Three Tales of Disillusion: Benchmarking Property Preserving Encryption Schemes

Frank Pallas[1] and Martin Grambow[2]

[1] Information Systems Engineering Research Group
TU Berlin, Germany
fp@ise.tu-berlin.de,
http://ise.tu-berlin.de
[2] Mobile Cloud Computing Research Group
TU Berlin, Germany
mg@mcc.tu-berlin.de,
http://mcc.tu-berlin.de

Abstract. Property preserving encryption mechanisms have repeatedly been proposed for ensuring confidentiality against cloud providers. However, the performance overhead introduced by such mechanisms has so far only been estimated theoretically or in overly simple settings. In this paper, we present results of first experiments corresponding to realistic scenarios. The results are noteworthy: The Boldyreva scheme for order preserving encryption generates an overhead of approx. 400 % for write operations and a 480-fold overhead for substantial range queries. Partial order preserving encoding introduces a 300 % overhead for inserts and fast-growing query times of approx. 9 seconds for queries over just 30.000 items. With Fully Homomorphic Encryption, in turn, we observed a runtime of 4,5 hours for just one simplified payroll calculation. These results allow for a more deliberate application of respective schemes in real-world business scenarios.

Keywords: property preserving encryption, privacy, confidentiality, cloud computing, benchmarking, order preserving encryption, partial order preserving encoding, fully homomorphic encryption

1 Introduction

Modern information systems are increasingly based on cloud services, ranging from plain virtual instances and hosted databases to higher-order services for data analytics or machine learning. From the business perspective, employing such services provides significant benefits in matters such as availability, elastic scalability, or pace of development.

On the other hand, the model of cloud computing requires strong trust in the employed provider, especially when the data that are to be stored and processed raise – for regulatory reasons or to protect business secrets – confidentiality requirements. While techniques and mechanisms for data in transit encryption are widely established and used in the context of cloud computing to achieve

confidentiality against external eavesdroppers, these leave data fully readable to the provider, posing risks of large-scale data leakage and provider-side data exploitation. Methods for client-side data at rest encryption based on established ciphers like AES, in turn, may be used for ensuring confidentiality against the provider but entail significant drawbacks in matters of implementable functionality: When data are encrypted with established algorithms before being stored in a cloud-hosted database, for example, these data cannot be meaningfully queried or processed on the provider side anymore, rendering the approach of client-side encryption useless for many application scenarios.

In order to resolve this contradiction between confidentiality and functionality, several schemes for so-called property preserving encryption have been proposed in the past, ranging from order-preserving [1] to fully homomorphic encryption [2]. All these mechanisms allow certain functionalities – sorting, querying, or even calculating – to be realized on top of encrypted data and thus eliminate the need for clear text to be known by a cloud provider at all. Given these characteristics, property preserving encryption schemes attracted significant attention in security research as well as in the policy domain [3–6].

Like every security mechanism, however, property preserving encryption comes at a cost in the form of overheads. A reasonable decision on the application of respective schemes in real information systems must therefore be based on a conscious weighing between these overheads and the benefits gained in matters of risk reduction or, respectively, realizable functionality. Even though the fact that property preserving encryption raises significant overheads is often recognized [7], the actual magnitude of these overheads has so far only been estimated on a purely theoretical basis [8–10] or through experiments with overly simplistic and thus unrealistic settings [11–13]. Without sound knowledge on the overhead to be expected in practice, in turn, business decisions on the practical application of property preserving encryption schemes lack solid grounding.

For similar reasons, gathering experiment-based evidence on the performance overhead of security mechanisms that is to be expected in realistic scenarios has established as a valuable source of insights relevant for making well-founded, rational, and reproducible trade-offs in the design and implementation of information systems with reasonable security or privacy requirements. For instance, it has been shown that HBase, one of the most important datastores used in the Big Data context, suffers a throughput drop of up to 47% as soon as its native data in transit encryption is used in a realistic setting [14], while other datastores like Cassandra or Amazon's database as a service (DBaaS) DynamoDB exhibit lower (ca. 20-30 %) or even no measurable impact [15]. Client-side encryption with traditional symmetric schemes on top of Cassandra and HBase, in turn, has been shown to significantly affect achievable throughput in a single-client setting, given the additional computational load induced [16]. With explicit regard to property preserving encryption schemes, first benchmarks applying different schemes on top of NoSQL stores suggest an overhead of up to 500 % for inserts and up to 1.800 % for queries [13], albeit under explicit exclusion of schemes that require additional components on the provider side, for one particular type of

payload, and, most importantly, under lab conditions intentionally eliminating network traffic, clearly motivating further experiments in realistic cloud settings.

In this paper, we thus empirically examine the practical applicability of different property preserving encryption schemes through experiments resembling realistic use cases of cloud computing. In particular, we determine the performance overhead generated by three schemes: the Boldyreva scheme for order-preserving encryption, the POPE (Partial Order Preserving Encoding) scheme of partially order-preserving encoding, and the FHEW (Fastest Homomorphic Encryption in the West) scheme for fully homomorphic encryption. To properly assess these schemes in line with their intended applications, we apply each scheme to a different realistic scenario specifically tailored to the respectively provided functionality. On this basis, our results provide clear indications about the limitations as well as viable use cases of the considered schemes and, thus, a valuable basis for reasoning about their practical applicability.

For this aim, we proceed as follows: In section 2, we present our general experimentation approach. Sections 3, 4, and 5 are then dedicated to our experiments conducted for the three mentioned schemes, respectively. For each of these cases, we briefly introduce the scheme and its provided capabilities, describe our experiment setting, present our benchmarking results, and discuss implications for its practical application in concrete, cloud-based information systems. Section 6 provides some common insights and concludes.

2 Experimentation approach

We conducted experiments for three established property preserving encryption schemes. Each of these experiments represents a realistic application scenario particularly fitted to the respective scheme, thus providing insights about the performance impact to be expected when the employed scheme is well-chosen based on the particular application requirements. The conducted benchmarks therefore had to differ between the experiments to match the specifics of the different encryption schemes.

Beyond this, our experiments follow established design objectives and practices for benchmarking cloud services in general [17] and with particular regard to security overheads [18]. In order to ensure reproducibility and to eliminate random side effects as far as possible, we deployed provider-side components on Amazon EC2 instances particularly fitted for the respective use case (e.g., db.m3.medium for a MySQL database), thus reflecting real application scenarios. For similar reasons, we also deployed the client-side components on Amazon EC2, using a m3.medium instance. Especially in matters of reproducibility and random side-effects, this leads to significant benefits over other approaches, even though we are aware that doing so limits realisticness to a certain extent. Client- and provider-side were always deployed in different regions (eu-central and eu-west) to reflect real use cases and to avoid unintended side effects of co-location.

Instead of implementing encryption schemes by ourselves, we used existing and well-established reference implementations. In every scenario, we wrapped

these implementations within an en- and decryption service which was used from our Java-based benchmarking application via a local socket interface. This model allowed us to use one benchmarking application independently from the language in which the employed libraries are programmed and, importantly, properly resembles potential real-world applications of respective schemes. We ensured that the overhead resulting from this architecture is negligible.[3] The application-specific loads representing real use cases, in turn, were provided in separate scripts to enhance the flexibility of our benchmarking application.[4]

We always aligned our experiments to the particular characteristics of the benchmarked scheme to avoid misaligned bottlenecks. For example, when a scheme raises significant portions of the overhead on the client side, we avoided to actually benchmark the provider side using multiple clients. Insofar, the benchmarking approach pursued herein differs from typical cloud security benchmarking practices where the usual approach is to employ multiple clients to ensure a low workload on the client while stressing the provider side [14, 15]. Based on these core considerations, we can now proceed with our three experiments:

3 Order Preserving Encryption (OPE)

Order preserving encryption (OPE) schemes preserve the order of encrypted values. If a plain value a is smaller than a second value b, the encrypted value $enc(a)$ is also smaller than $enc(b)$. Some examples of deterministic and symmetric OPE schemes are given by Agrawal et al. [19], Boldyreva et al. [8] and Kerschbaum [20].

OPE enables range queries over encrypted data, which is sufficient for several realistic scenarios, and can be easily applied to existing databases or key value stores. To request all datasets in a given range from x_1 to x_2, the client encrypts both values and queries from $enc(x_1)$ to $enc(x_2)$. Because of the order preserving property, it is possible to sort and group values. Moreover, OPE schemes allow joins of tables and counting values but eliminate the possibility to aggregate values (e.g., add, calculate average, etc.) [21]. These properties make OPE schemes particularly suitable for DBaaS applications which do not require computations on the provider side. Our experiment setting therefore resembles the scenario of a small crafting company which outsources their database with employee and customer information, order details and resource requirements to a cloud provider.

In line with our experimentation approach laid out above, we applied the prominently discussed OPE scheme given by Boldyreva et al. [8] to this DBaaS scenario. Expecting only limited load on the database itself, we used a db.m1.medium

[3] Respective experiments indicated a performance overhead of approximately 0.5 ms per call as compared to a native implementation. For the tested encryption schemes, this results in a maximum relative overhead of 2 percent (in most cases, it was significantly below that), which we deemed acceptable.

[4] The whole sourcecode employed for our experiments is available at https://github.com/martingrambow/ppe

RDS instance running a current version (5.6) of MySQL. Due to the functioning of the Boldyreva scheme, no further components or adaptations had to be made on the provider side. On the client side, we used an existing and well-established Python implementation[5] of the Boldyreva OPE scheme as the basis for our local en- and decryption service. To heighten the validity of our results even further, we also implemented a corresponding dummy service without actual encryption to avoid adulterated results. The complete setup of our experiment is illustrated in figure 1.

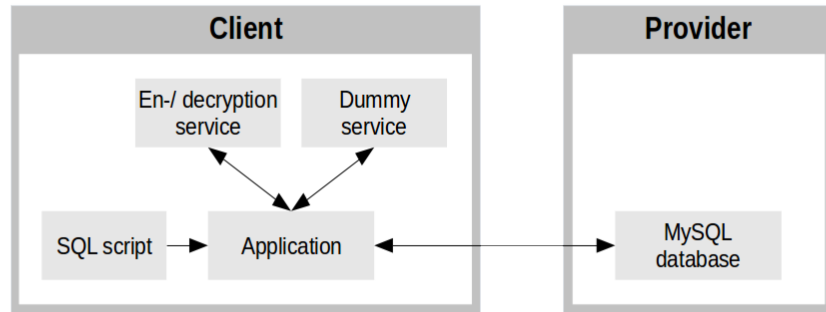

Fig. 1. Setup of OPE benchmarking experiment

As the employed library only supports integer values, we had to convert strings into sufficiently small integers within our encryption service first. For this, we initially translated any single character into a two-digit integer[6] and then concatenated them in groups of five, thus leading to integer values between 0 and 9.999.999.999 (10^{10} - 1). We then fed these integers into the Boldyreva encryption scheme, resulting in larger but similarly ordered integers, which we then splitted back into two-digit integers and converted back into characters. After joining these single characters, we received encrypted strings which are typically longer than the original ones but which still have the same ordering as the original plaintext strings. These OPE-encrypted strings are then used in the respective database operations.

To reflect realistic workloads for this application scenario, the external scripts fed into our application performed the following operations: First, the whole database is deleted and a new database with a representative relational schema comprising customers, orders, etc. (see above) is created. Second, the database is filled with 50.000 data items in total. Finally, seven selections, which request 41.500 data rows in total and also include sorting specifications and joins over multiple tables, are executed. Given the mode of operation of Boldyreva, we

[5] https://github.com/rev112/pyope

[6] For reasons of simplicity, we explicitly decided against more complex character encodings here.

expected no significant additional insights from executing updates and therefore did not benchmark them separately. Database deletion and creation were also not benchmarked at all, while filling and selection were benchmarked separately.

Table 1 summarizes our results. In the filling phase, the overall runtime is mainly determined by transportation (including the insertion process in the database itself) for the baseline setting with only the dummy-service being used. Here, inserting the 50.000 items took around 27 minutes overall. With Boldyreva-based order preserving encryption turned on, the same operations took more than 132 minutes (an increase by approx. 400 %), of which around 106 minutes (or 128 msec per value) fell upon the encryption service, while transportation time (including database inserts) stayed virtually constant. Within the encryption service, in turn, the impact of the above-mentioned conversions between text and integer was negligible: the raw Boldyreva encryption itself took more than 99,8 % of the service's overall runtime.

	Runtime of setup (in ms)			
	Baseline		Boldyreva	
Script / Part	All	Per value	All	Per value
Create	–	–	–	–
Fill ($n = 50.000$)				
Encryption service	12.396	$0,25$	6.382.946	$127,66$
Transportation (incl. DB)	1.581.543	$31,63$	1.569.455	$31,39$
Total	**1.593.939**	**31,88**	**7.952.401**	**159,05**
Select ($n = 41.500$)				
Decryption service	36.197	$0,87$	18.667.941	$449,83$
Transportation (incl. DB)	2.507	$0,06$	2.663	$0,06$
Total	**38.704**	**0,93**	**18.670.604**	**449,89**

Table 1. Experimental results of OPE setup

For the selection part of the experiment, results were even more devastating. While the time required for data transfer and executing the query itself increased only slightly (approx. 6,5 %), the decryption service took more than five hours with Boldyreva encryption as opposed to the baseline of 36 seconds required by the dummy service, thus confirming our assumption that the limiting factor will mainly be on the client side for this experiment.[7] Using order preserving encryption thus led to a more than 480-fold overall runtime in the selection part of the experiment.

On the provider side, consumed disk space increased by more than 75 %, which can, like the slight increase in time for transportation and query execu-

[7] Again, conversions between text and integers within the decryption service were negligible, accounting for less than 0,2 % of the overall service runtime.

tion, be explained by the larger values resulting from the Boldyreva scheme. For settings with many clients, this size increase would presumably also affect achievable throughput on the provider side, but compared to the significant overheads on the client side, this effect will only be decisive in few real-world settings.

Overall, the Boldyreva scheme of order preserving encryption is easily applicable to existing DBaaS offers and allows a limited set of operations, particularly including range queries, to be performed on encrypted data. The overhead of approximately 400 % for database inserts might be deemed acceptable for certain scenarios because of the achieved increase in confidentiality against the cloud provider. However – and even leaving aside further weaknesses of the scheme with regard to ciphertext-only attacks, which shall not be discussed in detail here – the massive decrease in query performance observed in our experiment with many query results will presumably disqualify the Boldyreva scheme of order preserving encryption for many practical use cases. If at all, Boldyreva-based order preserving encryption should thus only be applied in use cases where expectable query results are rather small and can thus be decoded on the client side in reasonable time.

4 Partial Order Preserving Encoding (POPE)

Partial Order Preserving Encoding was introduced by Roche et al. [12]. The general idea of this approach is to encrypt values in a semantically secure way and to store the ordering information in a tree structure. To insert a new item, the client simply has to encrypt the value with an arbitrary scheme and send the ciphertext to a POPE server on the provider side which stores all values in a buffered tree.

Put briefly, this tree consists of nodes which have an unlimited buffer of unsorted encrypted values and a sorted list of child nodes. If a new encrypted value is received, the POPE server inserts it into the buffer of the root node, which grows until a range query occurs. Before processing and answering such a query, the POPE server initiates a communication protocol with a comparison oracle on the client side: For every buffered value, it requests the proper child node and moves the value according to the answer from the oracle. Next, it similarly iterates recursively over all child nodes until all necessary buffers are cleared. Moreover, the POPE server inserts new nodes or rebalances the tree if necessary. For answering a range query, the provider can then identify the two nodes representing the start and the end of the requested range and return all (encrypted) values between them.

On this basis, POPE enables secure range queries over encrypted data which only reveal some information about the ordering of stored values to the provider. This makes POPE more secure than classical order preserving encryption schemes like Boldyreva. On the other hand, POPE requires additional communication between the POPE server on the provider side and the comparison oracle deployed on the client side, which leads to slower range queries when transportation

time increases. Moreover, the oracle has to be continuously reachable, limiting POPE's applicability for use cases with high availability requirements.

Like classical OPE schemes, POPE is particularly targeted to DBaaS scenarios which do not require computations on the provider side. However, data storage must be done within a dedicated POPE server and cannot be delegated to a standard database application. Furthermore, a separate tree structure must be maintained for every dimension that range queries should be executed upon.

To experimentally investigate the practical applicability of POPE, we assumed a database which maintains photo information, whereas the storage of the photos themselves was explicitly out of scope. We implemented the POPE client, server, and comparison oracle based on the python reference implementation given by Roche et al. [12].[8] We furthermore assumed five searchable dimensions providing typical query functionality for photos (year, month, hourOfDay, city, tag). For each of these dimensions, we instantiated a separate POPE server, POPE client, and comparison oracle.

Our benchmarking application used the POPE clients as proxies via local network interfaces. As provided by the reference implementation, inserts as well as queries (including the en- and decryption of values) were then handled by the POPE clients and forwarded to the respective POPE servers, each of which communicated with a corresponding comparison oracle. Update operations were not separately considered here because they are not yet supported by the established POPE library. To avoid unintended side-effects, we refrained from making own extensions. POPE clients and comparison oracles used standard AES-128 encryption in ECB mode as provided by the employed PyCrypto library and shared the secret key through the benchmarking application.[9]

Given the higher processing load we expected on the provider side for this setting, we used a m3.medium instance – the same instance type used on the client side – running all POPE servers. As a baseline for comparing our POPE results to, we used a db.m3.medium RDS instance with default configuration and MySQL 5.6 against which our benchmarking application executed the same workloads. Figure 2 illustrates this setup.

We used different repeatable workloads each of which alternated between a fixed number i of inserts and a fixed number q of range queries until a target number of 30.000 inserted items was reached. Range queries targeted either one or combined two dimensions. We performed experiments with different settings for i and q, using random as well as fixed queries to uncover potential side effects of the POPE-specific approach of query-triggered sorting.

Basically, our results are similar for different sequences of inserts and queries: Inserts into POPE servers always took between 200 and 300 ms and did not change significantly with growing datasets. Range queries to POPE, in turn, got significantly slower with increasing dataset size and had substantial runtime variance for random queries. Basically, however, we observed query times to grow linearly with increasing datasets, resulting in query times of up to 9 seconds for

[8] https://github.com/dsroche/pope

[9] A critical evaluation of the ECB mode in this scenario was not part of our focus.

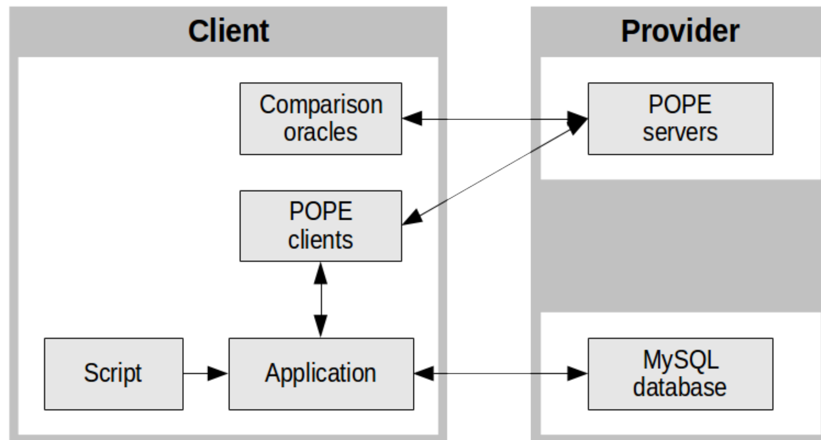

Fig. 2. Setup of POPE benchmarking experiment

a dataset of just 30.000 items with random queries. Linear growth of query times was also confirmed in fixed query experiments, which showed significantly lower variance and query times around 7 seconds for queries on 30.000 items. Given the approach taken by POPE, these runtimes can be attributed to the process of sorting recently added items into the tree structure only before answering queries. For every item, this process requires repeated interaction with the respective comparison oracle on the client side, leading to significant overheads.[10] In contrast, the baseline setting using a standard MySQL database always took around 60 msec (ca. 25% of POPE) per insert and queries were always faster than 100 ms for all workloads.

For our experiment with one random query following 10 inserts, 30.000 inserts overall, and nearly 13 million queried items in total, these differences summed up to a cumulated insert runtime of ca. 120 minutes for POPE as opposed to ca. 30 minutes required with MySQL and without any optimizations (see figure 3). Depending on the use case, this increase might be deemed acceptable for achieving confidentiality against the provider. For queries, however, numbers are rather disastrous: With POPE, the cumulated query runtime was above 130 minutes while plain MySQL served the same queries within less than 140 seconds (see figure 4). Other experiments with different settings for i and q resulted in similar relations.

Even leaving aside further downsides like the need for maintaining a separate tree for every attribute that should be queried, POPE thus already becomes impractical for use cases with several thousands of repeatedly inserted items and with many different queries to be executed over these items. However, POPE

[10] Client-side en- and decryption of values itself, however, took less than 100 μsec per value and can thus be considered rather irrelevant for the overall runtimes observed.

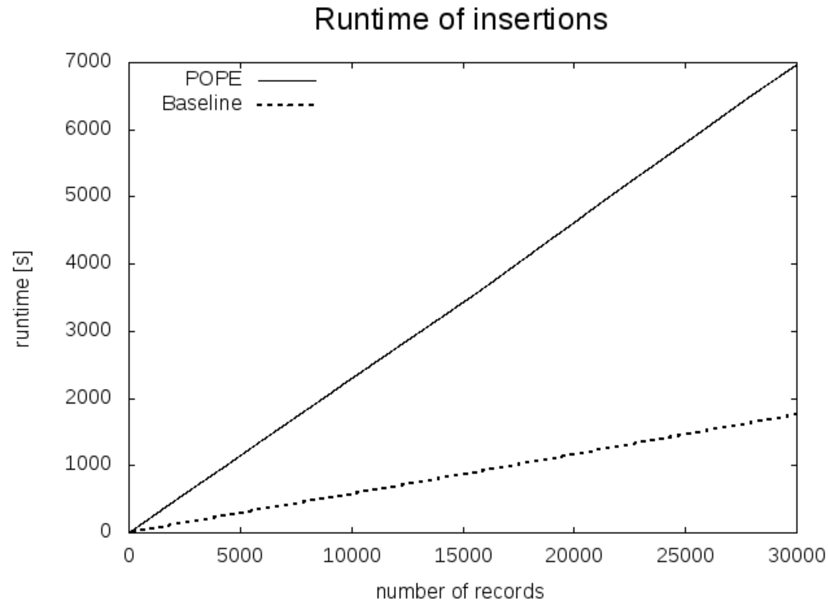

Fig. 3. Total runtime of insertions for POPE

might be suitable for scenarios with a small number of inserts and a larger number of queries over the same range. In such cases, only few items have to be sorted into the POPE trees – leading to fewer interactions with the comparison oracles – before queries can be answered. In practice, POPE should thus only be applied to few, highly specific use cases. As a broadly applicable scheme for achieving confidentiality against cloud providers, however, it largely disqualifies.

5 Fully Homomorphic Encryption (FHE)

As opposed to order-preserving encryption, homomorphic encryption schemes allow mathematical operations to be executed on encrypted data, ensuring that the decrypted result of these operations is similar to the result of respective operations being executed on unencrypted data, e.g., $dec(enc(a) + enc(b)) = a + b$. The supported operations vary among different schemes, whereas the most powerful homomorphic encryption schemes – called "fully homomorphic" – typically support multiplication *and* addition. By choosing binary values as plain messages, addition and multiplication can then be used to simulate logical AND and XOR gates, thus allowing to calculate *any* mathematical function [22].

Obviously, FHE's capability to compute with encrypted information makes it particularly interesting for our problem of ensuring confidentiality against cloud providers while still providing functionality beyond mere raw data storage.

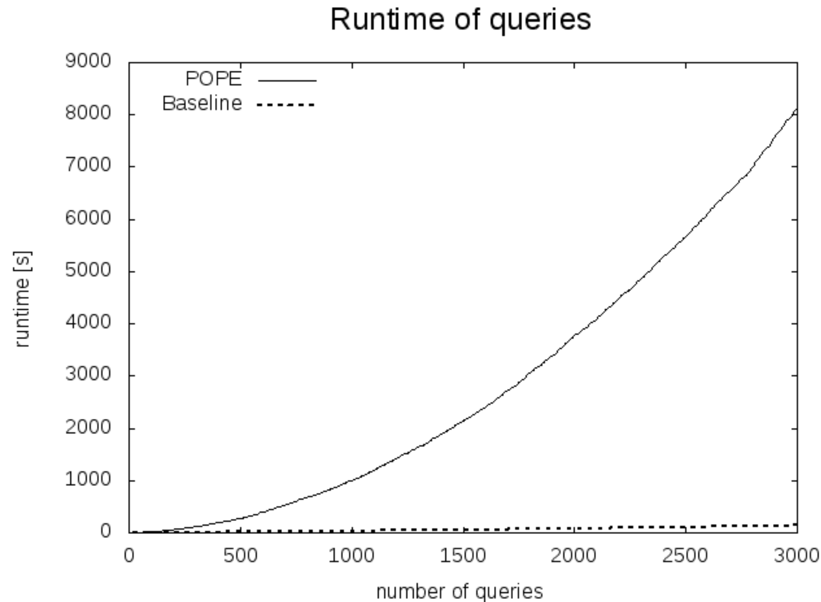

Fig. 4. Total runtime of range queries for POPE

Being able to perform arbitrary operations on encrypted data would thus be a door-opener for applying the concept of cloud computing to domains and scenarios that have so far been abstaining from it because of particularly strict confidentiality requirements. Consequently, FHE schemes have been suggested for application domains like e-health [10] or smart metering [23, 24].

We therefore explicitly wanted to benchmark the performance of real implementations of the "arbitrary operations" functionality of FHE based on simulated logical gates. Being aware that the performance overhead to expect would be substantial, we strove for a scenario with minimal viable complexity for our experiment and chose a simplified payroll calculation scheme that should be executed within the cloud, resembling a cloud-hosted and FHE-enabled salary calculation service. For this purpose, we built a basic service application (called EFHE) based on the well-established FHEW library.[11] FHEW implements the FHE scheme by Ducas at al. [25] which is broadly known for its superior performance as compared to other FHE schemes like the BGV scheme [26] implemented in HElib [11].

Our service application consists of a client and a server component, which encapsulate fundamental FHEW operations and make the scheme easily usable: The client encrypts and decrypts given integer values. The server part, in turn, simulates multiple combined gates and on this basis performs basic

[11] https://github.com/lducas/FHEW

arithmetic functions over the encrypted values, thus implementing FHE's core concept for executing arbitrary mathematical functions. Both components communicate with the core FHEW functionality via local network services and offer a simple socket interface which enables component-specific interactions. Given the comparably simple benchmarks that we used in this case, we abstained from implementing a dedicated benchmarking application but rather used direct parameterized executions of the EFHE client here. Again, both components were deployed on separate m3.medium instances. Figure 5 illustrates this setup.

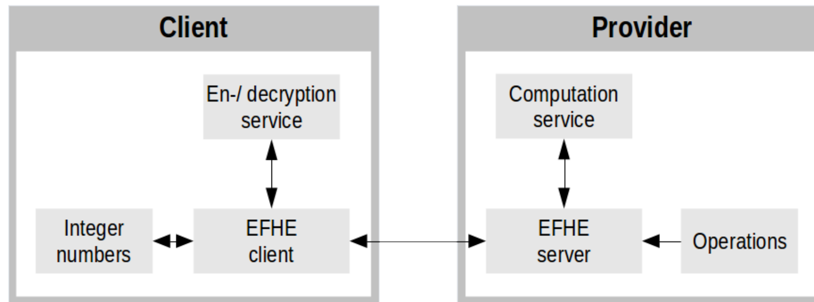

Fig. 5. Setup of EFHE benchmarking experiment

In an initial, rather explorative experiment, we performed additions, subtractions, multiplications and divisions for integers of 3 different dimensions (8, 16, and 24 bit). Next, we applied a simplified payroll calculation which comprises 3 additions, 1 subtraction, 4 multiplications and 4 divisions. Even this basic set of operations cannot be completely realized through plain FHE and therefore requires the above-mentioned approach of function implementation through simulated logical gates.

Table 2 shows the results of our first experiment. We performed each operation 10 times and converted each total runtime to the average time per operation. Because of the linear complexity of addition and subtraction circuits, respective runtime grows with the number of bits per integer value. In our experiment, addition and subtraction of 24-bit integers each took about 65 seconds on the employed m3.medium instance. The runtime of multiplications and divisions, in turn, grows exponentially with increasing bit length. We measured a runtime of about 18 minutes for the multiplication of two 24-bit integers, whereas respective divisions took almost one hour. Moreover, we performed 5 simplified payroll accounting calculations which comprise 12 operations each. On average, any such payroll calculation ran almost $4,5$ hours, which corresponds to a price of 0,33 USD at the time of writing just for the instance performing the computation on the provider side. The additional effort generated by local network services and

the conducting applications consumed less than $0,6\%$ of the total runtime and is thus negligible.

Besides computation, massive overheads also arose in terms of storage space and transfer volume. In the default configuration, FHEW consumes 2.004 bytes (16.032 bits) of disk space per bit of raw data or, respectively, 47 kByte per 24 bit integer value. This obviously also disqualifies it for applications with reasonable volumes of data to be processed.

	Runtime per op		
Bits	8	16	24
Addition	$18.621ms$ ($\sim 18s$)	$38.832ms$ ($\sim 39s$)	$66.096ms$ ($\sim 66s$)
Subtraction	$18.656ms$ ($\sim 19s$)	$38.775ms$ ($\sim 39s$)	$65.222ms$ ($\sim 65s$)
Multiplication	$87.942ms$ ($\sim 1,5min$)	$405.838ms$ ($\sim 6,8min$)	$1.055.858ms$ ($\sim 17,6min$)
Division	$316.534ms$ ($\sim 5min$)	$1.311.147ms$ ($\sim 22min$)	$3.299.631ms$ ($\sim 55min$)

Table 2. Runtime of basic arithmetic operations of n-bit integers

As it already becomes clear from these numbers, implementing arbitrary functions through simulated logical gates on top of FHE should currently only be considered a theoretical concept. Given the massive overhead generated in terms of time and, notably, costs to be borne, we do not even see single realistic use cases. As a basis for realizing cloud services with arbitrary functionality while still providing confidentiality against the provider, the approach of simulated logical gates on top of FHE is, by all means, far from being a realistic option. This might, of course, be different for use cases that can be implemented through plain FHE without the need for simulated logical gates or where schemes for "somewhat" homomorphic encryption are sufficient [10] in matters of functionality *and* confidentiality. Even in these cases, any consideration of practical applications should nonetheless be accompanied by sustainable benchmarks of arising overheads to avoid futile development paths as early as possible.

6 Conclusion

Property preserving encryption schemes have repeatedly been suggested for resolving the conflict between realizing functionality and achieving confidentiality against the provider in cloud-based information systems. In this paper, we

experimentally evaluated three well-established property preserving encryption schemes with regard to the expectable overhead when applied in realistic scenarios.

The covered schemes achieve the goal of functionality preservation through different approaches: The Boldyreva scheme uses a highly specific encryption algorithm that has order preservation as a core characteristic and thus allows for comparisons, range queries, etc. even when used on top of a standard database or a DBaaS service. The POPE scheme, in contrast, is used together with standard encryption algorithms like AES and achieves functionality through a highly specific, tree-based approach for data handling and a more complex interplay between different components on the client- and the provider-side.

Based on our experiments, we see only limited reasonable applications in concrete business information systems for both: The Boldyreva scheme produces acceptable overheads for inserts but should – if at all – only be applied in scenarios with comparably few query results, given the overhead arising from costly client-side decryption. The more complex POPE scheme, in turn, exhibits acceptable overheads for scenarios with few inserts but performs poorly as soon as new values must repeatedly be sorted into its data trees. Furthermore, POPE requires significant implementation efforts on the provider side, limiting reasonable scenarios for its practical application even further. In select use cases, however, both might represent a rational choice for bringing together confidentiality against the provider and functionality.

Finally, the approach of realizing arbitrary functions through simulated logical gates based on the FHEW scheme for fully homomorphic encryption, which significantly differs from the aforementioned ones, turned out to be technically possible but to raise tremendous overheads. Its application will therefore hardly be justifiable in any realistic use case.

Besides providing valuable insights for rational weighing decisions on the application of covered schemes in real-world information systems, our experiments also highlight the importance of empirical experiments resembling realistic application scenarios in general. Applying a similar approach to other property preserving encryption schemes beyond those covered herein would thus presumably help to better understand reasonable fields of application for these schemes, too. In any case, property preserving encryption schemes should not be applied blindly without experimentally evaluating their suitability for a given business scenario to prevent later disillusions.

7 Acknowledgments

Parts of the work presented herein have been supported by the European Commission through the Horizon 2020 Research and Innovation program under contract 731945 (DITAS project)

References

1. Moghadam, S.S., Gavint, G., Darmonti, J.: A secure order-preserving indexing scheme for outsourced data. In: IEEE International Carnahan Conference on Security Technology (ICCST 2016), IEEE (2016) 1–7
2. Gentry, C., Halevi, S.: Implementing Gentrys fully-homomorphic encryption scheme. In: Annual International Conference on the Theory and Applications of Cryptographic Techniques, Springer (2011) 129–148
3. Spindler, G., Schmechel, P.: Personal Data and Encryption in the European General Data Protection Regulation. JIPITEC **7**(2) (2016) 163–177
4. Bonfanti, M.E.: Lets Go for New or Emerging Security Technologies!... What About Their Impact on Individuals and the Society? Democrazia e Sicurezza-Democracy and Security Review (2) (2017)
5. Schulz, W., Hoboken, J.v.: Human rights and encryption. UNESCO (2016)
6. Acquisto, G.D., Domingo-Ferrer, J., Kikiras, P., Torra, V., de Montjoye, Y.A., Bourka, A.: Privacy by design in big data – an overview of privacy enhancing technologies in the era of big data analytics (2015) ENISA.
7. Danezis, G., Domingo-Ferrer, J., Hansen, M., Hoepman, J.H., Metayer, D.L., Tirtea, R., Schiffner, S.: Privacy and data protection by design – from policy to engineering (2014) ENISA.
8. Boldyreva, A., Chenette, N., Lee, Y., O'Neill, A.: Order-Preserving Symmetric Encryption. In: Proceedings of the 28th Annual International Conference on Advances in Cryptology: The Theory and Applications of Cryptographic Techniques, Springer (2009) 224–241
9. Damgård, I., Jurik, M.: A Generalisation, a Simplification and some Applications of Pailliers Probabilistic Public-Key System. In: Proceedings of the 4th International Workshop on Practice and Theory in Public Key Cryptography: Public Key Cryptography, Springer (2001) 119–136
10. Naehrig, M., Lauter, K., Vaikuntanathan, V.: Can homomorphic encryption be practical? In: Proceedings of the 3rd ACM Workshop on Cloud Computing Security, New York, NY, USA, ACM (2011) 113–124
11. Halevi, S., Shoup, V.: Bootstrapping for helib. In: Annual International Conference on the Theory and Applications of Cryptographic Techniques, Springer (2015) 641–670
12. Roche, D.S., Apon, D., Choi, S.G., Yerukhimovich, A.: POPE: Partial order preserving encoding. In: Proceedings of the 2016 SIGSAC Conference on Computer and Communications Security, ACM (2016) 1131–1142
13. Waage, T., Wiese, L.: Property preserving encryption in nosql wide column stores. In: OTM Confederated International Conference "On the Move to Meaningful Internet Systems". (2017) 3–21
14. Pallas, F., Günther, J., Bermbach, D.: Pick your choice in hbase: Security or performance. In: IEEE International Conference on Big Data. (2016) 548–554
15. Müller, S., Bermbach, D., Tai, S., Pallas, F.: Benchmarking the performance impact of transport layer security in cloud database systems. In: IEEE International Conference on Cloud Engineering (IC2E). (2014) 27–36
16. Waage, T., Wiese, L.: Benchmarking encrypted data storage in hbase and cassandra with ycsb. In: International Symposium on Foundations and Practice of Security. (2014) 311–325
17. Bermbach, D., Wittern, E., Tai, S.: Cloud Service Benchmarking: Measuring Quality of Cloud Services from a Client Perspective. Springer (2017)

18. Pallas, F., Bermbach, D., Müller, S., Tai, S.: Evidence-based Security Configurations for Cloud Datastores. In: Proceedings of the Symposium on Applied Computing. SAC '17, ACM (2017) 424–430
19. Agrawal, R., Kiernan, J., Srikant, R., Xu, Y.: Order Preserving Encryption for Numeric Data. In: Proceedings of the 2004 ACM SIGMOD International Conference on Management of Data, ACM (2004) 563–574
20. Kerschbaum, F.: Frequency-hiding order-preserving encryption. In: Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security, ACM (2015) 656–667
21. Malkin, T., Teranishi, I., Yung, M.: Order-Preserving Encryption Secure Beyond One-Wayness. (2013)
22. Armknecht, F., Boyd, C., Carr, C., Gjøsteen, K., Jäschke, A., Reuter, C.A., Strand, M.: A Guide to Fully Homomorphic Encryption. IACR Cryptology ePrint Archive (2015) 1192
23. Tonyali, S., Saputro, N., Akkaya, K.: Assessing the feasibility of fully homomorphic encryption for smart grid ami networks. In: International Conference on Ubiquitous and Future Networks. (2015) 591–596
24. Deng, P., Yang, L.: A secure and privacy-preserving communication scheme for advanced metering infrastructure. In: IEEE PES Innovative Smart Grid Technologies (ISGT). (2012) 1–5
25. Ducas, L., Micciancio, D.: FHEW: Bootstrapping Homomorphic Encryption in Less Than a Second. In: Annual International Conference on the Theory and Applications of Cryptographic Techniques (Eurocrypt), Springer (2015) 617–640
26. Brakerski, Z., Gentry, C., Vaikuntanathan, V.: (Leveled) Fully Homomorphic Encryption Without Bootstrapping. In: Proceedings of the 3rd Innovations in Theoretical Computer Science Conference, ACM (2012) 309–325