Breaking Encrypted Databases: Generic Attacks on Range Queries

Marie-Sarah Lacharité
Royal Holloway, University of London and NCC Group

August 2019

Briefing Abstract

Security researchers and practitioners have proposed many techniques for securely storing and querying outsourced data. In this report, we first provide an overview of common building blocks and the latest commercial and academic solutions, focusing on those that support range queries (e.g., selecting all records where the age attribute is between 18 and 65). These techniques are tailored to specific threat models. For example, if the database server is trusted but not the network, connections can be encrypted with TLS. If the database server is trusted but there is a risk of disk theft, full-disk encryption or page-level encryption of database files and logs (e.g., Transparent Data Encryption) can be enabled. If the database server is not trusted at all, a system that encrypts all data before uploading it (e.g., via a CipherCloud gateway or CryptDB proxy server) could be employed.

All of these solutions, however, leak some information when a query is processed – like the set of records matching the query, or the size of this set. This information leaks even to an observer who does not have any cryptographic keys. The source of the leakage can vary; it could be network traffic, observed memory accesses, or database logs recovered by forensic analysis. This leakage can be exploited by an attacker to break the encryption and recover values in the database. These attacks are entirely generic and do not depend on the database implementation. They have connections to graph theory, Golomb rulers, and machine learning. We also discuss proposed countermeasures and offer guidelines that practitioners can use when assessing the security claims of the latest and greatest database encryption solutions.

Context and Motivation

Many organizations store our sensitive information, such as medical history, financial records, or personal correspondence. However, they are often breached: billions of these records have leaked in 2019 alone [35]. Data breaches have become so common that there are now services, like have i been pwned [16] that will check if your email address or user ID has appeared in a leak.

Security and privacy researchers have investigated ways to make these leaks less harmful while keeping whatever functionality is needed on the stored data. For instance, is there a way to encrypt medical records so that policy makers can still efficiently query the data to look for trends and make decisions, but someone who runs off with the hard drive gets only gets useless, encrypted data? Could transaction logs be encrypted in such a way that a curious data center employee does not get access to the data even though they have system administrator privileges on the database server?

It is important to analyze proposed cryptographic techniques thoroughly; believing that data is secure when it is not could be catastrophic. This report is about how leakage from encrypted databases can still be exploited by an adversary.

The work presented in this report was carried out jointly by the author and Paul Grubbs, Brice Minaud, and Kenny Paterson, at Royal Holloway, University of London. The attack on access pattern was published in the proceedings of IEEE Security and Privacy 2019 [12] and also available on the IACR's Cryptology ePrint Archive [13]. The attack on volume leakage was published in the proceedings of the ACM SIGSAC Communications and Computer Security conference 2018 [14] and also available on ePrint [15].

Introduction: Side-Channel Attacks

As an introduction to encrypted database attacks, consider the (perhaps more familiar) notion of side channel attacks. Side channels are classes of vulnerabilities that arise from the implementations of algorithms or protocols, not the designs of the algorithms or protocols themselves. For example, if two people are having a conversation, a side channel could be anything other than the spoken words, like tone of voice, volume of speech, who the two speakers are, the duration of the conversation, when the conversation happens, or the speakers' body language. When communication occurs between machines instead of people, there are other side channels like power consumption, electromagnetic radiation, memory usage, cache usage, CPU usage, and so on. What is leaked through these side channels can reveal information about the message contents, or, if cryptography is involved, about the secret key.

For example, there have been side channel attacks on

- keystroke recovery from timing information in SSH interactive mode [34]. In this mode, each individual keystroke is sent in its own IP packet, so a network observer learns the length of what is typed. Further, the inter-stroke timing between pairs of letters can be used to build a statistical model that allows dramatically reducing the possible letters, speeding up dictionary attacks on passwords.
- video stream identification based on traffic burst analysis [33]. The MPEG-DASH streaming video standard packages frames into bursts of packets whose timing and size depend on the video content. This leakage can be exploited by an attacker to identify which video, from a known set, a target is watching. The adversary could directly observe the encrypted traffic or, e.g., obtain it by measuring contention on a shared network link.
- using error messages about whether an RSA ciphertext correctly decrypts or not to decrypt it [2]. In this classic cryptographic attack, the attacker uses an "oracle" that takes as input any RSA ciphertext, and reveals whether the underlying plaintext is correctly formatted or not. By crafting a series of modified ciphertexts, the attacker can decrypt a target ciphertext. Whether there was a formatting error could leak directly as an error message or through timing channels.

What we think of as side channel attacks are indeed similar to attacks on encrypted databases: they are about exploiting leakage (unintentional information) from communication between two parties. In encrypted database attacks, the two parties are a client, who queries its data to retrieve particular records, or rows, based on their values, and a database server, who hosts the data. This report is about new kinds of side channel attacks that can break database encryption. They exploit leakage from numeric range queries (like ”which records have value between 1 and 5”) between the client and database server. These attacks exploit *access pattern leakage* (which rows in the database matched the query) and *volume leakage* (how many rows matched it). The attacks use this leakage to break encryption – even if everything in the database is encrypted, an adversary can figure out the plaintext if these side channels exist.

Existing Approaches

Consider a simple client-server architecture where the server is storing a database. The first column of the database table (“ID”) is the primary key, unique for all rows. For simplicity, assume the “ID” values are sequential. The second column is “Value”, a number between 1 and N, which is the attribute the client is going to select rows on. For example, in a medical database, “ID” could be a patient identifier number, and “Value” could be the patient’s age. The table could have more columns, but we assume the client always queries the data based on this one. The type of queries the client makes is range queries; it always retrieves records based on whether the value falls in some numeric range, say 1 through 5, written as [1,5]. In SQL, this would be a query of the form

```
SELECT * FROM table WHERE Value BETWEEN X AND Y
```

The particular query language or database engine is not important, however, as long as (i) the client makes range queries over some finite set of values, assumed to be the integers $\{1,\ldots,N\}$, and (ii) the database server correctly returns the IDs of records whose values fall in that range.

Protecting Data in Transit

Securing data is all about understanding the threat model. Historically, protecting queries and results from a network eavesdropper was the first step. The client and server can protect their communications by encrypting them using the Transport Layer Security (TLS) standard [18]. TLS allows the client and server to authenticate each other and negotiate a session key to encrypt the queries and responses, so anyone observing the packets does not know what the query was or which records matched it. Encrypting data on the wire does not protect against everything, however.

Protecting Data at Rest

As people began to outsource their data to big data centers, they had to update their threat models to account for the risk of disk theft. Encryption at the filesystem level or column level addresses this issue. A typical query process might have the following steps:

1. The client uses TLS to encrypt its query and sends it to the server.

2. The server decrypts it, consults its search index, and fetches the relevant encrypted pages from disk.
3. It decrypts them in memory, processes the results, re-encrypts them with TLS, and sends them back to the client.
4. The client decrypts them.

The data on disk stays encrypted the entire time, so disk theft is mitigated.

Many major database vendors offer some variant of this type of encryption, usually called "Transparent Data Encryption" (TDE) or "Native Encryption". These solutions usually do not noticeably affect performance; the server can still index the plaintext data, so range queries can be answered efficiently. Some of these solutions also offer more granular field-level encryption, like format-preserving encryption or tokenization, but the data encryption key is usually still managed by the server.

Some examples include Transparent Data Encryption (TDE) offered by Microsoft for SQL Server [27], which encrypts data and transactions logs at the page level; Oracle Database's Transparent Data Encryption [31], which allows column or tablespace encryption of data and encryption of undo and redo logs; IBM's Native Encryption for DB2 [17], which offers tablespace encryption of data and encryption of transaction logs; and MongoDB's Encrypted Storage Engine [29], which encrypts all data files, but not logs.

When the database server manages the data encryption key, it is usually stored in a separate keystore, i.e., not on the same disk as the data. However, it is often accessible to curious database administrators, system administrators, or any user who gains such permissions. To prevent a full database server system compromise from revealing the data, it needs to be encrypted by the client, or via a proxy before it even gets to the server.

Protecting Data in Use

How to query encrypted data without letting the database server see raw, unencrypted data is an interesting and challenging problem. In industry, these solutions are usually called client-side field-level encryption: instead of being in a keystore attached to the database server, the client completely controls the keys.

Such solutions are offered by Microsoft's Always Encrypted [26], which offers field-level encryption in a client-side driver; MongoDB's Field Level Encryption in the upcoming 4.2 release [28]; and companies like CipherCloud [8] or solutions like CryptDB [32] that act as proxies between a client and database server.

Usually, these client-side encryption solutions offer only two basic types of encryption: deterministic and randomized. A third type, called Order-Preserving Encryption (OPE) is also sometimes used. One leaks repetition and makes range queries possible but inefficient, one is secure but does not support ranges, and one leaks order and makes range queries as efficient as they are on plaintext.

Deterministic encryption (DE) is probably the most widely used way to support queries on encrypted data, but it does not offer much functionality – only exact matches. With deterministic encryption, any repeated plaintext values show up as repeated ciphertext values. For example, each row that has a value of "1" will have the exact same ciphertext. This enables range queries: if the client wants to retrieve all records with values between 1 and 3 (i.e., in [1,3]), it could simply request all records whose values are in the set of encryptions {DE(1), DE(2), or DE(3)}. Although range queries are possible with deterministic encryption, the fact remains that any repetitions in the plaintext will show up in the ciphertext. Leaking the equality of val-

ID	Value
1	3
2	1
3	15
4	41
5	1

(a) Plaintext

ID	Value
1	`0x18fa83`
2	`0x5449a1`
3	`0x8b7630`
4	`0x10cae8`
5	`0x5449a1`

(b) DE

ID	Value
1	`0x5239fb`
2	`0x8e9d98`
3	`0x5a9f2e`
4	`0x4ff8e1`
5	`0xe89cfb`

(c) RE

ID	Value
1	182
2	84
3	2307
4	8932
5	84

(d) OPE

Above is an example illustrating the differences between the same values encrypted with Deterministic Encryption (DE), Randomized Encryption (RE), and Order-Preserving Encryption (OPE).

ues can be exploited when combined with information about the distribution of values. These attacks were evaluated on medical datasets by Naveed, Kamara, and Wright [30], and many others after.

With randomized encryption (RE), the server is unable to index the data or group values. Using randomized encryption, all encryptions of "1" are different, which offers better security, but the server would have no way to select all of the records with value "1".

Order-preserving encryption (OPE) [1, 5] is another method of encrypting numeric data, which does exactly what it sounds like: if $x < y$, then $\text{OPE}(x) < \text{OPE}(y)$. This type of encryption allows range queries and sorting over the ciphertexts. If the client encrypts values with OPE before sending them to the server, the server can still index the data just as if it was unencrypted, but it does not learn the exact values. When the client wants to perform a range query, all it has to do is encrypt the endpoints of that range.

Unfortunately, even an "ideal" OPE scheme (one that behaves like a random order-preserving function) must leak strictly more than order. In particular, about *half* of the plaintext bits leak [4] – which is why any efficient OPE scheme cannot offer much security. Further, some OPE schemes are also deterministic, leaking repeated values, and susceptible to frequency analysis attacks [11].

This property of OPE schemes motivated new types of schemes that sacrifice less security while still allowing range queries and sorting. These include techniques like Order-Revealing Encryption (ORE) [25, 7], which is a generalization of Order-Preserving Encryption. With OPE, it is possible to just look at two ciphertexts to compare them, but with ORE, it is necessary to compute a function over each pair to know which one is smaller. Some schemes do more complicated things, like build a search index that the server can traverse by itself, "destroying" nodes along it as it goes, and requiring the client to "re-generate" them after each query [3]. Although these schemes leak less than OPE, they still have some leakage.

A modern encrypted database might combine all of these types of encryption: TLS for the queries and responses, server-side disk encryption, and client-side field-level encryption. Even with all of these layers of security, there might still be some exploitable leakage in side channels. For instance, the server needs to know which records matched a range query in order to return the correct results – the identifiers of the records that matched the queried range (access pattern) could leak. Also, despite the client-server traffic being encrypted, its length is not hidden – how many records matched the queried range (volume) could leak.

Regardless of the type of encryption that is used to secure an encrypted database, practically all schemes can leak which records matched a query and how many matched it. How exactly these properties leak varies – it could be from an adversary man-in-the-middling connections to the database, getting access to undo/redo logs or query profiling logs, or simply observing traffic volume [10].

Exploiting Access Pattern Leakage

Suppose the client formulates a range query, sends it to the server, the server finds the matching records, and sends back the matching record identifiers to the client. Somehow, without learning the query endpoints, the adversary learns *which* records matched the query. This attack illustrates how such an adversary can fully reconstruct the values of all records in the database.

A closer look at ranges. Range queries have some important properties. Suppose the adversary observes two queries and some record identifiers are returned by each query. Since ranges are continuous intervals, any records matched by both queries must have values that are between the values of records that matched only the first query, and the values of records that matched only the second query.

Suppose there is a database with 10 records and their IDs are sequential from 1 to 10. If the adversary observes the access pattern leakage { 2,3,5,10 } and { 1,2,4,5,8 }, then it can deduce that records 3 and 10 have values less than records 2 and 5, which have values less than records 1, 4, and 8. As the adversary observes more queries, it can perform set intersections on the sets of matching record identifiers to group together the records and sort them by value, but there exists a data structure that facilitates this process.

PQ trees. PQ trees were discovered nearly 45 years ago [6]. The idea of using a PQ tree to order records in a database is not new [20], but it was never used to reconstruct values in a database. A PQ tree has two types of internal (i.e., non-leaf) nodes – P nodes and Q nodes. PQ trees encode a set of permutations on some group of elements, which are the leaves of the tree. The children of a P node can be reordered in any way: if a P node has k children, then there are $k!$ possible orderings of its children. The children of a Q node, on the other hand, can only be reflected; there are two possible orderings of its children.

PQ trees allow keeping track of the possible orderings of the records. Once the records are in as many groups as there are possible values, it is easy to work out the exact value of every record in the database. At the start, there is a single P node with as many children as there are records; all orderings are possible. After seeing a query, the nodes of the PQ tree must be re-arranged so that the records that it matched are next to each other in all orderings. There is a well defined, but slightly tedious procedure to carry out this re-arrangement. Eventually, after *sufficiently many* queries, the PQ tree will have a Q node at the root, and all records will be its direct children or in groups as its grandchildren. At this point, the adversary has figured out the order of all the records, from smallest to largest (or largest to smallest). The Q node's children are groups of records with the same value. If two records have the same value, then they must match exactly the same set of queries; there can be no range query that matches one but not the other. The first group is records with value 1, the second is records with value 2, and so on – the adversary can assign a value to each record in the database.

"Sufficiently many"? The success of this attack relies on the adversary seeing "sufficiently many" queries. Clearly, if it sees leakage from the same query, repeated many times, the attack will not succeed. To analyse how many queries are necessary, we must assume that the queries the client makes are sampled from some fixed distribution, like the uniform distribution (i.e., where the client is equally likely to make any possible range $[x,y]$). Then, some powerful results from statistical learning theory (specifically, the "ϵ-net theorem") apply and we may conclude that with high probability, after some number of queries, the attack will succeed.

If there are N possible values – if the goal is to get a PQ tree consisting of one Q node with N children – then the number of required queries is about $N \log N$. For example, if there are 100 possible values, then the attack would succeed after seeing about 500 queries.

A crucial assumption to the success of this attack is that there must be at least one record with every value. However, it is possible to adapt the analysis to count how many queries are required if the goal is merely to get sorted groups of records whose values are "close enough". For this kind of *approximate reconstruction*, the number of required queries does not depend on the number of possible values N; it depends only on how close the value in each group should be, relative to N. For example, to group together records whose values are at most 5% apart, only 60 or so queries are required, regardless of whether N is 50 or 500 or 5000.

This attack illustrated how access pattern leakage is sufficient to order all records and group them by value, from which it is possible to deduce the value of every single row in the database, bypassing the encryption. It uses a PQ tree to encode the leakage along the way. Even without the leakage from sufficiently many queries to exactly determine the values, it is possible to approximately recover values.

Access pattern leakage is significant. In many threat models, it might not leak to the adversary. Volume leakage, on the other hand, can be observed more easily: even a network adversary could have access to this side channel by monitoring network traffic. This is not the only way in which the number of records matching each query can leak, though – server-side log files could also reveal it. There are settings where an adversary cannot see access pattern leakage, but it can see volume leakage.

Exploiting Volume Leakage

Suppose that the adversary has learned how many records match every possible range query; it has the entire set of query volumes. It does not know which query corresponds to which volume, it sees only the volumes:

$$\{\, \mathrm{vol}([1,1]), \mathrm{vol}([1,2]), \ldots, \mathrm{vol}([1,N]), \mathrm{vol}([2,2]), \ldots, \mathrm{vol}([N\text{-}1,N]), \mathrm{vol}([N,N]) \,\}.$$

It can use these volumes to determine exactly how many records there are with each value in the database.

If the query distribution can be modelled, it is possible to figure out how many queries a network adversary would need to observe before it gets all volumes. Suppose all ranges are equally likely. Then, by applying a coupon collector bound, we may conclude that the expected number of observed queries until all volumes have been observed is about $N^2 \log N$.

Let R be the maximum observed volume, which must be the volume of the range 1 through N, which is also the number of rows in the database.

The idea of this attack is to identify *elementary volumes* among the set of all volumes. Elementary volumes are the volumes of *elementary ranges*, which are the ranges [1,1], [1,2], [1,3], and so on up to $[1,N]$. If the adversary can identify which volumes correspond to these queries, then it can deduce exactly how many records have each value: by subtracting the volume of the *k*-th elementary query from the volume of the (*k*+1)-st elementary query, it will get the number of records with value *k*+1:

$$\text{vol}([1,k+1]) - \text{vol}([1,k]) = \text{vol}([k,k]).$$

The goal of this attack is therefore to identify the set of elementary volumes. Elementary ranges and volumes have the following special properties:

1. Every elementary range has a *complementary* range such that the sum of their volumes is exactly *R*, the total number of rows in the database.
2. Any range is either an elementary range (if its left endpoint is 1) or a difference of two elementary ranges (otherwise).
3. The set difference of any two elementary ranges is also a range; the absolute difference of any two elementary volumes is also an observed volume.

Using these properties, the algorithm builds a graph to identify these elementary volumes. The graph is constructed as follows: its nodes are the set of observed volumes and there is an edge between two nodes if the absolute difference of their volumes was also an observed volume. Then, in this graph, the elementary volumes will correspond to nodes that form a *clique* – a subset of nodes that are all directly connected to one another. This is because of property 3 above: the difference of any two elementary ranges is also a range, so there must be edges between every pair of elementary volumes. Further, the volumes generated by that clique – of the nodes themselves and the differences corresponding to the edges – generate the set of all observed volumes. This is because of property 2 above – every range is either an elementary range or the difference of two elementary ranges.

Finding a clique in a graph is, in general, a hard problem, but some tricks allow finding a clique efficiently. This reconstruction algorithm has two phases: pre-processing and traditional clique-finding (which was rarely needed in our experiments, cf. [14, Figure 2]). With only volume leakage, it is impossible to distinguish counts from smallest to largest with counts from largest to smallest; if the element counts were mirrored, the exact same set of volumes would have been observed.

The pre-processing phase grows a set of necessary elementary volumes (volumes that must be elementary) and shrinks a set of candidate elementary volumes (volumes that might be elementary). When the set of necessary elementary volumes has size *N*, the algorithm has succeeded. Adding to the set of necessary elementary volumes can occur in the following ways:

- Adding the smallest complementary volume. By properties of ranges, it must be either vol([1,1]) or vol([N,N]). Since reconstruction is only up to reflection, the algorithm can arbitrarily pick one.
- Adding the largest complementary volume, which is *R*=vol([1,*N*]) by definition.
- If there is an observed volume that arises only as edges incident to a non-necessary candidate node, then adding that node.
- If there is an observed volume that arises only as a node, and not as the difference of any 2 nodes, then adding it.
- Lastly, adding all non-necessary nodes when the total number of candidate nodes is *N*.

After adding a new necessary elementary volume, the algorithm shrinks the set of candidate elementary volumes by removing any volumes that are not adjacent to all necessary elementary volumes. When the set of necessary elementary volumes has *N* nodes, the algorithm has successfully recovered the number of rows with each value, up to reflection These *N* elementary volumes are either the volumes of the ranges [1,1], ..., [1,*N*] (or the volumes of the ranges [*N*,*N*], [*N*-1,*N*], ..., [1,*N*]). To recover the counts of the number of records with value *k*, compute vol([1,*k*]) - vol([1,*k*-1]).

Limitations. This attack requires having the complete set of all possible volumes. The complete set must be observed in a period where there are no updated values or newly inserted records in the database. It also requires there to be at least one record with each value. We discuss these limitations – and some techniques for handling them – in our paper [14, 15].

Security Recommendations

Leaking access pattern and volume of query results can be devastating and lead to bypassing database encryption, regardless of what type of encryption is used. Here are some recommendations for practitioners from the point of view of a cryptographer.

Analyzing leakage. I suggest analyzing leakage from two angles: first, by making a comprehensive list of all types of leakage that could occur in your solution, and second, looking at specific points or operations and thinking about what type of leakage could arise there.

One way to classify properties that leak is by whether they apply to the values in records, to queries, or to responses. In addition to leaking the values themselves, records can leak the ordering of values, the existence of certain values, the number of distinct values, the distance between values, repeated values, etc. Repetition allows attacks based on frequency analysis – an adversary that already knows something about what distribution of values to expect can infer a lot about specific values if their frequencies are leaked. Next, queries can leak information about the range endpoints, when two queries repeat, the width of the range, when one range is a subset of another range, etc. In addition to leaking which rows matched or the number of matching records, the answers to queries can leak when two sets of matching records are the same, which records match the most queries, etc. See Kamara, Moataz, Ohrimenko's work [21] for a comprehensive list of types of leakage.

Next, think about at what points in the implementation all of these kinds of leakage can arise. It could be when adding records, making single or batched queries, or recording information to profile queries. It could also be when performing maintenance operations, such as data backups, key rotation, or updating a search index. Consider a kind of differential analysis: what changes in the adversary's view when two underlying properties of the data, query, or response are not the same?

Trade-offs. There is no panacea when it comes to encrypted databases; trade-offs are necessary. In addition to considering what could leak and where, you can think about which of the following techniques for mitigating leakage would be acceptable in your setting: restricting the type or granularity of queries, post-processing query results locally, adding dummy records, making dummy queries, or trusting hardware. Gaining security could come at the cost of incomplete results, probabilistically correct results, efficiency when querying data, forgoing compression or deduplication of data or queries.

Conclusion

This technical report was about new kinds of side channel attacks that can break database encryption for numeric data. It discussed two kinds of leakage, access pattern leakage and volume leakage, that can arise, from, e.g., an adversary that has compromised the database server or a network eavesdropper. An adversary can use access pattern leakage along with a PQ tree to figure out the value of every record in the database, either exactly or approximately, bypassing any encryption. Using volume leakage, an adversary can create and prune a graph to identify the special elementary query volumes, which in turns leads to completely determining how many records have each value. These attacks apply to any kind of scheme supporting numeric ranges queries that have these side channels. There is no simple way to eliminate this leakage in all threat models, nor are access pattern and volume leakage the only kinds of harmful leakage. However, there do exist countermeasures, such as adding dummy records, making dummy queries, and increasing the amount of client-side post-processing of query results.

Further Reading

Kellaris, Kollios, Nissim, and O'Neill introduced generic reconstruction attacks on outsourced databases using access pattern and communication volume [22]. Range queries are not the only type of query on which generic attacks lead to reconstructing data; such attacks have also been devised on access pattern leakage from one-dimensional *k*-Nearest Neighbour queries [24, 23]. For a broad introduction to the area of cryptographically protected database search (both constructions and attacks), see the excellent Systematization of Knowledge (SoK) paper by Fuller et al. from 2017 [9]. For a glimpse of other topics in the broader area of searching on encrypted data, see videos from the Encrypted Search workshop held in June 2019 at Brown University [19].

Acknowledgments

The author thanks her brilliant co-authors of the papers on which this briefing was based – Paul Grubbs (Cornell University), Brice Minaud (École Normale Supérieure, CNRS, PSL University, Inria), and Kenny Paterson (ETH Zurich) – and new co-workers at NCC Group for ideas on how to present this content at Black Hat.

References

[1] Rakesh Agrawal et al. "Order Preserving Encryption for Numeric Data". In: *Proceedings of the 2004 ACM SIGMOD International Conference on Management of Data*. SIGMOD '04. Paris, France: ACM, 2004, pp. 563–574.

[2] Daniel Bleichenbacher. "Chosen Ciphertext Attacks Against Protocols Based on the RSA Encryption Standard PKCS #1". In: *CRYPTO'98*. Ed. by Hugo Krawczyk. Vol. 1462. LNCS. Springer, Heidelberg, Aug. 1998, pp. 1–12. DOI: 10.1007/BFb0055716.

[3] Tobias Boelter, Rishabh Poddar, and Raluca Ada Popa. *A Secure One-Roundtrip Index for Range Queries*. Cryptology ePrint Archive, Report 2016/568. http://eprint.iacr.org/2016/568. 2016.

[4] Alexandra Boldyreva, Nathan Chenette, and Adam O'Neill. "Order-Preserving Encryption Revisited: Improved Security Analysis and Alternative Solutions". In: *CRYPTO 2011*. Ed. by Phillip Rogaway. Vol. 6841. LNCS. Springer, Heidelberg, Aug. 2011, pp. 578–595. DOI: 10.1007/978-3-642-22792-9_33.

[5] Alexandra Boldyreva et al. "Order-Preserving Symmetric Encryption". In: *EUROCRYPT 2009*. Ed. by Antoine Joux. Vol. 5479. LNCS. Springer, Heidelberg, Apr. 2009, pp. 224–241. DOI: 10.1007/978-3-642-01001-9_13.

[6] Kellogg S. Booth and George S. Lueker. "Testing for the Consecutive Ones Property, Interval Graphs, and Graph Planarity Using PQ-Tree Algorithms". In: *J. Comput. Syst. Sci.* 13.3 (1976), pp. 335–379. DOI: 10.1016/S0022-0000(76)80045-1. URL: https://doi.org/10.1016/S0022-0000(76)80045-1.

[7] Nathan Chenette et al. "Practical Order-Revealing Encryption with Limited Leakage". In: *FSE 2016*. Ed. by Thomas Peyrin. Vol. 9783. LNCS. Springer, Heidelberg, Mar. 2016, pp. 474–493. DOI: 10.1007/978-3-662-52993-5_24.

[8] CipherCloud. *CipherCloud Information Protection Overview v2.5*. June 2013. URL: http://pages.ciphercloud.com/rs/ciphercloud/images/CipherCloud%20Technical%20Whitepaper%20v2.5.pdf.

[9] Benjamin Fuller et al. "SoK: Cryptographically Protected Database Search". In: *2017 IEEE Symposium on Security and Privacy*. IEEE Computer Society Press, May 2017, pp. 172–191. DOI: 10.1109/SP.2017.10.

[10] Paul Grubbs, Thomas Ristenpart, and Vitaly Shmatikov. "Why Your Encrypted Database Is Not Secure". In: *Proceedings of the 16th Workshop on Hot Topics in Operating Systems*. HotOS '17. Whistler, BC, Canada: ACM, 2017, pp. 162–168. ISBN: 978-1-4503-5068-6. DOI: 10.1145/3102980.3103007. URL: http://doi.acm.org/10.1145/3102980.3103007.

[11] Paul Grubbs et al. "Leakage-Abuse Attacks against Order-Revealing Encryption". In: *2017 IEEE Symposium on Security and Privacy*. IEEE Computer Society Press, May 2017, pp. 655–672. DOI: 10.1109/SP.2017.44.

[12] Paul Grubbs et al. "Learning to Reconstruct: Statistical Learning Theory and Encrypted Database Attacks". In: *2019 IEEE Symposium on Security and Privacy*. IEEE Computer Society Press, 2019, pp. 496–512. DOI: 10.1109/SP.2019.00031.

[13] Paul Grubbs et al. *Learning to Reconstruct: Statistical Learning Theory and Encrypted Database Attacks*. Cryptology ePrint Archive, Report 2019/011. https://eprint.iacr.org/2019/011. 2019.

[14] Paul Grubbs et al. "Pump up the Volume: Practical Database Reconstruction from Volume Leakage on Range Queries". In: *ACM CCS 2018*. Ed. by David Lie et al. ACM Press, Oct. 2018, pp. 315–331. DOI: 10.1145/3243734.3243864.

[15] Paul Grubbs et al. *Pump up the Volume: Practical Database Reconstruction from Volume Leakage on Range Queries*. Cryptology ePrint Archive, Report 2018/965. https://eprint.iacr.org/2018/965. 2018.

[16] Troy Hunt. *Have I Been Pwned (HIBP)*. 2019. URL: https://haveibeenpwned.com/.

[17] IBM Knowledge Center. *Db2 native encryption*. Db2 11.5. 2019. URL: https://www.ibm.com/support/knowledgecenter/en/SSEPGG_11.5.0/com.ibm.db2.luw.admin.sec.doc/doc/c0061758.html.

[18] IETF TLS Working Group. *TLS*. 2019. URL: https://tlswg.org/.

[19] Institute for Computational and Experimental Research in Mathematics. *ICERM - Encrypted Search*. June 2019. URL: https://icerm.brown.edu/topical_workshops/tw19-1-es.

[20] Jonathan L. Dautrich Jr. and Chinya V. Ravishankar. "Compromising privacy in precise query protocols". In: *Joint 2013 EDBT/ICDT Conferences, EDBT '13 Proceedings*. 2013, pp. 155–166. DOI: 10.1145/2452376.2452397. URL: http://doi.acm.org/10.1145/2452376.2452397.

[21] Seny Kamara, Tarik Moataz, and Olga Ohrimenko. "Structured Encryption and Leakage Suppression". In: *CRYPTO 2018, Part I*. Ed. by Hovav Shacham and Alexandra Boldyreva. Vol. 10991. LNCS. Springer, Heidelberg, Aug. 2018, pp. 339–370. DOI: 10.1007/978-3-319-96884-1_12.

[22] Georgios Kellaris et al. "Generic Attacks on Secure Outsourced Databases". In: *ACM CCS 2016*. Ed. by Edgar R. Weippl et al. ACM Press, Oct. 2016, pp. 1329–1340. DOI: 10.1145/2976749.2978386.

[23] Evgenios M. Kornaropoulos, Charalampos Papamanthou, and Roberto Tamassia. *Data Recovery on Encrypted Databases With k-Nearest Neighbor Query Leakage*. Cryptology ePrint Archive, Report 2018/719. https://eprint.iacr.org/2018/719. 2018.

[24] Evgenios M. Kornaropoulos, Charalampos Papamanthou, and Roberto Tamassia. "Data Recovery on Encrypted Databases with k-Nearest Neighbor Query Leakage". In: *2019 IEEE Symposium on Security and Privacy*. IEEE Computer Society Press, 2019, pp. 245–262. DOI: 10.1109/SP.2019.00015.

[25] Kevin Lewi and David J. Wu. "Order-Revealing Encryption: New Constructions, Applications, and Lower Bounds". In: *ACM CCS 2016*. Ed. by Edgar R. Weippl et al. ACM Press, Oct. 2016, pp. 1167–1178. DOI: 10.1145/2976749.2978376.

[26] Microsoft SQL Docs. *Always Encrypted (Database Engine)*. 2017. URL: https://docs.microsoft.com/en-us/sql/relational-databases/security/encryption/always-encrypted-database-engine?view=sql-server-2017.

[27] Microsoft SQL Docs. *Transparent Data Encryption (TDE)*. 2019. URL: https://docs.microsoft.com/en-us/sql/relational-databases/security/encryption/transparent-data-encryption.

[28] MongoDB. *Client-Side Field Level Encryption*. MongoDB Manual 4.2. June 2019. URL: https://docs.mongodb.com/manual/core/security-client-side-encryption/.

[29] MongoDB. *Encryption at Rest*. MongoDB Manual 4.2. June 2019. URL: https://docs.mongodb.com/manual/core/security-encryption-at-rest/.

[30] Muhammad Naveed, Seny Kamara, and Charles V. Wright. "Inference Attacks on Property-Preserving Encrypted Databases". In: *ACM CCS 2015*. Ed. by Indrajit Ray, Ninghui Li, and Christopher Kruegel. ACM Press, Oct. 2015, pp. 644–655. DOI: 10.1145/2810103.2813651.

[31] Oracle. "Using Transparent Data Encryption". In: *Oracle Database Online Documentation Library, 12c Release 1 (12.1.0.2)*. .Part 1. 2019. URL: https://docs.oracle.com/database/121/ASOAG/asopart1.htm.

[32] Raluca Ada Popa et al. "CryptDB: Protecting Confidentiality with Encrypted Query Processing". In: *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*. SOSP '11. Cascais, Portugal: ACM, 2011, pp. 85–100. ISBN: 978-1-4503-0977-6. DOI: 10.1145/2043556.2043566. URL: http://doi.acm.org/10.1145/2043556.2043566.

[33] Roei Schuster, Vitaly Shmatikov, and Eran Tromer. “Beauty and the Burst: Remote Identification of Encrypted Video Streams”. In: *USENIX Security 2017*. Ed. by Engin Kirda and Thomas Ristenpart. USENIX Association, Aug. 2017, pp. 1357–1374.

[34] Dawn Xiaodong Song, David A. Wagner, and Xuqing Tian. “Timing Analysis of Keystrokes and Timing Attacks on SSH”. In: *USENIX Security 2001*. Ed. by Dan S. Wallach. USENIX Association, Aug. 2001.

[35] Wikipedia contributors. *List of data breaches — Wikipedia, The Free Encyclopedia*. 2019. URL: `https://en.wikipedia.org/w/index.php?title=List_of_data_breaches&oldid=910064898`.