Encrypting Analytical Web Applications

Benny Fuhry
SAP Research
benny.fuhry@sap.com

Walter Tighzert
SAP Research
walter.tighzert@sap.com

Florian Kerschbaum
SAP Research
florian.kerschbaum@sap.com

ABSTRACT

The software-as-a-service (SaaS) market is growing very fast, but still many clients are concerned about the confidentiality of their data in the cloud. Motivated hackers or malicious insiders could try to steal the clients' data. Encryption is a potential solution, but supporting the necessary functionality also in existing applications is difficult. In this paper, we examine encrypting analytical web applications that perform extensive number processing operations in the database. Existing solutions for encrypting data in web applications poorly support such encryption. We employ a proxy that adjusts the encryption to the level necessary for the client's usage and also supports additively homomorphic encryption. This proxy is deployed at the client and all encryption keys are stored and managed there, while the application is running in the cloud. Our proxy is stateless and we only need to modify the database driver of the application. We evaluate an instantiation of our architecture on an exemplary application. We only slightly increase page load time on average from 3.1 seconds to 4.7. However, roughly 40% of all data columns remain probabilistic encrypted. The client can set the desired security level for each column using our policy mechanism. Hence our proxy architecture offers a solution to increase the confidentiality of the data at the cloud provider at a moderate performance penalty.

Keywords

encrypted database; encrypted web application; homomorphic encryption; stateless proxy

1. INTRODUCTION

When outsourcing software to the cloud, clients are concerned about the confidentiality of their data, but they still want to process as much data as possible in the cloud. Motivated hackers or malicious insiders could try to steal the clients' data. Furthermore, governments now have two options to seize the clients' data – at the cloud provider and at the client – which may cause problems for the clients in case of cross-border investigations.

Encryption is a commonly proposed countermeasure to this problem, but processing data while it is encrypted is difficult. Fully homomorphic encryption [19] is the panacea; highly secure, but currently too slow for practical adoption and difficult to integrate into existing cloud applications. Commercial vendors [1, 2, 3, 4] offer (format-preserving) deterministic and order-preserving encryption as an alternative. This solution has the advantage that the application does not need to be modified and the performance penalty is small. Yet, the security of the solution is questionable, since the encryption must not interfere with any possible function of the application or the application's functions must be restricted. For example, if the application displays a table of results that can be sorted according to any column, by default all columns must be order-preservingly encrypted whether the sorting is used or not. Encryption for processing numbers – as limitedly offered in additive homomorphic encryption – is not supported at all by these solutions.

Mylar by Popa et al. [40] uses (multi-key) searchable encryption. This allows stronger (than deterministic) encryption, but requires to implement a different database and search interface in the application. Moreover, searchable encryption only offers limited search capabilities (equality in the case of Mylar) and processing of encrypted numbers is not supported.

In this paper, we present another solution approach to this problem. We adjust the encryption to the functionality needed by the application and allowed by the data owner. The idea of onion encryption was first popularized by CryptDB [39], but only the databases can be outsourced to the cloud while remaining protected. We extend the protection to cloud based web application by introducing a proxy. The end users only have to utilize their web browser to access the web application. While our solution provides extended protection, it achieves a couple of architectural advantages:

- *Encryption is adjusted* to the application's control-flow. Only those database fields that are being used for search are encrypted deterministically (or order-preservingly). The other fields remain on a stronger encryptions.
- *Additively homomorphic encryption* is supported for processing numbers as prevalent in many analytical applications.
- Although our encryption scheme is stateful the proxy performing the encryption and decryption at the client

CCSW'16, October 28 2016, Vienna, Austria

© 2016 Copyright held by the owner/author(s). Publication rights licensed to ACM. ISBN 978-1-4503-4572-9/16/10...$15.00

DOI: http://dx.doi.org/10.1145/2996429.2996438

is *stateless*. The proxy encrypts the requests and decrypts the responses in adjustment to the application's control flow and functions, but it only has to store the keys. This implies that the proxy can be easily replicated, administered and scaled for performance. The proxies even can be deployed at spatially separated locations of multi-national clients.

- Only the applications' *database driver of the application server* – a standard component – *is modified*. The web application can continue to query using the full functionality of SQL. The database – including its search algorithms – is not modified.

We are aware that when handling plaintext data in the web browser cross-site scripting attacks may reveal this data to adversaries. ShadowCrypt [22] offers a solution to this problem, but our solution is orthogonal to their approach enhancing the encryption for protection against the cloud server. Nevertheless, as we will discuss later, our proxy is able to partially process database queries which may pose a problem for computationally weak devices, such as smartphones, when implemented in the browser (as ShadowCrypt is).

The remainder of the paper is structured as follows: In the next section, we review related work for systems operating on encrypted data in the cloud. We then present the architecture, system components and threat model of the web applications we support in Section 3. In Section 4 we describe the algorithms implementing the encryption, decryption and database operations in this architecture. We present the evaluation results of our implementation in terms of performance, networking overhead and security in Section 5. In Section 6 we give the conclusions of work.

2. RELATED WORK

We distinguish three types of related technologies: encryption between application server and database, encryption using a proxy, encryption in the browser.

2.1 Database Encryption

Several technologies exist for end-to-end encryption of a database. Hacigümüs et al. introduced processing SQL directly over a deterministically (DET) encrypted data in [20]. In their scheme search is very efficient, since the database does not need to be modified and search operates as on plaintexts. Still, range queries were problematic in their original proposals. Order-preserving encryption (OPE) [8] solved range queries also with plaintext search. OPE was later formalized by Boldyreva et al. in [10, 11]. Still, static OPE cannot achieve optimal security and later schemes with ideal security were presented [31, 38].

Popa et al. built the CryptDB system [39] on top of this research. It uses onion (adjustable) encryption and modifies the database driver for encryption and decryption. We advance over this approach by extending the encryption to the web application running on an application server. This fits the common software-as-a-service (SaaS) model better than solely database encryption. The encryption keys reside at the client and the cloud servers cannot access plaintext data. Thereby, the database server and the application server can be deployed in the cloud, while the total control of the plaintext data is solely in the hands of the client.

Song et al. introduced searchable encryption as a standard semantically secure encryption scheme that only leaks the search and access pattern of queries [43]. A number of schemes were introduced later with sublinear search time [12, 15, 21, 23]. Yet, all of the approaches based on searchable encryption suffer from limited search capabilities. Many functions, such as grouping, joins or aggregations – as commonly used in any SQL database – are not supported. Hence using searchable encryption requires significantly changing the application and is not suited at all for our targeted analytical applications.

Another approach of securing databases in the cloud is to separate the database between two cloud providers. Aggarwal et al. first introduced this concept in [7]. Later approaches extended this to multiple databases and cloud providers [14, 18, 33].

While this is a very suited approach in terms of performance, since little to no encryption is required, it is somewhat questionable from a security perspective. It rests on the assumption that the cloud providers offer a common service, i.e., they collaborate, but do not exchange data, i.e., they do not collude. This seems somewhat paradoxical given that the working assumption for all of these approaches is that the cloud provider is untrusted. Our approach works with a single cloud provider. We see this as a clear advantage.

There exist also a large number of encrypted database and web applications which have been developed with specific protocols for the application, e.g., benchmarking [24, 25, 32, 13, 28], RFID tracking [30] or reputation systems [26].

2.2 Encryption By A Proxy

Diallo et al. presented CloudProtect [16] in 2012. The idea is to introduce a proxy that encrypts data before sending it to the cloud and decrypt it in responses. However, complex operations – such as aggregations – require to decrypt the involved data at the cloud, because these operations are not possible with (deterministic) encryption. Data is re-encrypted after the operation was performed.

As with all of these approaches (until our proxy), the functionality supported by the proxy is quite limited. CloudProtect supports Google Calendar and Google Docs. The proxy needs to be adapted to these or new applications, i.e., when the application changes the proxy might need to change. We anticipate that this model of operation does not scale, since thousands if not millions of applications need to be supported in the future. We therefore built our proxy into an architecture that is independent of the application.

HPISecure [42] was later introduced and is similar to CloudProtect. It does not support adaptive decryption and re-encryption and hence has a simpler, easier-to-deploy design, but does not support the same functionality or security. Nonetheless, it is still dependent on the functionality of the application and needs to adapt its configuration to application changes.

There are several commercial systems that follow the proxy encryption model, e.g. [1, 2, 3, 4]. All of these suffer the same problems as the two approaches described before. A functionality and security feature supported by our architecture is not even mentioned in any of these solutions: the ability to process encrypted numbers using additively homomorphic encryption.

2.3 Encryption In The Browser

Popa et al. also developed Mylar [40], a system to build web applications using searchable encryption. As mentioned before, any system based on searchable encryption requires modified search functions. A novel feature of Mylar is its support for data sharing by proxy re-encryptable searchable encryption. Proxy re-encryption [9] supports changing an encryption key without intermediate decryption. Hence each user can have its own key and still the cloud provider does not learn plaintexts, but can perform (simple) searches. Mylar employs several features, such as code signing, in order to protect the data in the browser from malicious client-side application code. A browser extension is required for this functionality.

Our architecture works independent of the browser and does not require any modification to it. It is not incorporated in our current implementation, but it is straightforward to add a code signature verification at the proxy. Thereby, even weaker devices can benefit from this security feature and several dedicated browser extension for different operating system are unnecessary. We also require less change to the application. Further, we support more operations than simple search, such as grouping, joins and aggregations that are typical for analytical applications.

ShadowCrypt [22] is also installed as a browser extension. It uses the browser's Shadow DOM to perform encryption and decryption. This has the strong advantage that malicious application code cannot access the data. This is indeed a viable alternative technology to our proxy encryption approach once the Shadow DOM is generally available in all browsers. Still, the ShadowCrypt approach so far only supports the protection of textual data and no calculations at all. It is orthogonal to our approach and therefore would benefit from extension with the type of technologies we present in this paper.

3. ARCHITECTURE

In this section, we describe the application architecture our implementation supports and the threats we protect against. We emphasize that our implementation is only one instance of the general architecture that supports a wider range of technologies than we could support in our implementation.

3.1 System Components

Figure 1 gives an overview of the components involved in our design. The end users utilize a web browser on their preferred device to connect to a web application. The web application is deployed on a server at the SaaS provider and it is backed by a database on at a DBaaS provider. The SaaS and DBaaS providers and the servers are not necessarily different. The main component we introduce is a proxy component between the end user's browser and the application server that is able to intercept every message. We additionally modify the database driver at the application server.

On end user interaction, the browser downloads the client-side application resources (e.g. HTML file, images, JavaScript files) and separately loads the content data afterwards. We target our implementation to this two-tier web application architecture that cleanly separates content data requests from other application resources. We found such an architecture in SAP's UI5 framework for web applications.

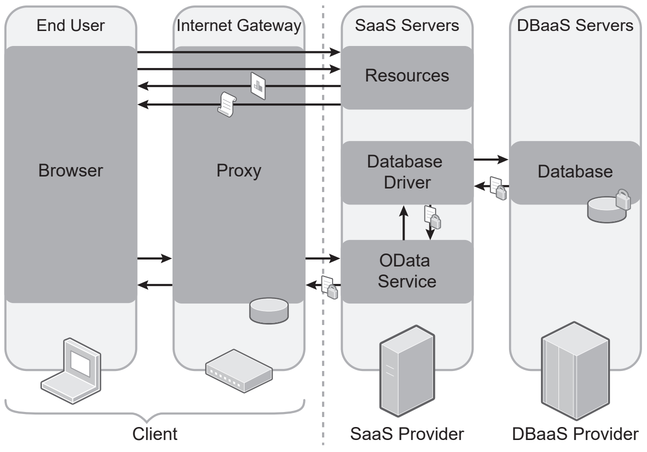

Figure 1: Architectural Components

Another example is the Meteor JavaScript web application framework. However, the separation is no strict prerequisite for our concept. The proxy could also scan every transferred message for relevant information, but this has negative performance implications and it is error-prone.

In SAP's UI5, the application resources are downloaded and most of the client-side application code is executed as JavaScript in the web browser. The JavaScript issues content data requests via OData requests, which are processed by the HANA XS Engine on the application server. OData [36] is a data access protocol that allows resources, identified using URLs and defined in a data model, to be published and edited by web clients using simple HTTP messages. It shares some similarity with JDBC and ODBC, but OData is not limited to relational databases. The HANA XS translates these OData requests into SQL queries for the HANA database back end. The client-side application code processes the OData responses and displays them using dynamically created HTML.

Our proxy intercepts every message that is passed through it. Then, it decides if further processing is necessary based on a special HTTP header. This approach allows client-side application code that is not written explicitly for our design. Only the special header must be added at all messages that contain sensitive information. This allows great flexibility, because applications that use our protection and others can be used in parallel without explicit routing.

In our implementation, only the OData content data requests and responses contain this header, because only these messages contain sensitive information. The proxy encrypts data values in requests and decrypts them in responses. As a result, the application and database operate only on encrypted data. The OData service transforms the incoming OData requests to SQL queries and transforms the SQL result sets to OData responses. We modify the database driver that performs this transformation to match to the adjustable encryption in the database.

The main task of our proxy is to encrypt and decrypt values. However, we explain in Section 4.2.5 that the proxy additionally supports post-processing of queries. This is required if a part of the SQL query that was created by the OData service is not supported on encrypted data. Loosely speaking, the database driver splits the database query into two parts in this case: the first one that can be executed on encrypted data and the second one that – in combination with the first one – cannot be executed on encrypted data. The second one, however, can be executed after decryption. The first and the second query composed sequentially deliver the same result as the original query. The result of the first database query – along with the second query – are sent to the proxy. The proxy decrypts the result, stores the plaintext values in a local database and executes the second query on the plaintexts. Only the result of the second query is sent to the browser.

Our proxy also supports TLS requests and responses (i.e., HTTPS connections) by introducing a man-in-the-middle: The proxy pretends to be the server towards the client and pretends to be the client towards the server. Commonly, TLS would prevent this "attack", because the session key is tied to the public key of the server. This public key in turn is part of the server's certificate, which is signed by a trusted certificate authority (CA). The proxy has no access to the server's private key and thus could not access the session key and read the encrypted contents. The workaround is to register the proxy as a trusted CA by installing a root certificate on the client's device (works as long as HTTP Public Key Pinning (HPKP) [17] is not employed). The new CA is then used by the proxy to create dummy certificates for each TLS protected site that the client visits. The proxy changes the actual public key to a key for which it knows the corresponding private key and is then able to decrypt the messages. It is important to note that this introduces a man-in-the-middle for every connection routed through the proxy, i.e., the proxy is able to read every message exchanged with every HTTPS protected website. However, this may be an acceptable approach, especially in an enterprise setting, because the proxy is trusted by the clients and most enterprises like to control all clients. Further security implications depend on the deployment of the proxy. In the remainder of this paper, we assume an unprotected connection.

3.2 Threat Model

The dotted line in Figure 1 shows the trust boundary. The components on the left are assumed to be trusted. Our system is designed to increase the burden for accessing sensitive data at the cloud provider's site (right). The threat entails hackers or malicious insiders trying to access the client's data.

We do not, however, protect against a malicious application programmer. Since a significant portion of the application is executed in JavaScript on the plaintext at the browser, the application could siphon the plaintext to any untrusted destination. Our design is compatible with code signing approaches for the code executed at the browser, but these also only introduce a level of indirection, since the application provider (programmer) needs to sign the code and is inherently trusted. Our proxy is ideally suited to verify the signatures and hence no modifications to the browsers, such as a browser extension, are necessary.

Currently, we also do not protect against malicious attackers that exploit vulnerabilities in the application's code. Cross-site scripting attacks may introduce (unsigned) JavaScript code that can again siphon the decrypted query results to any destination. As mentioned before, our approach can be combined with ShadowCrypt [22] to protect against this threat.

Besides the encryption of the client's data, we implement a policy mechanism for key release (see Section 4.2.3). In order to adjust the encryption, the database driver needs to receive the key for the higher layers of the onion encryption. We implement a policy mechanism, such that the client's proxy will not release arbitrary keys to the cloud provider. Instead, the proxy checks against a policy for an allow or deny decision defined by the client. The application's performance may be adversely affected in case of deny, but the security settings are preserved. This prevents key exfiltration attacks by the cloud provider.

Our encryption is designed to provide the maximum security given the functionality and performance needed to execute the application. We only encrypt those columns deterministically that the client actually used for search. Hence we anticipate several columns to remain under a stronger than deterministic encryption. Furthermore, we integrate additively homomorphic encryption for restricted number processing. Using proxy post-processing we support the full functionality of SQL and the application programmer can resort to this standard interface.

4. ALGORITHMS

4.1 Encryption Schemes

We employ several different encryption schemes that support different operations in SQL. We use probabilistic encryption (standard AES encryption in GCM mode), deterministic encryption (Pohlig-Hellman encryption [37] over the elliptic curves), order-preserving (Boldyreva et al.'s encryption scheme [10]) and additively homomorphic encryption (Paillier encryption [35]). We briefly explain the features of each encryption scheme (except AES, which we consider well-known).

4.1.1 Deterministic Encryption

Deterministic encryption maps each equal plaintext to a single ciphertext. Hence equality comparisons can be performed between ciphertexts as well as between plaintexts. This enables the database in the cloud to perform equality comparisons using the same algorithm as used for plaintexts – no change to the database is required.

By default, we encrypt each database column with a different key. This allows comparison between the column-values and a constant supplied in the query (encrypted with the same key), but comparison between columns are not possible. Proxy re-encryption is used to also allow these comparisons: the cloud provider transforms ciphertexts encrypted with key K_A to ciphertexts encrypted with K_B without revealing the plaintext and without downloading the data. Changes are made persistent to save costs for future queries. Our database driver loads the current state out of the database and follows the algorithm by Kerschbaum et al. [29] for choosing which column to adjust. This algorithm converges to a stable state and it achieves the best possible worst case bound of necessary re-encryption.

4.1.2 Order-Preserving Encryption

In order-preserving encryption, the order of the plaintext is preserved in the ciphertexts. Hence greater-than comparisons can be performed between ciphertexts as between plaintexts. This enables the database in the cloud to perform greater-than comparisons using the same algorithm as used for plaintexts.

We encrypt each database type with its own key, since we assume that comparisons between different types are the rare exception and no proxy re-encryption scheme for (secure) order-preserving encryption is known. Furthermore, our order-preserving encryption scheme needs to be deterministic, because the onion layering of the encryptions that we use: the order-preserving layer is surrounded by a deterministic encryption layer. Every lower layer has to support functionality of higher layers.

Naveed et al. [34] presented attacks on deterministic and especially on order-preserving encryption. Our solution applies a policy checking mechanism (see Section 4.2.3) that provides protection against these attacks, because the client is able to prohibit the decryption of columns with low frequency. Furthermore, the weaknesses are orthogonal to our approach and recently presented OPE schemes [27, 41] that counter these attacks, can be used.

4.1.3 Additively Homomorphic Encryption

In additively homomorphic encryption, a specific operation on the ciphertexts maps to addition of the plaintexts. Hence, additions can be performed using the ciphertexts. Yet, the result of the addition is still encrypted in the homomorphic encryption scheme. This can be used to implement aggregations in the database. One needs to replace plaintext addition by the homomorphic operation (in our case modular multiplication). In many databases this can be done using user-defined functions (UDF), although a native implementation is usually more efficient. Multiplication between encrypted columns is not supported by our homomorphic encryption scheme, but we support multiplication by a plaintext (e.g. a value added tax rate).

4.2 Encryption Adjustment

4.2.1 Encryption Layers

Every value in the database is encrypted by multiple onions with layered encryption schemes. The database operations in SQL queries are matched to specific layers. One onion initially stores a 3-layer ciphertext for each data item x in a column. x is first encrypted using order-preserving encryption: $E^{OPE}(x)$. This ciphertext is then secondly encrypted using deterministic encryption: $E^{DET}(E^{OPE}(x))$. Finally this already layered ciphertext is encrypted using probabilistic encryption: $E^{RND}(E^{DET}(E^{OPE}(x)))$. Another onion stores x surrounded by additively homomorphic encryption: $E^{HOM}(x)$.

All keys for the encryption schemes are maintained only at the proxy. Let $\mathcal{K}_X^{RND|DET|OPE}$ be the secret key for column X for the respective encryption scheme RND, DET or OPE. In order to simplify key management, we derive all keys from a master key using a secure key derivation scheme.

4.2.2 Adjusting Decryption

The onions are designed in a way that the lower layers support strictly more operations than the higher layers. Furthermore, we can safely assume that probabilistic encryption is more secure than deterministic, which is more secure than order-preserving encryption. We follow the basic idea of CrpytDB [39] and adjust the encryption as needed by removing encryption layers.

The database driver maintains the state of the encryption of each column. Whenever the client-side application issues a query the database driver determines the database operations executed by the query, maps those to the respective encryption schemes and compares them to the state of the encryption. An adjusting decryption is performed, if the column is encrypted using a higher layer than required. The proxy sends the required key(s) along with the request to the database driver, which then removes the higher layer(s) by decryption. If the column is already encrypted in appropriate layer, no action is necessary.

The adjustment is never reversed except by administrator intervention. The assumption is that once the cloud provider has learned a ciphertext, it can use it for cryptanalysis. Since it may be difficult to determine when exactly a successful attack has occurred, we operate under the worst-case assumption that the cloud provider is always subverted by the adversary.

4.2.3 Policy Checking

On the one hand, an advantage of adjustable onion encryption is the flexibility that the executed queries do not need to be known in advance, but the encryption is adjusted to the queries actually executed. Thus, only the actually taken execution paths influence the level of encryption. On the other hand, the application may perform queries that require encryption levels, which are unacceptable to the client. To cope with this problem, we implement a policy checking mechanism in addition to the key management in the proxy.

A policy may specify which encryption schemes are allowed for a database column. Usually it only makes sense to specify policies according to the encryption layers, i.e., whether a column may be exposed in deterministic or order-preserving encryption, although our policy language allows arbitrary specifications.

Consider the following examples: Medical images can be stored in a cloud database. Let this column be named `xray`. In most applications there is little need to search these in the database. Therefore, the data can remain encrypted with probabilistic encryption. This corresponds to the encryption level desirable by data protection legislation. The client can set a policy that the only allowed encryption layer for column `xray` is RND implying that the keys $\mathcal{K}_{xray}^{RND}$ and $\mathcal{K}_{xray}^{DET}$ are never revealed to the cloud servers.

In order to compress the policy specification to the most important security settings, we assume a default policy of exposing all encryption layers except the plaintext, i.e., the keys $\mathcal{K}^{OPE}$ and $\mathcal{K}^{HOM}$ are retained at the client at all times. The client then only needs to specify those columns that deviate from the regular behavior of adjustable onion encryption.

4.2.4 Stateless Adjustment

Compared to [39], we do not only protect the database. Instead, we additionally protect the web application. This introduces a challenge: the database driver maintains the state of the database whereas the keys are held and the encryption of the query parameters is performed by the proxy. Hence the proxy, which also must perform (syntactic) query analysis in order to determine the necessary encryption scheme and keys, must operate without knowledge of the state of the database. We solve this challenge in the following manner: The proxy determines the necessary encryption scheme based on the OData query. The proxy assumes that the layered encryption is at the highest layer, i.e., the initial state. The proxy determines all necessary decryption keys for adjusting decryption from the highest layer to the necessary layer. Afterwards, it checks the configured policy-restrictions regarding encryption keys. If no rule denies, it sends the keys along with the query. The database driver analyses the resulting SQL query, determines the necessary encryption scheme and compares them to the database state. If an adjustment is necessary, it uses the supplied keys for decryption and caches the keys. Otherwise, it discards the keys.

The proxy also encrypts the constant parameters of the queries to match the encryption of the database. It assumes that the layered encryption is at the highest layer necessary for the operation and encrypts the parameters correspondingly. The database driver then compares the assumed layer against the actual layer. If the actual layer is lower, it uses the cached keys to adjust the encryption of the parameter by decrypting the higher layer. Then the query is executable on the ciphertexts.

We illustrate this algorithm by an example. Assume we start with a freshly uploaded database and all columns are encrypted using randomized encryption. Furthermore, assume the client starts by issuing the following OData query (we show the corresponding SQL as a footnote):

`http://host/service/people?`
`$select=name&$filter=age ge 21`[*]

The result of this query contains the names of all people 21 and older. Two columns appear in the query – `name` and `age`. Only projection is used for `name`, so it can remain encrypted with probabilistic encryption whereas a range selection is performed over `age`. Hence the constant `21` must be encrypted using order-preserving encryption. Additionally, the `age` column at the database has to be on this layer, which might require the removal of encryption layers.

The proxy encrypts the constant in the OData query and forwards the query to the application server in the cloud. The necessary keys for layer removal are added to this query (if no policy-restriction denies):

`http://host/service/people?`
`$select=name&$filter=age ge` $E^{OPE}(21)$[†]
`KEYS:` $\mathcal{K}_{age}^{RND}$, $\mathcal{K}_{age}^{DET}$

The database driver in the cloud adjusts the encryption of the database column *age* to order-preserving encryption by decryption using the two supplied keys. The query can now be performed on the encrypted database using standard database operators. The keys are stored at the database driver for later layer removals.

Assume now that the client next issues the following OData query:

`http://host/service/people?`
`$select=name&$filter=age eq 65`[‡]

The result of this query contains the names of all people aged 65. The same two columns as above appear in the query, but this time an equi-selection over `age` is performed. Hence `age` must (only) be encrypted using deterministic encryption.

The proxy in this case forwards the following query message to the cloud.

`http://host/service/people?`
`$select=name&$filter=age eq` $E^{DET}(E^{OPE}(65))$[§]
`KEYS:` $\mathcal{K}_{age}^{RND}$

The database driver does not need to perform any adjustment of the database, since the column is already encrypted using order-preserving encryption (a lower layer). However, the parameter is surrounded by a deterministic encryption layer, which does not match the current database state. For that reason, the database driver uses the cached key $\mathcal{K}_{age}^{DET}$ to decrypt the query parameter to $E^{OPE}(65)$. It then performs the query on the encrypted database using standard database operators.

In case of joins the proxy may not be able to precisely determine the necessary keys for re-encryption (or adjustment), since the current key of the column may be different depending on the sequence of previous queries. We therefore offer a callback interface in the proxy for the database driver to request such keys. We apply strict policy checking to these requests (see Section 4.2.3).

4.2.5 *Post-Processing*

Not all encryption schemes support all database operations. For instance, OPE does not support aggregations. Hence, it may be the case that operators follow each other that require incompatible encryption schemes. Common examples are sorting or selecting of aggregate values. Consider the following query:

`http://host/service/people?$select=zipcode,`
`totalIncome&$orderby=totalIncome&$top=3`[¶]

This query returns the zip codes with the top 3 aggregate incomes among all people. In order to execute this query the database first needs to compute the aggregate income for all zip codes via additively homomorphic encryption scheme. Then the database needs to sort the rows by this homomorphically encrypted values. Yet, it cannot do this, because the values are randomized. Hence this query cannot be entirely executed on the database server.

We solve this problem by splitting such queries into two or more parts at the application server. Our proxy sends the entire query including the keys that are required for adjustment ($\mathcal{K}_{zipcode}^{RND}$ in the example) to the application and consequently to the database driver. The database driver then builds the operator tree in relational algebra. It processes the tree from the leaves to the root. For each node it maintains the list of supported encryption schemes. Once it encounters a parent that does not support any of the current encryption schemes it splits the query. This might happen multiple times. The database server executes the lower part(s), which leads to one or multiple temporary results. Additionally, the driver synthesizes the upper part of the SQL query that combines the temporary results to the final result. The temporary table(s) and the upper part are returned to the proxy. It is always possible to create an SQL

[*]`SELECT name FROM ppl WHERE age` $>= 21$
[†]`SELECT name FROM ppl WHERE age` $>= E^{OPE}(21)$
[‡]`SELECT name FROM ppl WHERE age` $= 65$

[§]`SELECT name FROM ppl WHERE age` $= E^{DET}(E^{OPE}(65))$
[¶]`SELECT TOP 3 zipcode, SUM(income) FROM ppl GROUP BY zipcode ORDER BY SUM(income)`

query split that lead to sequential processing at the server and then at the proxy. However, there might exist several split alternatives by rearranging the SQL query to transfer less data. More optimal split mechanisms are subject to on-going research.

In the running example, the part executed on the server is:

```
SELECT zipcode, SUM(income) FROM ppl GROUP BY
zipcode INTO temp
```

The database server uses the key $\mathcal{K}^{RND}_{zipcode}$ to decrypt zip codes in the database to deterministic encryption if necessary. The aggregation is possible with additively homomorphic encryption without any adjustments. The resulting `temp` table contains all zip codes and the aggregated income of people at a specific zip code. This table along with the upper part of the query is returned to the proxy:

```
RESULT: temp[zip,sum(income)]
UPPER PART: SELECT TOP 3 * FROM temp ORDER BY
sum(income)
```

The proxy loads the temporary table(s) into a local caching database, decrypts all values, executes the upper part of the query returned by the database driver and creates a OData response that is forwarded to the browser. The result set returned from the local database after performing the upper part SQL query exactly matches the result of the original query. Since all queries can be executed on plaintext data, this algorithm allows to execute all SQL queries.

We note that – even in the running example – the temporary table(s) may be larger than the final result table. In the running example, the temporary database contains a row for each zip code whereas the result table contains three rows. This is inevitable given the expressive power of SQL. However, the number of zip codes is normally a lot smaller than the number of people, which means that the aggregation of values – the computation intensive work – is performed in the cloud.

In our evaluation we also measure the expansion of message sizes due to our encryption and post-processing (see Section 5.5). We show that the potential increase in data transfer can be managed. Despite the fact that it can be very critical, encryption of numerical values is not supported at all in the commercial solutions available on the market. For instance, a credit card application where numerical values represent personally identifiable data requires this type of encryption. We emphasize that during our evaluation, using real-world applications, we encountered that all queries require a split of the SQL query. For analytical applications it is typical to sort aggregated values. We conclude that without our splitting algorithm encrypting the numerical values would not be possible in analytical applications.

5. EVALUATION

5.1 Test Setup

We implemented our architecture for evaluation. The proxy component was implemented using mitmproxy – a Python proxy – and a Java application for query processing (including post-processing) and encryption/decryption. The modification to the database driver was done directly inside the HANA XS application server. We ran our experiments using the setup as in Table 1.

	Client	Proxy	Cloud Server
Operating system	Windows 8.1 64-Bit	SUSE Linux ES 11 64-Bit	SUSE Linux ES 11 64-Bit
Software	Chrome 44	mitmproxy & Java 7 Server VM	SAP HANA SP7
CPU	Intel Core i5-5300M @ 2.6GHz	Intel Xeon E5-2670 @ 2.6GHz	Intel Xeon E5-2670 @ 2.6GHz
Memory	8GB	256GB	256GB
Network connection	1 Gb/s	1 Gb/s	1 Gb/s

Table 1: Test Environment

5.2 Example Application

We evaluate our proxy's ability to support the query functionality that is required by the SHINE Sales Dashboard. SHINE is a part of a standard SAP HANA deployment and intends to demonstrate the breadth of functionality of native SAP HANA applications [5].

The SHINE Sales Dashboard is a comprehensive analytical application developed for sales managers. It comprises several charts and tables in three tabs containing the sales of a company. Our architecture supports all tabs, but we only describe the the first tab, because it contains the most complex requests. It provides representations of important sales information (see Figure 2). More specifically, four charts are displayed at this tab that represent the following:

- **Top left:** The sales distribution per region (AFR, APJ, AMER, EMEA).
- **Top right:** The sales distribution per country (e.g., DE, US, FR).
- **Bottom left:** Companies with a discount greater than zero for a selected region (sales manager can select the region).
- **Bottom right:** The top ten customers ordered by total sales.

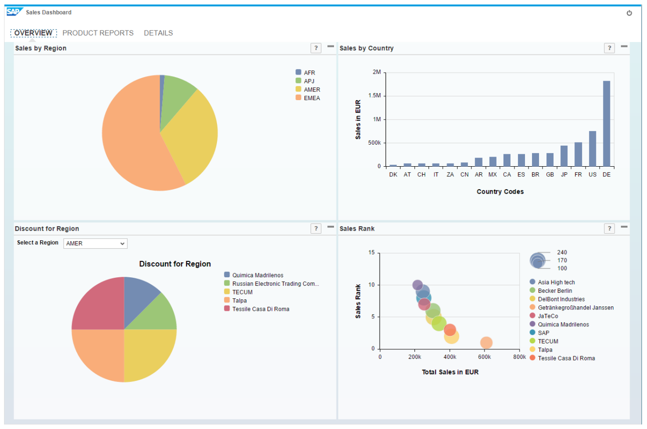

Figure 2: SHINE dashboard screenshot

The four charts are implemented using four OData queries. We emphasize that all four queries require post-processing and hence no number encryption could have been integrated without our novel proxy architecture. We briefly describe the specific queries.

Top left chart (Ch1): The query triggered by this chart is quite complex. The main task of the query is to join several tables with sales order information. Eight distinct tables are involved and about 10,000 unique data rows are selected during the join. Finally the SQL query adds up the net amount for each region in the join result and sorts the total results.

Our architecture is able to perform the complex joins and the sum of sales for each region directly on the encrypted table. Thereby the client's infrastructure saves the computation intensive part of the query. However ordering aggregate values leads to a conflict of encryption schemes and requires post-processing: the proxy sorts the total results after decryption.

Top right chart (Ch2): The query is very similar to the one of the top left chart except that the sum is calculated over each country instead of each region.

Bottom left chart (Ch3): This chart triggers the most complex query in the application. The join operation is equal to the join operation required in the two upper charts. However, the results are not just totalized for this chart. Instead, four additional calculations are executed:

- All sales from a specific region (chosen by the sales manager) within a period of time are selected and the total is computed per company.
- The ten companies with the highest total sales are selected.
- The discount of these companies is evaluated based on the region, number of orders, order rank, total sales and sales rank. The order and sales ranks are classifications that are based on a comparison to other companies in the selected region and time period.
- All companies with a discount greater than zero are selected.

Again, our architecture is able to perform the complex joins directly on encrypted data at the cloud provider. The first additional step requires equality matches and range queries and can also be executed directly on the database. Unfortunately, already the second step requires post-processing, because it uses an order of total values. Therefore, the third and fourth steps can also not be performed in the cloud and have to be done by post-processing at the proxy.

In the example data shipped with the application we had 45 rows in the temporary table. This is still significantly smaller than the thousands of individual sales entries.

Bottom right chart (Ch4): The query of this chart is very similar to the one of the bottom left chart except that only the total sales are returned (without calculation of the discount).

5.3 Performance

The most important performance metric we measure is page load time, since it influences the user's perception of the application. As page load time we understand the time from the beginning of the first HTTP request (triggered by the end user) until the last HTTP response is received and displayed. We use Chrome's DevTools to measure the page load time and only one page load at a time was measured. We compare the page load time for three different setups:

- Plaintext version: plaintext data is used and the connection is directly from browser to application without a proxy.
- Proxied plaintext version: plaintext data is used and every message is routed through mitmproxy but only forwarded.
- Encrypted version: encrypted data is used and every message is routed through mitmproxy. The content data messages are processed by the Java application.

We assume that all necessary layer removals and re-encryptions were already done, because they are only a one time overhead.

Figure 3 depicts the mean page load time of ten experiments including the 95% confidence intervals. Our architecture increases the page load time by an average factor of 1.5 including post-processing in our test setup. This is an absolute value of 1.58s.

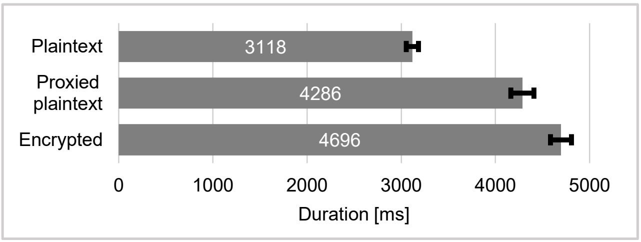

Figure 3: Page load time (ms)

The increase of 1.58s is composed of two parts. Firstly, the time introduced by the (mitm)proxy to decide which messages have to be processed and which are only forwarded. Secondly, the time introduced by the en- and decryption in the (Java) proxy and the processing of encrypted values at the database. The first part is independent of our design, but only due to the used proxy software. The second part is additional time introduced by the design.

Our measurement of the proxied plaintext version in Figure 3 depicts that already the interception is responsible for a factor of 1.4 and 1.17s absolute. This means that the introduction of a proxy into the architecture and especially the not overly efficient processing of mitmproxy is responsible for the largest part of the increased time consumption. This is unfortunate, since it is not the focus of our research, but may falsify our otherwise impressive results. We conclude that a dedicated, fast proxy could vastly reduce the difference between the plaintext and the encrypted version. However, we can assume the proxied plaintext version as a baseline to show the impact of our encryption scheme. The factor between the proxied plaintext and the encrypted version – the factor introduced by data protection – is only 1.1 (410 ms absolute).

We then identify in detail where the additional time is spent by measuring the time for each individual HTTP request by the browser. An HTTP request can be either for application resources (web pages, images, etc.) or for content data (OData data requests). We add the times of all HTTP requests for the three different versions. The results are depicted in Figure 4.

We see in Figure 4 that the sum of the request times increases from 1.9 seconds in the plaintext version to 5.3 seconds in the proxied plaintext version and to 6.1 seconds in

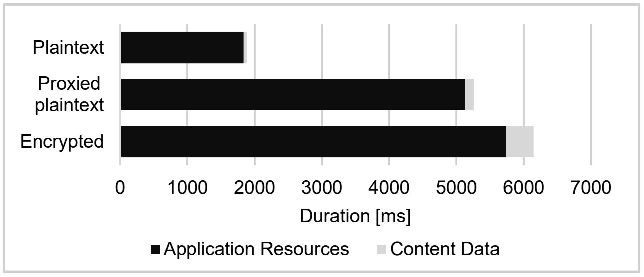

Figure 4: Request overhead (ms)

the encrypted version. This is much larger than the increase of the page load time. The explanation is that we measure a distributed, partially parallel system and the browser can sent multiple requests in parallel and render the display while handling further requests. Hence the impact of prolonged HTTP request is not immediately impacting the user. Furthermore, we see that the time spend for simply forwarded application resources increases by introducing a (slow mitmproxy) proxy. This again substantiate the argument that mitmproxy takes time to process messages and a faster proxy would improve the performance. The time difference at the resources between the proxied plaintext and the encrypted version accounts for the header identification to decide if a message has to be processed. This time would also be reduced with a fast proxy. We emphasize that the content data – i.e., the encryption related – increase in the requests only accounts for less than 0.5 seconds.

We then examine in detail where the time introduced by encryption – the content data part of Figure 4 – is spent. For each request we measure:

- Time spent in the Java component of the proxy, i.e., the time for (post-) processing the query, encrypting parameters and decrypting results.
- Time spent at the application and database server.
- Time spent for network transfer and the processing in mitmproxy (other category).

We measure for each of the four queries – Ch1 to Ch4 – of our application. The results are depicted in Figure 5.

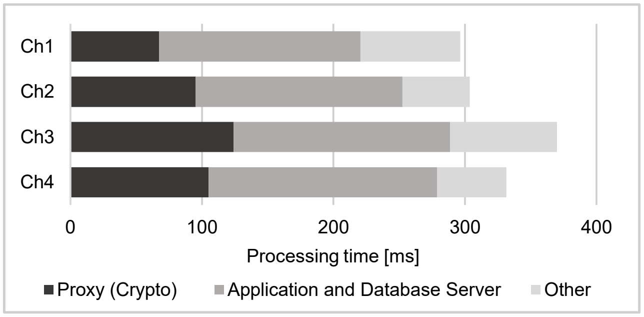

Figure 5: Content data processing time (ms)

We see in Figure 5 that query Ch3 has the highest processing time. This is not surprising, since it is the most complex query. We furthermore see that the most time is spent by the servers processing the encrypted values in the cloud. This is also not surprising, since the database needs to perform modular multiplications for the additive homomorphic encryption. The differences between the queries are mostly accounted for by the processing time in the crypto part of the proxy. We explain the difference by the following two observations. First, for the second part of the queries – Ch3 and Ch4 – the proxy needs to decrypt more entries. Second, it needs to perform more complex post-processing.

Furthermore, we examined the scalability of our approach via an artificial test application. The only functionality of this application is to show a table with data values. The data values are loaded through an OData service and the corresponding SQL query requests data from a single table containing five columns (without any calculations or joins). We present the content data processing time for six different versions of this application in Figure 6. The "DetX" versions request data from tables that contains deterministically encrypted values, i.e., all plaintext values are surrounded by a DET layer. The "ProbX" versions request data from tables with probabilistic encrypted values. More specifically, all values are surrounded by a DET and an RND layer. Besides the encryption, the versions differ in the number of data rows: 10, 100 and 1000.

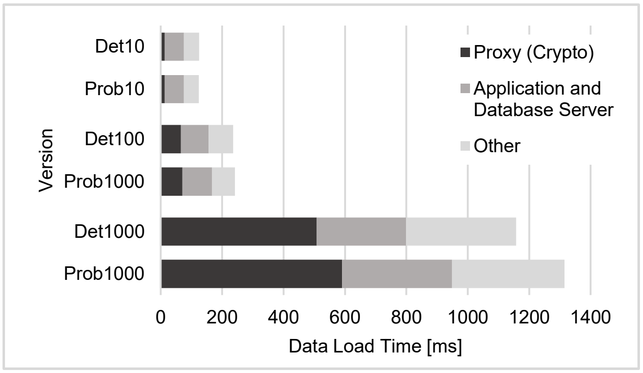

Figure 6: Content data processing time for different query sizes (ms)

The figure shows that the processing time per row does not increase. Instead, the processing time average per row is $12.5ms$, $2.4ms$ and $1.2ms$ for 10, 100 and 1000 deterministically encrypted rows, respectively (95% confidence interval: $\pm$ 0.191ms, $\pm$ 0.036ms and $\pm$ 0.013). This shows that the amortized time per data row quickly decreases when increasing the number of transferred data rows. Another observation is that takes roughly $1.3s$ to query, transmit and decrypt 5000 values encrypted with DET and RND. However, the proxy only requires $0.59s$ to perform the decryption. This shows that the decryption only adds a moderate overhead.

We summarize the encrypted processing of analytical application adds little overhead in general and in particular little overhead noticeable by the end user. The processing of encrypted data in the proxy adds comparable overhead as the processing of encrypted data in the cloud. Hence we do not introduce a major performance bottleneck by our architecture. We conclude that encrypted processing using the proxy architecture is a viable alternative in terms of performance from an end user perspective.

5.4 Concurrent requests

The performance measurements in the last sections were based on the assumption that only one end user accesses a protected application and this end user loads the next page only after the last requested page was fully loaded. We now relax this assumption and analyze how our design handles multiple end users that access a page at the same time.

We again use the artificial test application for this test. More specifically, we utilize Prob100, i.e., a table with 100 data rows and 5 columns whereas each values is surrounded by a DET and an RND layer. 100 rows were used, because SAP UI5 only loads 100 rows by default (independently of the rows in the database). Therefore, this can be assumed as a good indicator for many SAP UI5 based applications.

We used the Apache JMeter framework [6] to perform the performance measurements with multiple requests. JMeter is not able to parse JavaScript and the relevant requests are based on JavaScript calls. Therefore, we directly executed the content data call. We configured JMeter to start 55 concurrent threads and to perform a fixed amount of content data requests per second (RPS).

The graph in Fig. 7 depicts the peak load test. It shows that the implementation scales well for 1 to 29 RPS: only a small increase in the content data load time is observable. However, the system measurement breaks at 30 RPS with our hardware configuration. We call this point the system overload point in the following. It depends on the used hardware and it cannot be prevented (if only one machine is used). The reason for overload is that the load time of one or multiple requests exceed the one second time interval. Nevertheless, JMeter again starts the same amount of requests in the next second (30 in our example). At this point, one or multiple requests are still in the queue, because the processing was not finished in the last second. This leads to even more requests that cannot be processed within this second. In theory the request load time would increase without a limit. It is clear that every hardware reaches its system overload point at one specific RPS value.

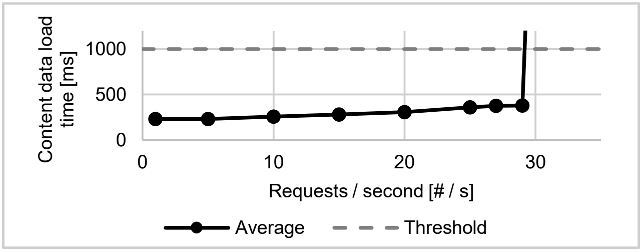

Figure 7: Average load times for multiple content data requests per second

This peak load test also located the bottleneck of the design in our test setup: the CPU of machine with the proxy was fully occupied. This is presumably caused by the decryption operations. In contrast, only around 800 KB/s were transferred from the proxy to the application server and approximately 1 GB memory were used even at 29 RPS. We used high end hardware, but this shows that a far slower network connection and far less memory would have delivered comparable results.

It is important to note that the RPS at the system overload point does not equal the number of users that can use the system simultaneously. The maximal supported user amount corresponds to the average time (in seconds) between requests per user multiplied with the maximal supported RPS value. For instance, around 400 end users could use the system simultaneously (assuming an equal load distribution) if they on average trigger one request every 16 seconds and the hardware infrastructure supports 25 RPS.

Our measurement reveals the specific system overload point in our experimental setup. However, our solution is especially well suited to shift this point, because the proxy is stateless. The proxy can be replicated to as many application servers as required and a load balancer could distribute the requests to satisfy the requested load.

5.5 Message Size

We measure the expansion of messages sizes due to our encryption and post-processing. The application resources, which actually account for the majority of network traffic (see Figure 4), are excluded as they aren't changed. Table 2 shows the messages sizes and the increase factor between plaintext and encrypted processing in data request and response. We see a significant increase in response – around a factor of 4 – already for queries Ch1 and Ch2 where no additional data is transferred as part of the temporary result (before post-processing). Queries Ch3 and Ch4 show larger increases – around a factor of 13 –, since they require to transfer additional data entries for post-processing. Still, the overhead seems manageable as also underpinned by our performance evaluation.

	Content data request			Content data response		
	Plain.	Enc.	Incr. factor	Plain.	Enc.	Incr. factor
Ch1	756	805	1.06	811	2801	3.45
Ch2	758	807	1.06	1091	5323	4.88
Ch3	858	1475	1.72	951	13812	14.52
Ch4	732	781	1.07	1076	12511	11.63

Table 2: Message sizes (in bytes)

5.6 Security

We first evaluate the level of encryption of a database while running our example application. Of course, order-preserving encryption allows better cryptanalysis than deterministic encryption, which allows better cryptanalysis than randomized encryption. Still, all columns remain encrypted – at least at the order-preserving layer – and we can also encrypt numbers for aggregation using additively homomorphic encryption.

Phase 3 in Figure 8 shows the encryption state after executing all features of the SHINE application. We count the number of columns encrypted in probabilistic (RND), deterministic (DET) and order-preserving (OPE) encryption. Due to its analytical nature and the high number of joins in the application we indeed expected a high number of deterministic and order-preserving encryptions. Still, approximately 40% of all columns are either used only for analytic processing or retrieval and thus can remain on randomized encryption. None of the columns used in this application contain personally identifiable data. Therefore, the clients have to decide if this protection level is sufficient or if they need to deploy a stricter policy via our key policy mechanism. Here we achieve a clear security benefit over other solutions that limit the encryption of aggregate values.

Furthermore, Figure 8 shows the evolution of the encryption layers over multiple executions of the application. The phases 1, 2 and 3 correspond to the successive execution of features as encountered during test runs. In phase 1, we only viewed the application pages. The initially triggered content data request still require joins and sorts, which leads to the

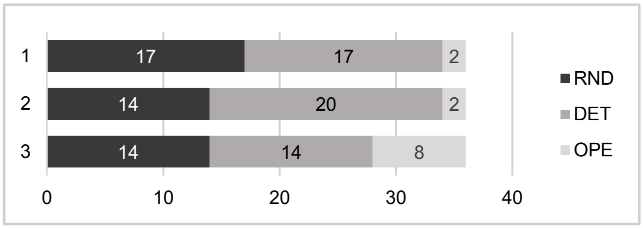

Figure 8: Encryption layers over times 1, 2 and 3

17 columns on deterministic encryption and 2 columns on order-preserving encryption. In phase 2, we used comparison filters on the UI to select certain entries, which let to more DET encrypted columns (20). Finally in phase 3, we used a typical feature of our application: sorting by user-specified columns. Columns were decrypted to the order-preserving layer only after this sorting was explicitly triggered. Thereby, 6 DET columns were decrypted to OPE encryption. This shows that architecture can adjust the encryption to the features of the application that are actually used and executed

Opposed to other approaches – that require the application to be deployed in a trusted environment – our design is well suited to dynamically meet the clients' security requirements, even if the application is deployed at an untrusted cloud provider. Especially our policy checking mechanism provides a great flexibility for the clients. For instance, they can forbid any key publishing in the beginning and evaluate the behavior of the system. The application is still fully functional, but this leads to bad performance, because almost all content data requests lead to post processing. However, the clients can dynamically reduce the restrictions to find their ideal trade-off between security and performance.

6. CONCLUSIONS

In this paper we examine encrypting web applications that use significant number processing on the database. Both, the web application and the database, can be deployed in the cloud. Nevertheless, the client remains in full control over the encryption keys. They even can configure fine-grained key publishing rules to achieve the security guarantees they want. We employ additively homomorphic encryption and a client-server split in order to post-process data after decryption on the proxy. Our architecture introduces a very moderate performance penalty despite complex analytical processing. We significantly enhance encryption to roughly 40% probabilistic encrypted data columns.

We conclude that even encrypted number processing in web-based cloud applications – a typical SaaS offering – is technically feasible. The integration effort on client-side as well as on application provider side is small. The client only needs to install (and configure the policy for) a proxy. The application provider only needs to modify the database driver. For most applications no application specific configuration in the proxy is necessary and hence the application can change without a necessary modification at the proxy. This is a further advantage compared to many existing – including commercial – solutions. Our proxy architecture is stateless and can hence be easily replicated for further performance scaling.

7. ACKNOWLEDGMENTS

This work has received funding from the European Union's Seventh Framework Programme and Horizon 2020 Research and Innovation Programme under grant agreements No. 609611, No. 644579 and No. 644412 of the PRACTICE, ESCUDO-CLOUD and TREDISEC projects.

8. REFERENCES

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

[2] http://www.eweek.com/c/a/Security/Salesforcecom-Acquires-SaaS-Encryption-Provider-Navajo-Systems-331154.

[3] http://www.perspecsys.com/.

[4] http://www.vaultive.com/.

[5] http://scn.sap.com/servlet/JiveServlet/downloadBody/60270-102-1-222286/HANA_SPS08_NEW_SHINE.pdf.

[6] Apache JMeter - apache JMeter^{TM}.

[7] G. Aggarwal, M. Bawa, P. Ganesan, H. Garcia-Molina, K. Kenthapadi, R. Motwani, U. Srivastava, D. Thomas, and Y. Xu. Two can keep a secret: a distributed architecture for secure database services. 2005.

[8] R. Agrawal, J. Kiernan, R. Srikant, and Y. Xu. Order preserving encryption for numeric data. In *Proceedings of the ACM International Conference on Management of Data*, SIGMOD, 2004.

[9] M. Blaze, G. Bleumer, and M. Strauss. Divertible protocols and atomic proxy cryptography. In *Proceedings of the 17th International Conference on Advances in Cryptology*, EUROCRYPT, 1998.

[10] A. Boldyreva, N. Chenette, Y. Lee, and A. O'Neill. Order-preserving symmetric encryption. In *Proceedings of the 28th International Conference on Advances in Cryptology*, EUROCRYPT, 2009.

[11] A. Boldyreva, N. Chenette, and A. O'Neill. Order-preserving encryption revisited: improved security analysis and alternative solutions. In *Proceedings of the 31st International Conference on Advances in Cryptology*, CRYPTO, 2011.

[12] D. Cash, J. Jaeger, S. Jarecki, C. Jutla, H. Krawczyk, M. Rosu, and M. Steiner. Dynamic searchable encryption in very-large databases: Data structures and implementation. In *Proceedings of the 21st Network and Distributed System Security Symposium*, NDSS, 2014.

[13] O. Catrina and F. Kerschbaum. Fostering the uptake of secure multiparty computation in e-commerce. In *Proceedings of the third International Conference on Availability, Reliability and Security*, ARES, pages 693–700, 2008.

[14] V. Ciriani, S. De Capitani Di Vimercati, S. Foresti, S. Jajodia, S. Paraboschi, and P. Samarati. Combining fragmentation and encryption to protect privacy in data storage. *ACM Transactions on Information and System Security*, 13(3), 2010.

[15] R. Curtmola, J. Garay, S. Kamara, and R. Ostrovsky. Searchable symmetric encryption: improved definitions and efficient constructions. *Journal of Computer Security*, 19(5), 2011.

[16] M. Diallo, B. Hore, E. Chang, S. Mehrotra, and N. Venkatasubramanian. Cloudprotect: managing data privacy in cloud applications. In *Proceedings of the 5th IEEE International Conference on Cloud Computing*, CLOUD, 2012.

[17] C. Evans, C. Palmer, and R. Sleevi. Public Key Pinning Extension for HTTP. RFC 7469 (Proposed Standard), Apr. 2015.

[18] S. Foresti. *Preserving privacy in data outsourcing*. Springer Verlag, 2010.

[19] C. Gentry. Fully homomorphic encryption using ideal lattices. In *Proceedings of the Symposium on Theory of Computing*, STOC, 2009.

[20] H. Hacigümüs, B. R. Iyer, C. Li, and S. Mehrotra. Executing sql over encrypted data in the database-service-provider model. In *Proceedings of the ACM International Conference on Management of Data*, SIGMOD, 2002.

[21] F. Hahn and F. Kerschbaum. Searchable encryption with secure and efficient updates. In *Proceedings of the 21st ACM Conference on Computer and Communications Security*, CCS, 2014.

[22] W. He, D. Akhawe, S. Jain, E. Shi, and D. Song. ShadowCrypt: encrypted web applications for everyone. In *Proceedings of the 21st ACM Conference on Computer and Communications Security*, CCS, 2014.

[23] S. Kamara, C. Papamanthou, and T. Roeder. Dynamic searchable symmetric encryption. In *Proceedings of the 19th ACM Conference on Computer and Communications Security*, CCS, 2012.

[24] F. Kerschbaum. Building a privacy-preserving benchmarking enterprise system. *Enterprise Information Systems*, 2(4):421–441, 2008.

[25] F. Kerschbaum. Practical privacy-preserving benchmarking. In *Proceedings of the IFIP International Information Security Conference*, SEC, pages 17–31, 2008.

[26] F. Kerschbaum. A verifiable, centralized, coercion-free reputation system. In *Proceedings of the 8th ACM Workshop on Privacy in the Electronic Society*, WPES, pages 61–70, 2009.

[27] F. Kerschbaum. Frequency-hiding order-preserving encryption. In *Proceedings of the 22nd ACM Conference on Computer and Communications Security*, CCS, 2015.

[28] F. Kerschbaum, D. Dahlmeier, A. Schröpfer, and D. Biswas. On the practical importance of communication complexity for secure multi-party computation protocols. In *Proceedings of the 2009 ACM Symposium on Applied Computing*, SAC, pages 2008–2015, 2009.

[29] F. Kerschbaum, M. Härterich, P. Grofig, M. Kohler, A. Schaad, A. Schröpfer, and W. Tighzert. Optimal re-encryption strategy for joins in encrypted databases. In *Proceedings of the 27th IFIP Conference on Data and Applications Security and Privacy*, DBSEC, 2013.

[30] F. Kerschbaum and N. Oertel. Privacy-preserving pattern matching for anomaly detection in rfid anti-counterfeiting. In *International Workshop on Radio Frequency Identification: Security and Privacy Issues*, RFIDSec, pages 124–137, 2010.

[31] F. Kerschbaum and A. Schröpfer. Optimal average-complexity ideal-security order-preserving encryption. In *Proceedings of the 21st ACM Conference on Computer and Communications Security*, CCS, 2014.

[32] F. Kerschbaum and O. Terzidis. Filtering for private collaborative benchmarking. *Emerging Trends in Information and Communication Security*, pages 409–422, 2006.

[33] J. Köhler and K. Jünemann. Securus: from confidentiality and access requirements to data outsourcing solutions. In *Proceedings of the 8th IFIP International Summer School on Privacy and Identity Management for Emerging Services and Technologies*, 2013.

[34] M. Naveed, S. Kamara, and C. V. Wright. Inference attacks on property-preserving encrypted databases. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pages 644–655. ACM.

[35] P. Paillier. Public-key cryptosystems based on composite degree residuosity classes. In *Proceedings of the 18th International Conference on Advances in Cryptology*, EUROCRYPT, 1999.

[36] M. Pizzo, R. Handl, and M. Zurmuehl. Odata version 4.0 part 1: protocol. Technical report, OASIS, http://docs.oasis-open.org/odata/odata/v4.0/odata-v4.0-part1-protocol.pdf, 2014.

[37] S. C. Pohlig and M. E. Hellman. An improved algorithm for computing logarithms over gf(p) and its cryptographic significance. *IEEE Transactions on Information Theory*, 24(1):106–110, 1978.

[38] R. A. Popa, F. H. Li, and N. Zeldovich. An ideal-security protocol for order-preserving encoding. In *Proceedings of the 34th IEEE Symposium on Security and Privacy*, S&P, 2013.

[39] R. A. Popa, C. M. S. Redfield, N. Zeldovich, and H. Balakrishnan. Cryptdb: protecting confidentiality with encrypted query processing. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles*, SOSP, 2011.

[40] R. A. Popa, E. Stark, J. Helfer, S. Valdez, N. Zeldovich, M. F. Kaashoek, and H. Balakrishnan. Building web applications on top of encrypted data using mylar. In *Proceedings of the 11th Symposium on Networked Systems Design and Implementation*, NSDI, 2014.

[41] D. Roche, D. Apon, S. G. Choi, and A. Yerukhimovich. Sql on structurally-encrypted databases. Technical Report 1106, IACR Cryptology ePrint Archive, 2015.

[42] E. Saleh and C. Meinel. Hpisecure: towards data confidentiality in cloud applications. In *Proceedings of the 13th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, CCGRID, 2013.

[43] D. X. Song, D. Wagner, and A. Perrig. Practical techniques for searches on encrypted data. In *Proceedings of the 21st IEEE Symposium on Security and Privacy*, S&P, 2000.