Why Your Encrypted Database Is Not Secure

Paul Grubbs Thomas Ristenpart Vitaly Shmatikov
Cornell Tech

Abstract

Encrypted databases, a popular approach to protecting data from compromised database management systems (DBMS's), use abstract threat models that capture neither realistic databases, nor realistic attack scenarios. In particular, the "snapshot attacker" model used to support the security claims for many encrypted databases does not reflect the information about past queries available in any snapshot attack on an actual DBMS.

We demonstrate how this gap between theory and reality causes encrypted databases to fail to achieve their "provable security" guarantees.

1 Introduction

Continuing large-scale compromises of systems that manage sensitive information have motivated active research on the design and implementation of *encrypted databases*. Encrypted databases operate on top of a commodity database management system (DBMS) such as MySQL or MongoDB but store data in an encrypted form so that even if the DBMS or underlying OS is compromised, the attacker cannot obtain the protected data. These systems include research prototypes such as CryptDB [46], Arx [45], and Seabed [42], as well as deployed industry solutions [14, 54].

For efficiency, encrypted databases rely on specialized encryption schemes that allow the server, given only the ciphertexts, to perform some computations in response to client queries. The price is the leakage of partial information about plaintexts. Recent work [10, 22, 23, 27, 29, 39] demonstrated how an attacker can recover plaintext data if he observes queries over the encrypted database. This attacker is called a *persistent attacker*. In response, designers of encrypted databases have focused on a weaker *snapshot attacker* who can only obtain a single static observation of the compromised system. Many recent encrypted databases make strong claims of "provable security" [42, 45, 49] against snapshot attacks.

The theoretical models used to support these claims are abstractions. They are not based on analyzing the actual information revealed by a compromised database system and how it can be used to infer the plaintext data. It is well-known that forensic analysis of storage systems reveals much about their operation [17, 18, 58], but this knowledge has not, to date, been reflected in the security models used by the designers of encrypted databases.

In this paper, we take a system-centric view of encrypted databases and investigate what an attacker would learn in a realistic scenario: stealing a disk, performing SQL injection, or rootkitting the OS. We demonstrate that a "snapshot" attacker, which is the main security model of most encrypted databases, is largely a myth. Modern DBMS's keep logs, caches, and data structures that, in any realistic snapshot attack, reveal information about past queries. This leakage is inherent in today's production environments because a DBMS must maintain caches and other metadata to adapt the system to the workload and help manage its performance.

We then concretely demonstrate how an attacker can exploit this system-level information to break the claimed security guarantees of encrypted databases. We conclude with guidelines for future research.

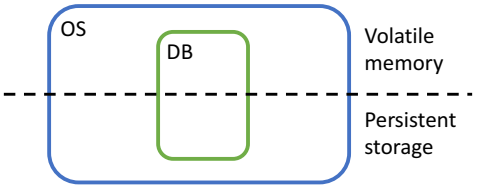
2 Attacks on Databases

We use a simple abstraction to explain (1) academic threat models in the encrypted database (EDB) literature and (2) concrete attacks that DBMS's face in reality.

Academic threat models. The strongest threat model (e.g., mentioned in [45, 48]) is an *active attacker* who fully compromises the DBMS server and performs arbitrary malicious operations. Such attacks are difficult to defend against (viz. [22]), and recent designs for efficient EDBs no longer claim security against active attacks.

Instead, the latest security models focus on passive attacks that do not interfere with DBMS functionality. The weaker version is a **snapshot attacker** [22, 32, 45] who obtains "a snapshot of the database (tables and indices included)" [45]. The stronger version is a **persistent passive attacker** [22] who compromises the DBMS server and passively observes all its operations. The latter includes observing the queries issued to the database and how they access the encrypted data.

Observations of query evaluation are particularly damaging in EDBs that rely on property-revealing encryption



	Logs	Diagnostic Tables	Data Structures
Disk theft	✓		
SQL injection	✓	✓	
VM snapshot leak	✓	✓	✓
Full-system compromise	✓	✓	✓

Figure 1: We use a simple abstraction of DB-hosting systems (left) to explain the discrepancies between the academic threat models and the information revealed in actual attacks (right).

(PRE), such as order-revealing encryption [32], deterministic encryption [4], and searchable encryption [11]. All PRE schemes leak some information about plaintexts in order to support certain computations by the DBMS. Some PRE ciphertexts always leak [4, 7], enabling powerful snapshot attacks that recover plaintexts [10, 23, 39]. Other PRE schemes [11, 32, 48] leak only if the attacker observes accesses to the ciphertexts (e.g., search queries). By definition, a persistent attacker can exploit the leakage from queries and accesses to recover plaintexts [10, 22, 27, 29].

Since PRE schemes are always vulnerable to persistent attacks, many EDBs claim security against snapshot attacks only. Examples include the latest claims [47,49] for CryptDB [46] and Mylar [48] (revised after the original claims were shown false by, respectively, [39] and [22]), new systems Arx [45] and Seabed [42], and new cryptographic schemes such as Lewi-Wu order-revealing encryption [32]. A common implicit assumption for these systems is that snapshot attackers will *not* obtain past queries. We will show that this assumption is false in commodity DBMS's under realistic snapshot attacks.

A simple system abstraction. We will treat a DBMS as if it consists of (a) the DB software running as one or more user-level processes and (b) the rest of the system, including the OS and other applications (OS, for brevity). Therefore, the state of the system has four parts: volatile DB state (data in RAM and CPU registers), persistent DB state (data on disk), volatile OS state, and persistent OS state—see Figure 1. For simplicity, we assume the database is not sharded across multiple machines, i.e., even if the database is replicated, every machine has a full copy of the data.

We use MySQL as our running example, but similar caches, logs, and data structures exist in all practical DBMS's and can be recovered via forensic analysis (e.g., see [8] for MongoDB).

Concrete attacks. An oft-cited threat to DB security is ***disk theft***, i.e., theft of persistent storage [2, 16, 20, 32, 37, 41, 46, 49]. Full-disk encryption (FDE) can mitigate this threat, but EDBs aim to protect data even in the absence of FDE. Without FDE, this attack yields the persistent OS and DB state, but not any volatile state.

SQL injection is an old but still prevalent [59] attack. It also enables arbitrary code injection and full control of the memory space of the DB process [15, 24], thus yielding the persistent and volatile DB state.

DBMS's increasingly run on virtual machines (VMs), exposing them to the threat of ***VM image leaks*** [3, 9, 19, 52]. Some VM snapshots only contain the persistent storage, whereas full-state snapshots also include the VM's memory and CPU registers. We focus on the latter. This attack yields the persistent and volatile OS and DB state.

Finally, a ***full-system compromise*** involves rooting the DBMS and gaining full access to the persistent and volatile OS and DB state. This enables persistent passive and active attacks, but "smash-and-grab" attacks that simply grab available data and leave are prevalent [59].

The table on the right of Figure 1 summarizes DBMS-specific data yielded by the attacks of each type. In the rest of the paper, we explain how this data reveals information about past queries, thus breaking the security models of EDBs and enabling recovery of plaintext data.

3 Logs on Disk

We start by investigating what information about past queries can be gleaned from the log files on disk that are required for high availability and transactional semantics in a production DBMS.

Inferring writes. Industrial databases must support transactions with ACID properties (atomicity, consistency, isolation, durability). We'll use MySQL as an example. Other DBMS's such as SQL Server, Postgres, and MongoDB use similar techniques.

MySQL's default storage engine, InnoDB, uses circular *undo* and *redo* logs to give the database layer multi-version concurrency control. Both logs record changes to the individual database records at the byte level. Using standard forensic techniques for reconstructing insert, update, and delete transactions from these logs [17, 18], an attacker who compromised the disk can reconstruct queries that modified the database. The number of reconstructed queries depends on insertion size and volume. For example, with 1 write modifying a 20-byte field per

second, the undo and redo logs of default size (50 Mb) store 16 days' worth of inserts.

Recent work [22] showed that the timing of queries reveals sensitive data in certain applications of encrypted databases. In MySQL, timing can be extracted from a separate *binary log* (binlog) used to support replicated transactions and point-in-time recovery [5]. Binlog stores the text of every transaction that modifies any row of the database, along with its UNIX timestamp. It is not enabled upon installation but must be turned on for high availability and therefore will be present on the disk of production MySQL servers. This log is so important that a utility for reading it (`mysqlbinlog`) comes preinstalled with MySQL [6]. Its contents are never purged unless the administrator executes a special command. A similar mechanism for replicated transactions in MongoDB also records transaction timestamps [36]. Even without this log, the default primary key of each MongoDB document contains its creation time [8].

MySQL's binlog also enables the attacker to compute the correlation between the timestamps and the rate of change in the log sequence numbers (LSN). The attacker can thus infer the approximate timestamps for the transactions in the undo and redo logs that are no longer present in the binlog.

This leakage is inherent in ACID databases. Transactional guarantees require the ability to roll back recent transactions (perhaps even across database crashes), thus information about recent database modifications must persist on the disk.

Inferring reads. The easiest way for information about reads to end up on disk is through too-verbose logging. In MySQL, the general query log records every query, including SELECT, but few systems enable it because it takes huge amounts of disk space. Instead, on many production MySQL systems, the "slow query" log [55] records transactions that take an unusually long time.

A more subtle way to extract information about read-only queries is from the buffer pool file. On shutdown and at other points during normal server operation, MySQL creates a file in the data directory containing the current pages in the buffer pool in LRU order. This is done to avoid a "warm-up" period of slow responses after a restart. This file reveals information about several previous SELECT queries, such as the paths through the B+ tree that MySQL took when evaluating them.

4 Diagnostic Tables

SQL injection is still a common way to compromise databases [59]. Designers of encrypted databases often assume that a SQL injection attack reveals only the database's view of the data itself. But modern DBMS's include tables—extractable via SQL injection—that store a great deal of performance statistics, intended to help tune specific databases to their workloads and diagnose problems and performance bottlenecks [33, 35, 50, 57].

The **information_schema** database in MySQL [26] aggregates information about the internal state of the DBMS, including contents of caches and how many connections are active. It also includes a `processlist` table with the timestamped list of all currently executing queries. By injecting a SELECT query on this table, an attacker can obtain queries made by other users.

The **performance_schema** database [44] aggregates statistics about query execution, such as the number of queries per second and the amount of contention for synchronization objects. It also contains a `threads` table with the current statements being executed by all threads, enabling a SQL-injection attacker to monitor queries.

This database also keeps information about all past queries. The `events_statements_current` table stores the current statement being executed by any thread. The `events_statements_history` table stores the most recent queries made by any thread (essentially, the most recent queries appearing in `events_statements_current` for all threads). The number of queries stored per thread is configurable (10 by default). In addition to the text of the query, it also stores the number of rows examined by the query and returned to the client.

MySQL does not store historical information about every individual query, but `performance_schema` stores statistics about all query "types" made since the database was last restarted. The "type" is determined by a simple canonicalization algorithm which removes the arguments but preserves the select-from-where structure of the query and the attributes it uses. So, for example, the queries `SELECT * FROM CUSTOMERS WHERE STATE='IN'` and `SELECT * FROM CUSTOMERS WHERE STATE='AZ'` have the same canonical form, which is different from the canonical form of the queries `SELECT * FROM CUSTOMERS WHERE AGE >=25` and `SELECT * FROM CUSTOMERS WHERE STATE='IN' AND AGE >=25` (the `WHERE` clause has multiple constraints in the latter).

Even if the DBMS has internal access controls, SQL injection can be leveraged into arbitrary code execution [15, 24] that bypasses all access restrictions within the DBMS process.

5 In-memory Data Structures

The strongest snapshot attack scenario involves the attacker obtaining an image of the virtual machine execut-

ing the DBMS or, alternatively, rootkitting the OS (but only making a single observation of the system). This snapshot reveals a point-in-time state of the entire persistent and volatile memory. We focus on the internal data structures of the database process because they reveal information about past queries, especially accesses to the individual pages in its cache.

To adaptively improve performance and support (amortized) constant-time retrieval for frequently accessed database pages, InnoDB keeps per-page metadata and access counters. If a page is accessed often, InnoDB indexes its contents in an **adaptive hash index** [1]. Postgres has a similar mechanism for tracking accesses to individual pages to handle eviction from its buffer cache.

Further, the **query cache** in MySQL is an internal key-value map that can be configured to keep the results of certain SELECT queries [51] so that answering them is essentially free. Unlike the buffer pool, this cache is strictly internal to MySQL and cannot be exposed via `information_schema` (see Section 4), but will be visible to a whole-system snapshot attacker. Other commodity DBMS's, too, implement some form of query caching—e.g., Microsoft SQL Server caches queries and their execution plans but not the full result set [34].

Even if the query cache is disabled, queries persist in MySQL's internal heap long after they've been executed. We performed a simple experiment with MySQL in the default configuration. First, we issued a SELECT query with a random string as the column name. This random string appears nowhere in the database, thus the query does not match any rows. Then, we issued 100 SELECT queries which matched some rows and 900 that did not. Then, we inserted 500 random rows and made 1,000 more SELECT queries, waited around twenty minutes and made 100,000 more SELECT queries. After this, we dumped the memory of the MySQL process.

The full text of the original query appeared in three distinct locations in memory, and the random string appeared in three additional locations by itself. We verified that this is not a peculiarity of how MySQL handles column names by repeating the experiment with a random string parameter in a WHERE clause. This leak is not surprising since MySQL is not designed for security-critical operations and does not implement secure deletion. In Section 6, we show that in the context of encrypted databases this otherwise minor oversight has dramatic implications for the (lack of) security.

6 How Systems Fail

We now explain how the confidentiality of data in encrypted databases can be broken by snapshot attackers using techniques described above. We focus on the encrypted databases that have been designed to work on top of commodity DBMS's.

At-rest encryption. This protection works the same way in most DBMS's: a key, stored in memory but not on disk, is used to encrypt the database files on disk. An attacker who compromises only the disk will therefore learn nothing useful (except via side channels such as relative sizes of encrypted objects), but any higher level of access will reveal the entire data.

Token-based systems. Many encrypted databases are based on schemes that delegate a query-specific trapdoor to the server. The server uses it to reveal information about the plaintexts necessary to answer the query. For example, CryptDB and Mylar [46,48] use variants of the scheme of Song et al. [56]. More advanced examples include the ORE scheme of Lewi and Wu [32] and the searchable encryption scheme of Cash et al. [11].

For any such scheme, semantic security [21] cannot be achieved if the attacker obtains even a single token value. Intuitively, semantic security requires that even if the attacker knows the original query, he cannot tell the difference between an encrypted record that matched the query and one that didn't. If the attacker observes the query token, he can apply it to the encrypted database and recognize which records match and which don't, thus breaking the definition of semantic security.

As we explained in Section 5, the text of queries (and, therefore, the search token) is stored in several locations in MySQL. Queries also appear in two log files, the query cache, `performance_schema` and `information_schema`. They can even be found after the fact in the internal heap of the DBMS. Tokens will thus be available to any realistic snapshot attacker.

The consequences depend on the system. For CryptDB, Mylar, and any other system using variants of searchable encryption [11,30,46,48], a snapshot attacker can use the leakage-abuse attacks such as [10] to infer the query and the plaintext of any record it matches.

These attacks exploit the observation that the number of results that match a query is often unique across a corpus, e.g., 63% of the 500 most frequent words in the Enron email corpus have a unique result count. With partial knowledge of the encrypted documents, unique counts immediately reveal the value of the corresponding encrypted keyword. Since the search functionality also reveals which documents contain the keyword, this attack also recovers partial content of the encrypted documents.

Lewi-Wu ORE . In the Lewi-Wu scheme [32], query tokens reveal ordering information and, in some parameter regimes, individual plaintext bits. A damaging attack against an ORE with similar leakage was demonstrated in [23], but it does not directly apply to the Lewi-Wu ORE because the Lewi-Wu scheme is not deterministic.

Nevertheless, the Lewi-Wu scheme reveals equality of plaintexts when a value in the database is queried. This leaks a partial histogram to a snapshot attacker, who can combine it with the bit leakage from the query tokens and the "binomial attack" of [23], to which the Lewi-Wu scheme is vulnerable even in the absence of tokens.

To show the consequences of this leakage, we simulated an attack on the Lewi-Wu scheme (with block size of 1 bit). We sampled a database of 32-bit integers and several range queries (both an upper and lower bound), all uniformly at random. We then computed the leakage resulting from each set of queries if executed against a given database, aggregating the results over 1,000 trials.

For a database of size 10,000 and only *five* simulated range queries, the average fraction of bits leaked (out of possible 320,000) is surprisingly high, around 12%, i.e., 4 bits of each 32-bit value are leaked on average. For twenty-five range queries, the fraction is 19%. If fifty range queries are found in the memory snapshot (this is not inconceivable in practice; MySQL can create dozens of threads for query processing and network I/O), the snapshot attacker recovers 25% of the bits (on average, 8 bits of each 32-bit value). We did not attempt to estimate whether the leakage would be equally severe for non-uniform distributions of range queries.

In summary, query tokens found in system snapshots enable a snapshot adversary to recover large amounts of protected data in all existing encrypted databases.

Seabed. Seabed's ORE scheme [42] is known to be insecure [23]. The attack of [23] uses the known plaintext distribution (auxiliary model), which is publicly available for many types of data. It starts by computing all possible comparisons between the ciphertexts, as permitted by the ORE scheme, to learn some bits of the underlying plaintexts. Then, it creates a bipartite graph in which each ciphertext is a node on the left-hand side and each possible plaintext is a node on the right-hand side, and draws an edge between a left-hand node and a right-hand node only if the bits it learned about the left-hand ciphertext match the bits of the right-hand plaintext. Each edge in the graph is weighted using frequency information. Finally, the attack recovers the most likely plaintext for each ciphertext by finding a matching in the graph.

For data in the columns that need to support joins, Seabed uses basic deterministic encryption (DET). This data is therefore vulnerable to the frequency analysis attack described below. For data in the columns used as filters in count or aggregation queries, Seabed attempts to prevent frequency analysis using the SPLASHE scheme, which creates a different column for each possible plaintext. Analytics queries such as aggregations are rewritten and encrypted so they are evaluated on the correct column. For example, if the plaintext 10 corresponds to the `c3` column of `table`, the query `SELECT count(*) FROM table WHERE a = 10` is converted into an equivalent of `SELECT ashe(c3) FROM table`, where "ashe()" is a custom summation over ciphertexts [42, Table 2].

If two queries have different values in the WHERE clause, they will operate on different columns (after rewriting). If SPLASHE were to run on MySQL, the `events_statements_summary_by_digest` table in the `performance_schema` database will canonicalize them to different forms. This table will thus count the number of queries made *for each plaintext*. This reveals the exact histogram of queries for each plaintext value to any attacker with a snapshot of the DBMS memory. If SPLASHE runs on Spark, the attacker can simply obtain queries from the event history server [57] or from the heap of the worker nodes.

Characterizing the exact leakage of query distributions (as opposed to plaintext distributions) is an open problem in general, but if the attacker has a sufficiently good model of the query distribution, then basic inference attacks like frequency analysis can be used.

Frequency analysis is a very simple cryptanalytic technique which would work here in two steps. In the first step, the observed histogram of the ciphertexts and the histogram of the query distribution model would both be sorted in decreasing order. So, for example, the most frequently occurring ciphertext query would be the first element in the list of queries, and the most frequently occurring query according to the model would be the first element on its list. In the second step, the elements of the lists are matched by rank: the first elements are matched with each other, then the second elements, and so on. Lacharité and Paterson [31] proved that this simple process is a maximum-likelihood estimator for the encryption function, meaning that this attack is most likely to correctly recover the underlying plaintexts.

While frequency analysis can recover the plaintext corresponding to a given column in Seabed, it will not recover the value of that column for a particular row. However, to save space, an enhanced version of SPLASHE uses deterministic encryption with padding for infrequent plaintext values, rather than creating a dedicated column in the schema. The `performance_schema` will leak a query histogram for the frequently occurring values (as described above) in this scheme, but will not leak a histogram of the infrequently occurring values. Nevertheless, a partial histogram could be reconstructed from the logs or in-memory data structures. This leakage is even more damaging against enhanced SPLASHE because the frequency analysis attack described above can reveal the value in the enhanced SPLASHE column for a particular row. Combined with other leakage about frequent values from query patterns and any DET- or ORE-encrypted columns, this enables even more dam-

aging cross-column inference attacks.

Arx. Arx [45] uses a treap-based data structure to evaluate range queries on encrypted data in a single network round-trip using chained garbled circuits. Because index values are encrypted using standard encryption, the authors claim semantic security against snapshot attacks.

An important property of the Arx scheme is that after each range query, the nodes of the treap become "consumed" and must be repaired; essentially the client must supply a new encryption of the node's value which overwrites the old value. Reads and writes are thus perfectly correlated because a read of any node is immediately followed by a write to the same node. If Arx runs on MySQL, MongoDB, or a similar DBMS, a snapshot of the system's persistent state will contain a transcript of every range query made on the index because the write corresponding to each read will be recorded in the transaction logs. This breaks semantic security and gives a snapshot attacker much of the information a persistent attacker would have. For example, this snapshot attacker will have ordering information about the upper and lower bounds of encrypted range queries, as well as the frequency of visits to each node in the tree.

Arx uses a two-round protocol to hide the relationship between the node values of the range query index and the database rows holding that value, thus a snapshot attack on the range query data structure does not immediately reveal the plaintext in a given row. Nevertheless, the leakage is sufficient to recover the values in the index using a variant of the bipartite matching attack from [23] described above. The index does not leak the frequencies of individual values, but transaction logs do leak the frequencies of visits to each value in the index. These frequencies can be used in combination with auxiliary data about the distribution of queries to recover these values.

Moreover, transaction logs leak the *rank* of the queries, i.e., the number of values in the index less than the query. Prior work has shown how to exploit this leakage [23], and we conjecture that a similar approach can be used to recover the plaintext of the encrypted queries. We leave full development of this attack to future work.

7 Discussion

Deploying encrypted databases on commodity DBMS's can have unexpectedly bad consequences for security. Logs, caches, and data structures kept by DBMS's leak information that is not accounted for in the threat models used by the designers of encrypted databases. Critically, today there is no such thing as a "snapshot" attacker who cannot observe past queries, workloads, and access patterns—because any realistic snapshot of the system contains this information. We demonstrated how this leads to confidentiality breaches.

We focused on commodity DBMS's, but similar issues arise in schemes using custom or non-standard databases [12, 28, 40, 43]. There is a trend in database design towards adaptively changing the structure of the database based on the workload [13, 25, 53]. We expect that the snapshots of such databases leak even more information about past queries.

There appears to be an inherent conflict between security and transparency: if the internal information about workloads is available to the developers and administrators, it is also available in some form to a snapshot attacker. The tension between effective caching and security was noted in the early research on history-independent data structures [38], but whether history independence can be achieved for practical encrypted databases remains an open question. Solving it requires new research into designing and implementing databases that efficiently hide queries and access patterns.

We conclude with guidelines and recommendations for the different research communities working in the area of encrypted databases.

Cryptographers: A desirable property of new schemes is that the encrypted data by itself leaks nothing about the plaintext. This does not imply security against "snapshot" or "offline" attacks when the scheme is actually deployed in a real DBMS.

Systems researchers: Any system that uses any kind of property-revealing encryption must be assessed using state-of-the-art leakage-abuse attacks as part of the standard evaluation, and the results of the assessment must be presented in the evaluation section of the paper.

PC members of systems conferences: Any paper that proposes an encrypted database but does not assess its security under known attacks should be treated with suspicion. Be very skeptical of claims of "provable confidentiality," especially if not supported by a thorough security evaluation, and solicit external reviews from cryptographers and security researchers to validate these claims.

Acknowledgements

Grubbs and Ristenpart both have large financial stakes in Skyhigh Networks. This work was supported in part by NSF grants CNS-1330308, CNS-1514163, and a generous gift from Microsoft.

References

[1] InnoDB adaptive hash index. https://dev.mysql.com/doc/refman/5.7/en/innodb-adaptive-hash.html.

[2] Arvind Arasu, Spyros Blanas, Ken Eguro, Raghav Kaushik, Donald Kossmann, Ravishankar Ramamurthy, and Ramarathnam Venkatesan. Orthogonal security with Cipherbase. In *CIDR*, 2013.

[3] Marco Balduzzi, Jonas Zaddach, Davide Balzarotti, Engin Kirda, and Sergio Loureiro. A security analysis of Amazon's elastic compute cloud service. In *SAC*, 2012.

[4] Mihir Bellare, Alexandra Boldyreva, and Adam O'Neill. Deterministic and efficiently searchable encryption. In *CRYPTO*, 2007.

[5] The binary log. `http://dev.mysql.com/doc/refman/5.7/en/binary-log.html`.

[6] mysqlbinlog — utility for processing binary log files. `http://dev.mysql.com/doc/refman/5.7/en/mysqlbinlog.html`.

[7] Alexandra Boldyreva, Nathan Chenette, Younho Lee, and Adam O'Neill. Order-preserving symmetric encryption. In *EUROCRYPT*, 2009.

[8] Matt Bromiley. MongoDB forensics. `http://tinyurl.com/zkexl86`, 2015.

[9] Sven Bugiel, Stefan Nürnberger, Thomas Pöppelmann, Ahmad-Reza Sadeghi, and Thomas Schneider. AmazonIA: When elasticity snaps back. In *CCS*, 2011.

[10] David Cash, Paul Grubbs, Jason Perry, and Thomas Ristenpart. Leakage-abuse attacks against searchable encryption. In *CCS*, 2015.

[11] David Cash, Stanislaw Jarecki, Charanjit Jutla, Hugo Krawczyk, Marcel-Cătălin Roşu, and Michael Steiner. Highly-scalable searchable symmetric encryption with support for boolean queries. In *CRYPTO*, 2013.

[12] Melissa Chase and Seny Kamara. Structured encryption and controlled disclosure. In *ASIACRYPT*, 2010.

[13] Surajit Chaudhuri and Vivek Narasayya. Self-tuning database systems: A decade of progress. In *VLDB*, 2007.

[14] Ciphercloud. `http://www.ciphercloud.com`.

[15] Muhaimin Dzulfakar. Advanced MySQL exploitation. Black Hat Las Vegas, 2009.

[16] InnoDB tablespace encryption. `https://dev.mysql.com/doc/refman/5.7/en/innodb-tablespace-encryption.html`.

[17] Peter Frühwirt, Marcus Huber, Martin Mulazzani, and Edgar R Weippl. InnoDB database forensics. In *AINA*, 2010.

[18] Peter Frühwirt, Peter Kieseberg, Sebastian Schrittwieser, Markus Huber, and Edgar Weippl. InnoDB database forensics: Reconstructing data manipulation queries from redo logs. In *ARES*, 2012.

[19] Tal Garfinkel and Mendel Rosenblum. When virtual is harder than real: Security challenges in virtual machine based computing environments. In *HotOS*, 2005.

[20] Tingjian Ge and Stan Zdonik. Fast, secure encryption for indexing in a column-oriented DBMS. In *ICDE*, 2007.

[21] Shafi Goldwasser and Silvio Micali. Probabilistic encryption. *Journal of Computer and System Sciences*, 28(2):270–299, 1984.

[22] Paul Grubbs, Richard McPherson, Muhammad Naveed, Thomas Ristenpart, and Vitaly Shmatikov. Breaking web applications built on top of encrypted data. In *CCS*, 2016.

[23] Paul Grubbs, Kevin Sekniqi, Vincent Bindschaedler, Muhammad Naveed, and Thomas Ristenpart. Leakage-abuse attacks against order-revealing encryption. In *S&P*, 2017.

[24] Bernardo Damele Assumpção Guimarães. Advanced SQL injection to operating system full control. *Black Hat Europe*, 2009.

[25] Stratos Idreos, Stefan Manegold, Harumi Kuno, and Goetz Graefe. Merging what's cracked, cracking what's merged: Adaptive indexing in main-memory column-stores. In *VLDB*, 2011.

[26] MySQL information schema. `https://dev.mysql.com/doc/refman/5.7/en/information-schema.html`.

[27] Mohammad Saiful Islam, Mehmet Kuzu, and Murat Kantarcioglu. Access pattern disclosure on searchable encryption: Ramification, attack and mitigation. In *NDSS*, 2012.

[28] Seny Kamara and Tarik Moataz. SQL on structurally-encrypted databases. Cryptology ePrint Archive, Report 2016/453, 2016.

[29] Georgios Kellaris, George Kollios, Kobbi Nissim, and Adam O'Neill. Generic attacks on secure outsourced databases. In *CCS*, 2016.

[30] John Kubiatowicz, David Bindel, Yan Chen, Steven Czerwinski, Patrick Eaton, Dennis Geels, Ramakrishan Gummadi, Sean Rhea, Hakim Weatherspoon, Westley Weimer, et al. Oceanstore: An architecture for global-scale persistent storage. *ACM Sigplan Notices*, 35(11):190–201, 2000.

[31] Marie-Sarah Lacharité and Kenneth G. Paterson. A note on the optimality of frequency analysis vs. ℓ_p-optimization. Cryptology ePrint Archive, Report 2015/1158, 2015. http://eprint.iacr.org/2015/1158.

[32] Kevin Lewi and David J Wu. Order-revealing encryption: New constructions, applications, and lower bounds. In *CCS*, 2016.

[33] Microsoft. DBCC for SQL server. https://msdn.microsoft.com/en-us/library/ms188796.aspx, 2016.

[34] Microsoft. Microsoft SQL Server caching mechanisms. https://msdn.microsoft.com/en-us/library/cc293623.aspx, 2016.

[35] Mongo. Monitoring in MongoDB. https://docs.mongodb.com/manual/administration/monitoring/, 2016.

[36] Replica set oplog. https://docs.mongodb.com/manual/core/replica-set-oplog/.

[37] Microsoft transparent data encryption. https://msdn.microsoft.com/en-us/library/bb934049.aspx.

[38] Moni Naor and Vanessa Teague. Anti-persistence: History independent data structures. In *STOC*, 2001.

[39] Muhammad Naveed, Seny Kamara, and Charles V Wright. Inference attacks on property-preserving encrypted databases. In *CCS*, 2015.

[40] Muhammad Naveed, Manoj Prabhakaran, and Carl A Gunter. Dynamic searchable encryption via blind storage. In *S&P*, 2014.

[41] Oracle transparent data encryption. http://www.oracle.com/technetwork/database/options/advanced-security/index-099011.html.

[42] Antonis Papadimitriou, Ranjita Bhagwan, Nishanth Chandran, Ramachandran Ramjee, Andreas Haeberlen, Harmeet Singh, Abhishek Modi, and Saikrishna Badrinarayanan. Big data analytics over encrypted datasets with Seabed. In *OSDI*, 2016.

[43] Vasilis Pappas, Fernando Krell, Binh Vo, Vladimir Kolesnikov, Tal Malkin, Seung Geol Choi, Wesley George, Angelos Keromytis, and Steve Bellovin. Blind Seer: A scalable private DBMS. In *S&P*, 2014.

[44] MySQL performance schema. http://dev.mysql.com/doc/refman/5.7/en/performance-schema-statement-tables.html.

[45] Rishabh Poddar, Tobias Boelter, and Raluca Ada Popa. Arx: A strongly encrypted database system. Cryptology ePrint Archive, Report 2016/591, 2016.

[46] Raluca Ada Popa, Catherine Redfield, Nickolai Zeldovich, and Hari Balakrishnan. CryptDB: Protecting confidentiality with encrypted query processing. In *SOSP*, 2011.

[47] Raluca Ada Popa, Emily Stark, Jonas Helfer, Steven Valdez, Nickolai Zeldovich, M. Frans Kaashoek, and Hari Balakrishnan. Building web applications on top of encrypted data using Mylar. Cryptology ePrint Archive, Report 2016/893, 2016.

[48] Raluca Ada Popa, Emily Stark, Steven Valdez, Jonas Helfer, Nickolai Zeldovich, and Hari Balakrishnan. Building web applications on top of encrypted data using Mylar. In *NSDI*, 2014.

[49] Raluca Ada Popa, Nickolai Zeldovich, and Hari Balakrishnan. Guidelines for using the CryptDB system securely. Cryptology ePrint Archive, Report 2015/979, 2015.

[50] Postgres. The statistics collector. https://www.postgresql.org/docs/current/static/monitoring-stats.html, 2016.

[51] MySQL query cache. https://dev.mysql.com/doc/refman/5.7/en/query-cache.html.

[52] Thomas Ristenpart and Scott Yilek. When good randomness goes bad: Virtual machine reset vulnerabilities and hedging deployed cryptography. In *NDSS*, 2010.

[53] Alon Shalita, Brian Karrer, Igor Kabiljo, Arun Sharma, Alessandro Presta, Aaron Adcock, Herald Kllapi, and Michael Stumm. Social Hash: An assignment framework for optimizing distributed systems operations on social networks. In *NSDI*, 2016.

[54] Skyhigh Networks. https://www.skyhighnetworks.com.

[55] The slow query log. http://dev.mysql.com/doc/refman/5.7/en/slow-query-log.html.

[56] Dawn Xiaodong Song, David Wagner, and Adrian Perrig. Practical techniques for searches on encrypted data. In *S&P*, 2000.

[57] Apache Spark. Monitoring and instrumentation. http://spark.apache.org/docs/latest/monitoring.html, 2016.

[58] Patrick Stahlberg, Gerome Miklau, and Brian Neil Levine. Threats to privacy in the forensic analysis of database systems. In *SIGMOD*, 2007.

[59] Verizon data breach incident report. https://regmedia.co.uk/2016/05/12/dbir_2016.pdf, 2016.