Adversarial Models in Encrypted Search

**Authors:** Crypto Team

**Request from:** Product

**Date:** July 15, 2022

The purpose of this document is to give a high level introduction to the ways in which the security of encrypted search is evaluated. Note that, here, we are only concerned with the confidentiality of the data and queries and not their integrity.

There are several threat models one can consider when analyzing the security of an encrypted database. Roughly speaking, one can think of database adversarial models along two dimensions: (1) what the adversary has access to, e.g., the database, disk, memory or the entire server; and (2) how long it has access to it, e.g., a snapshot in time, multiple snapshots in time or continuously. These two dimensions lead to different threat models but in this document, we summarize the most important ones.

# Adversarial Models

### Disk-Level Snapshot Adversaries

A disk-level snapshot adversary has access to a snapshot of the server's disk. This includes the database/collection, its indexes, various logs and virtual memory swap space. This captures data breaches that occur due to disk theft in data centers or to lost or stolen laptops. Disk-level adversaries are very easy to protect against using disk encryption since they only access data at rest.

### Database-Level Snapshot Adversaries (a.k.a. Ideal Snapshot Adversaries or DBA Adversaries)

A database-level snapshot adversary has access to the database/collection in memory and on disk. A multi-snapshot adversary is one that can get multiple snapshots at different instants in time. An atomic (multi) snapshot adversary is one that gets snapshots only after an operation terminates but never during the execution of an operation. This captures malicious DBAs and is useful for customers that wish to use end-to-end encryption to enforce "separation of duties". In the cryptography research literature a database-level atomic multi-snapshot adversary is usually simply referred to as a snapshot adversary.

### Memory-Level Snapshot Adversaries (a.k.a. Real Snapshot Adversaries)

A memory-level snapshot adversary has access to the entire memory and disk of the server at a particular point in time. This means that at that instant, it can access the entire database/collection, any keys stored in memory, all the caches and all the logs. Memory-level adversaries are more powerful than disk- and database-level adversaries because they have access to more information. A multi-snapshot adversary is one that can get multiple snapshots at different instants in time. An atomic (multi) snapshot adversary is one that gets snapshots only after an operation terminates but never during the execution of an operation.

### Server-Level Persistent Adversaries (a.k.a. Persistent Adversaries)

A server-level persistent adversary has access to the entire server on a continuous basis. This means that it can access memory, disk, the network, registers and trusted execution environments. Server-level persistent adversaries are more powerful than memory-level snapshot adversaries because they have access to more information and for a longer period of time. This means that, in addition to the contents of memory, they also have access to all the queries ever made to the database. This captures settings where the entire server has been compromised for a long period of time.

# Leakage Analysis

All practical encryption schemes leak some information about the plaintext to the adversary. For example, standard encryption (i.e., non-searchable) leaks the message’ s length. Similarly, practical encrypted search solutions reveal *some* information about data and/or queries, however, the goal is to minimize this leakage while not affecting performance.

When analyzing the security of an encrypted search solution, we will describe precisely what information gets leaked about the data and queries to the different adversaries outlined above. We describe leakage based on the notion of a *leakage pattern* which is a formal description of a specific type of leakage. Most solutions reveal more than one leakage pattern, so we refer to the collection of leakage patterns revealed by a solution as its *leakage profile*.

Consider, for example, an encrypted search solution based on deterministic encryption that supports point queries (i.e., exact match) and document insertions. In the following, we consider two types of adversaries: a snapshot adversary and a persistent adversary.

In the (single) snapshot model, we consider one leakage function that captures what gets leaked about the database at the time a snapshot occurs. We call this the *snapshot leakage* and, for the case of deterministic encryption, it is:

Λ\_det = L\_SN > (freq, vsize),

by which we mean that the scheme (i.e., deterministic encryption) has leakage profile Λ\_det with snapshot leakage L\_SN that leaks *at least* (freq, vsize), where *freq* and *vsize* are leakage patterns defined as follows:

1. freq(DB) returns the number of times each value of the database appears,
2. vsize(DB) returns the size of every value in the database.

Note that deterministic encryption leaks considerably more than freq and vsize because it also reveals correlations between ciphertexts. For example, when applied naively to keyword documents, deterministic encryption also leaks the location of the keywords in the file which can reveal additional information, e.g., if the files contain data about US cities and states, then seeing two equal ciphertexts next to each other suggests they might encrypt the strings “New York”, “New York”. In the case of FLE 1.0, using deterministic encryption leaks correlations that can be inferred from the field names, e.g., if the database stores driver license information that includes fields for eye color and hair color, then seeing the same ciphertexts in these fields suggest they might encrypt “black”.

In the persistent model, we will consider three leakage functions: (1) *setup leakage* which captures what gets leaked when the database is first encrypted and sent to the server; (2) (point) *query leakage* which captures what gets leaked when the database is queried; and (3) *insertion leakage* which captures what gets leaked when a document is inserted. The leakage profile of this solution is then:

Λ\_pers = (L\_S, L\_Q, L\_I) > ((freq, vsize), (qeq, rlen), (freq, vsize)),

by which we mean that the scheme has leakage profile Λ\_pers with setup leakage L\_S > (freq, vsize), query leakage L\_Q > (qeq, rlen) and insertion leakage L\_I > (freq, vsize). Here *qeq* and *rlen* are leakage patterns defined as follows:

1. qeq(DB, q\_1, …, q\_m) indicates which queries are for the same values.
2. rlen(DB, q\_1, …, q\_m) returns the response length for each query q\_i.

As illustrated above, a given solution usually has different leakage profiles against different adversaries. When thinking about snapshot adversaries, the leakage profile of a solution can also change depending on the moment the snapshot was taken as we will see below.

#### System Leakage

So far, our analysis has focused on the leakage of the cryptographic scheme itself (in our example deterministic encryption) but in practice the scheme is executed in a larger environment that could add to this leakage. For example, if a scheme is integrated into a database system then we might have to consider how the database and/or the operating system impacts what the adversary sees. More concretely, if the database and OS cache the search tokens, then the adversary will have access to more leakage than what is captured by the cryptographic scheme’s leakage profile.

One of the benefits of analyzing schemes in the persistent model is that leakage profiles against persistent adversaries coincide with system leakage, i.e., they capture all the possible leakage an adversary can receive. This is not necessarily true, however, for leakage profiles against snapshot adversaries.

Consider, for example, a scheme based on deterministic encryption, a set of queries (q\_1, …, q\_m) and a memory-level snapshot adversary A. Furthermore, let’s assume that the server caches information related to at most the last 5 operations.

It follows that after the ith operation, A will receive not only the snapshot leakage

L\_SN(DB) >= (freq(DB), vsize(DB))

from executing the cryptographic operations but will also receive the query leakage

L\_Q(DB, q\_{i-5}, …, q\_i) >= (qeq(DB, q\_{i-5}, …, q\_i), rlen(DB, q\_{i-5}, …, q\_i)),

from the caches.

Focusing on a specific query, however, we can see that the leakage revealed by the cache about the query will only be temporarily available. More concretely, if we focus on q\_2 then we can see that query leakage about q\_2 will only be included in snapshots that occur between the 2nd and the 7th query. Starting from the 8th query, the adversary will not be able to learn anything about q\_2 since it will not be included in any snapshots.

# Leakage Attacks

A leakage attack is an algorithm that tries to exploit the leakage of an encrypted search solution to recover information about the underlying data and/or operations. Leakage analysis as described above can formally describe leakage but it does not predict whether this leakage can be exploited or not. That is, whether there exists an attack that can efficiently recover useful information about the data and/or queries from the leakage.

Because we don’t have a “theory of leakage” to tell us whether a leakage profile is exploitable or not, we rely on cryptanalysis. In other words, we try to design attacks against every leakage profile we are aware of to see which ones stand up to scrutiny. Of course, the absence of an attack on a leakage profile L does not mean that L is secure; it could simply be that no one has figured out how to attack it yet. But cryptanalysis helps us gain insight into the potential weaknesses of certain leakage profiles and settings under which they can be vulnerable.

The reason it is so difficult to ascertain whether a leakage profile is secure or not is because it depends, not only on the leakage, but also on the characteristics of the workload and on the adversary’s auxiliary information (i.e., what it may already know about the data and queries).

This is important to really understand cryptanalytic results in encrypted search. Often (but not always) when a leakage attack is presented, favorable assumptions are made about the workload and the adversary’s auxiliary information so that the attack “works”. So it is important to understand the real-world performance of an attack on real workloads and under realistic assumptions. To facilitate this, we recently developed a tool called Leaker (<https://eprint.iacr.org/2021/1035.pdf>) that can be used to run the state-of-the-art attacks against a given leakage profile using realistic workloads (including your own) and under realistic assumptions.

More long term, we are also working on a theoretical framework to fully analyze the security of a leakage profile. Such a framework would allow us to completely understand the security properties of a leakage profile.