifying the `WHERE` predicates of the query. We will show the execution times of the query rewriting, query execution (dark gray in the Figure), result decryption (white) and post-processing (light gray). Query execution, which is handled by Laravel, has a share of about 30% of the execution time,

while client-side decryption needs around 60% of the time. Notice that query execution and decryption are handled by a specialized DBMS and a highly optimized crypto-library. Thus, this part of the demo shows that integrating TEAL into a less performant scripting language such as PHP does not incur unreasonable costs.

5 Summary

The purpose of this demonstration is to show how a formal framework for query processing on encrypted data can be integrated into an existing software architecture. To this end, we present TEAL, our approach for Transparent Encryption in the Abstraction Layer of a database system (DBAL). TEAL implements a state-of-the-art framework for secure storage that re-uses the application programming interface of the Laravel DBAL. This allows TEAL to integrate seamlessly into PHP applications using Laravel, including object-relational mapping, transaction management, storage management and database-vendor independence.

Our demonstration reveals how our installation component achieves this integration. Furthermore, we explain how TEAL re-writes and processes a given plain-text query, and how it obtains the correct plain-text query result from the encrypted response provided by the secure storage provider. Finally, we show the increase in the execution times of a query as a result of encryption. Our demonstration visualizes which processing step of TEAL requires which share of the execution time. We demonstrate that TEAL scales linearly with the size of the query result.

References

1. R. Agrawal et al. Order Preserving Encryption for Numeric Data. In *Proc. of the ACM SIGMOD Intl. Conference on Management of Data*, 2004.
2. R. Agrawal et al. Order Preserving Encryption for Numeric Data. In *Proc. of the ACM SIGMOD International Conference on Management of Data*, 2004.
3. V. Ciriani et al. Combining Fragmentation and Encryption to Protect Privacy in Data Storage. *ACM Transactions on Information and System Security*, 13(3), 2010.
4. E. Damiani et al. Balancing Confidentiality and Efficiency in Untrusted Relational DBMSs. In *Proc. of the ACM Conference on Computer and Communications Security*, 2003.
5. European Parliament and the Council of the European Union. Directive 95/46/EC. Official Journal L 281, 11/23/1995, p.31., 1995.
6. H. Hacigümüş et al. Executing SQL over Encrypted Data in the Database-Service-Provider Model. In *Proc. of the ACM SIGMOD Intl. Conference on Management of Data*, 2002.
7. B. Hore, S. Mehrotra, and G. Tsudik. A Privacy-preserving Index for Range Queries. In *Proc. of the International Conference on Very Large Data Bases*, 2004.
8. R. A. Popa et al. CryptDB: Protecting Confidentiality with Encrypted Query Processing. In *Proc. of the ACM Symposium on Operating Systems Principles*, 2011.