Practical Volume-Based Attacks on Encrypted Databases

Stephanie Wang
Rishabh Poddar
Jianan Lu
Raluca Ada Popa

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2019-50
http://www2.eecs.berkeley.edu/Pubs/TechRpts/2019/EECS-2019-50.html

May 16, 2019

Copyright © 2019, by the author(s).
All rights reserved.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission.

Practical Volume-Based Attacks on Encrypted Databases

by Jianan Lu

Research Project

Submitted to the Department of Electrical Engineering and Computer Sciences, University of California at Berkeley, in partial satisfaction of the requirements for the degree of **Master of Science, Plan II**.

Approval for the Report and Comprehensive Examination:

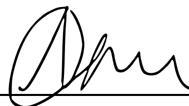
Committee:

Professor Raluca Ada Popa
Research Advisor

May 16, 2019

(Date)

* * * * * * *



Professor Alessandro Chiesa
Second Reader

2019.05.14

(Date)

Abstract

Databases are the key component of most computer systems today. Because of the valuable and sensitive data they store and process, these database systems have become the primary target of digital attacks. For example, confidential information (*e.g.*, social security number, home address) of over 140 million people is leaked in 2017 from Equifax, one of US's largest credit reporting companies.

This prevalence of database breaches spurs more interests towards building secure databases in both academia and industry. There have been a set of proposed works that can protect data using advanced encryption schemes or hide query access patterns, albeit at some performance cost. However, recent work has also shown that volume leakage is a significant vulnerability that can be exploited to reconstruct the entire database even when using state-of-the-art designs with strongest security guarantees.

In this work, we present new attacks for recovering the content of individual user queries, assuming no leakage from the system except the number of results. Unlike previous volume-based attacks that rely on assumptions either too stringent or unrealistic for many real-world systems, our attacks directly leverage real application semantics running on top of these database systems. The key insight is that, by exploiting the behavior of specific applications, one can immediately have an attack without making further assumptions like prior work does about the underlying system.

Contents

Contents 2
1 Introduction 4
2 Related Work 6
2.1 Cryptographic schemes and systems 6
2.2 Related attacks 6
3 Attack model 8
3.1 System Model 8
3.2 Application model 10
3.2.1 File injection 10
3.2.2 Query replay 10
4 Our Attack 12
4.1 Overview 12
4.2 Attack Algorithm 12
5 Evaluation 15
5.1 Simulation Using Encrypted Email Databases 15
5.1.1 Setup 15
5.1.2 Evaluation 15
5.2 Case Study of Gmail Inbox Search 17
5.2.1 Setup 17
5.2.2 Key Characteristics 18
5.2.3 End-to-end Attack 18
5.2.4 Other Applications 19
6 Mitigations 20
6.1 Disable File Injection 20
6.2 Prevent File Measurement 20
6.3 Block Query Replay 20
7 Conclusion 21
References 22

Acknowledgements

I am really thankful to my advisor, Professor Raluca Ada Popa, for guiding me through this project and teaching me how to conduct research in system security. I have learned a lot from her, to name a few, as how to think of security problems, to tackle technical challenges, to position one's work in a convincing way, and to give good deliverable. Without her great advice and support, I would not be able to make this work or continue to pursue my academic interest in graduate school!

I am also super grateful to my collaborators, Rishabh Poddar and Stephanie Wang. When I first joined this project, I had little knowledge of computer networks and no experience in launching attacks against real systems. They helped me to design my experiments in a step-by-step manner, taught me a lot of technical skills, and answered a lot, if not too many, of my questions.

I am also very thankful to Wen Zhang who taught me how to use various network attack tools. His assistance directly helps me make positive progress in this work.

Finally, a big thank-you to Professor Alessandro Chiesa for generously giving me great feedback on this report!

1 INTRODUCTION

In recent years, the interest towards building databases with stronger security guarantees has increased drastically. One powerful technique is to add a layer of encryption that can permit the querying of encrypted data. Even when attackers break in and compromise the database server, they cannot steal the data or read its content without having access to the correct decryption keys. A number of practical encrypted database systems are proposed [3, 10, 12, 17, 19, 28, 40, 47, 50, 58] and some have already been integrated into existing infrastructures. For example, Microsoft deployed a new feature, called Always Encrypted [40], for Azure SQL Database or SQL Server databases in 2017. This service gives data owners the freedom to encrypt sensitive columns of their own data such that unauthorized parties who no more have direct access to the data can still run services on it.

The majority of these schemes reply on property-preserving encryption [6, 7, 33, 37, 46] or searchable encryption [8, 10, 11, 13, 27, 31, 34, 36, 43, 44, 54, 55]. However, most of them leak *query access patterns*. At a high level, given a query, the query access pattern is the set of records that satisfy the query condition. Consider the example of an email application: a user issues a search query for a keyword over their emails. The mail server typically stores an inverted index (also called a *secondary index*) for each user's mailbox, which maps a keyword to a list of emails. When fetching the results of a queried keyword, most of these schemes leak to the attacker the set of email identifiers that match the keyword (*i.e.*, the access patterns of the query), even though the email bodies remain encrypted. A set of recent works [2, 9, 14, 21, 26, 29, 30, 32, 35, 38, 49, 60] has shown that such access patterns leak significant information to a compromised server. In the context of the email example, they were able to reconstruct the keyword that the user searches for, as well as email contents.

Many of these works discuss *oblivious protocols*, such as ORAM (Oblivious RAM) [24, 57] or PIR (Private Information Retrieval) [20], as a solution to this leakage. These schemes hide access patterns: even an attacker eavesdropping at the database server does not learn which records are accessed and hence cannot infer the queried keywords. These schemes are often regarded as giving a very strong security guarantee, the main downside largely being their slow performance.

However, in seminal work, Kellaris *et al.* [32] showed that even strongest schemes today, the ones that enable encryption and provably conceal access patterns, still leak *result count of queries*. The attacker neither knows the content of individual queries (which are encrypted), nor does it learn which documents were returned in response (*i.e.*, access patterns remain hidden). Instead, it only observes the total number of returned records, or the *volume* of query results. This can allow attackers to reconstruct the *database counts*, *i.e.*, the number of documents in the database containing each particular value. The primary contribution of Kellaris *et al.* was to show that volume-based attacks were possible. Because their methods depend on certain assumptions about the queries (*e.g.*, amount, type, distribution), their attacks are limited in scope, if not yet practical.

In this project, we improve upon the state-of-the-art and show that volume-based attacks are more likely to be practical. Unlike Kellaris *et al.* and other literature on volume-based attacks, our work targets at the application layer *without making any assumptions about user queries or the underlying data*. In particular, we focus on real applications that allow users to search for keywords over a *secondary index*, a common data structure in database systems that maps keys to a set of matching records. In the encrypted database literature, this corresponds to the model of searchable encryption schemes [8, 10, 11, 13, 27, 31, 34, 36, 43, 44, 54, 55]. Our aim is to recover the content of individual queries that search for a specific keyword in the database.

Our key insight is that, by directly leveraging real application semantics, one can immediately have an attack without making any further assumption about the underlying database system. Furthermore, it allows our attack to be more efficient and thus eminently practical.

By and large, real-world applications today leak far more information than just the volume of results. However, privacy-conscious services have begun deploying sophisticated schemes to plug traditional sources of leakage, including access patterns (*e.g.*, the Signal messaging service [1, 39]). The takeaway of our work is that as practitioners take steps for enhancing the privacy guarantees of their applications in future, they must also account for the leakage of result volumes. Application-specific behavior that facilitates easy exploitation of this leakage (as demonstrated by our attacks) must be patched.

In §2, we discuss important works related to encrypted database systems and volume-leakage attacks. §3 describes our threat model in more detail. We survey 11 representative applications and identify two key characteristics that can be exploited by attackers—(i) file injection, and (ii) automatic query replay. In particular, Gmail inbox search fits our setting seamlessly, and satisfies both the aforementioned properties. Given these attacker abilities, we present an attack algorithm that is able to reconstruct user queries with 100% confidence on secondary indices in §4. Subsequently, we evaluate our attacks on an encrypted database using the Enron email dataset in Section 5.1. We also perform an end-to-end attack on the Gmail web client by simulating a server-side adversary in Section 5.2. Our attack on Gmail completes within a matter of minutes, demonstrating the feasibility of our techniques. Finally, we introduce potential mitigations in §6, and end this report in §7.

2 RELATED WORK

To access or compute on encrypted data, the community has developed a rich set of cryptographic schemes and protocols, as well as encrypted database systems. Each of these schemes and systems makes various tradeoffs between information revealed to the server, supported functionality, and performance. A recent set of attack papers study the information an attacker can obtain from these schemes and systems, referred to as *leakage-abuse* attacks by Cash *et al.* [11]. Most of the attacks in this category leverage leakage from data relations or access patterns, and very few works target at these systems relying only on volume leakage, as our work does.

2.1 Cryptographic schemes and systems

There are a multitude of ways to access or compute on encrypted data, such as property-preserving or property-revealing encryption [6, 7, 33, 37, 46] or searchable encryption [8, 10, 11, 13, 27, 31, 34, 36, 43, 44, 54, 55]. For a comprehensive survey, see [53]. We take ORAM and PIR as two representative examples below and will discuss them in more detail in Section 3.1. It is important to note that our work directly applies to all these systems that leak the volume of results, not limited to ORAM or PIR.

Oblivious RAM techniques [24, 57] and Private Information Retrieval (PIR) [20] schemes enable a client to access data items stored at the server without the server knowing the query requested. These two types of schemes consider different models and employ different techniques, but ultimately, the goal of both is to hide the query from the server.

Many works leverage ORAM for different purposes. For example, ObliviStore [56] and CURIOUS [5] show how to use ORAM for cloud storage. TaoStore [52] shows how to support asynchronicity in multi-user cases. These systems leak the volume of results to the server. Roche *et al.* [51] propose an ORAM scheme (called vORAM) that supports variable-sized data blocks by including them within an ORAM node (or bucket) on the same path. While such a scheme confers some degree of hiding, it limits the amount of data that can be included on a path in this way, say L files, and the attacker sees how many ORAM paths are fetched. Hence, the attacker can estimate the number of results with an error margin of L. In the database setting, this error margin can be made relatively small, because the database fetches the rows that match the keyword (not just the row identifiers), and these cannot all be stored on the same path. Moreover, Naveed [42] demonstrates that, in general, extending ORAM schemes to hide the number of query results is (for a large fraction of queries) slower than streaming the database through the client.

Some works [4, 45, 59] build SQL databases or keyword indices on top of PIR. For example, to perform an index search for a keyword k, the client performs PIR retrievals to traverse the index and select every value in the index. The server does not know which data items were fetched, but it still sees the number of results.

2.2 Related attacks

When considering the amount of leakage that attacks exploit, there are at least three categories: attacks exploiting data relations, attacks exploiting access patterns, and attacks exploiting result set size but not access patterns or data relations. The last category is the most challenging because the attacker needs to work with the least amount of information. At the same time, this category is also the least studied. Our attack is in this last category, and we now discuss other volume-based attacks.

Cash *et al.* [10] point out that if an attacker knows the exact number of times a keyword appears in a victim's documents, and if that result size is unique to this keyword, the attacker can identify the keyword when seeing the result size. In comparison, our attack does not assume the attacker

knows the frequency of each keyword in a victim's index—indeed, when attacking a specific user in the email application, the attacker often does not have access to the victim's mailbox and does not know these counts. Moreover, many keywords don't have unique counts (*e.g.*, 99% words in the Enron dataset, Section 5.1), making this attack not work for these keywords. Our attack can recover 100% queries in a dictionary in realistic scenarios without access pattern information.

Kellaris *et al.* [32] also show how an attacker can reconstruct contents of a field in the database given only the result size, but their attack differs from ours in assumptions and target. First, Kellaris *et al.* assume that (1) the user makes range queries that are *uniformly* distributed on that column, a property on which their algorithm relies crucially; and (2) the user makes $O(N^4 \log N)$ queries where N is the size of the domain. Such a large number of queries is infeasible for the attacker to observe in many settings. Very recently, Grubbs *et al.* [25] improved upon the results of Kellaris *et al.* by demonstrating attacks that do not make assumptions on the distribution of queries, as long as all possible range queries are issued. For queries drawn from a uniform distribution, their attack requires $O(N^2 \log N)$ queries.

In contrast, our attack requires only a single query to be issued by the user, followed by $O(\log |D|)$ replays, which is often less than 10 in number (§5). Our attack also makes no assumptions about the query distribution—assuming a uniform range query distribution is not realistic for many applications. On the other hand, unlike the aforementioned works, our attack requires the ability to inject and replay queries. We validate that this can be achieved in realistic scenarios in (Section 3.2). A second difference is that the aforementioned attacks reconstruct the database (out of range queries), but not individual query keywords; we reconstruct queries, but do not target the overall database. However, we note that a similar reconstruction follows as a direct consequence of our attack, where the original counts for each keyword could be determined if queries for all possible keywords are issued.

3 ATTACK MODEL

In this section, we discuss the generic system model (Section 3.1) and the application model (Section 3.2) that is vulnerable to our attacks. We present the three key attack assumptions: (1) that the system leaks volume, (2) that the application allows data injection, and (3) that the applications automatically replays queries under certain conditions. We then demonstrate the validity of our assumptions by studying a number of concrete instances for both the system and application models. In particular, we examine 11 popular web applications that allow users to issue keyword search queries over an inverted index—we find that (i) all 11 applications allow attackers to inject data into the victim user's index; and (ii) 5 of the 11 applications also replay queries automatically without user intervention.

3.1 System Model

We consider systems in which an untrusted server (the adversary in our setting) maintains a secondary index in an encrypted database. The index maps a keyword to a list of documents or database rows (referred to as *files*, henceforth) that the keyword appears in and is stored on the server for query efficiency. Whenever the application proxy or the client queries the index for a keyword, the user receives the corresponding list of files containing the keyword. We assume that the query's execution reveals no information to the adversary except the number of results. That is, there is only *volume leakage*.

More formally, similar to Kellaris *et al.* [32], we define a database $\mathcal{D}$ as a set of records that associate keywords with the files from a collection $\mathcal{F}$ that the keywords appear in:

$$\mathcal{D} = \{(w, f) : w \in f, f \in \mathcal{F}\}$$

A query for word w is a function q_w (where w is private) that maps $\mathcal{D}$ to a list of matching files in $\mathcal{F}$:

$$q_w(\mathcal{D}) = \{f : (w, f) \in \mathcal{D}\} .$$

An implementation of such a database may internally use one layer of indirection, so that the first query returns a list of file pointers, or indices into $\mathcal{F}$, and the subsequent queries are used to fetch the file contents from $\mathcal{F}$.

The adversary's goal is to identify the private keyword w using only the size of the result set $|q_w(\mathcal{D})|$.

Examples. We now illustrate the relevance of volume-based attacks by discussing concrete examples of cryptographic systems that only leak the *volume of results* to attackers. In particular, we consider a client-server model where the database stored at the server is encrypted using sophisticated techniques that also hide access patterns, and the server maintains a secondary index over the encrypted data.

ORAM-based systems. In the ORAM model (Figure 1, left), the server stores the secondary index in an ORAM instance, with a trusted proxy containing the ORAM key. An (optional) application server lies in front of the proxy, and clients access the system via the application server. As is the case in many database-backed applications, the application server also implements access control over the data. Now we consider the case where a database administrator backs the database with ORAM, which hides access patterns from the server in addition to the result contents. However, even with a guarantee of this strength, volume leakage is possible for a passive attacker because the *size* of the result contents is not hidden. In addition, even if a layer of indirection is used, so that the first query only returns a list of file pointers, the number of files returned can still be measured by recording the number of subsequent queries made.

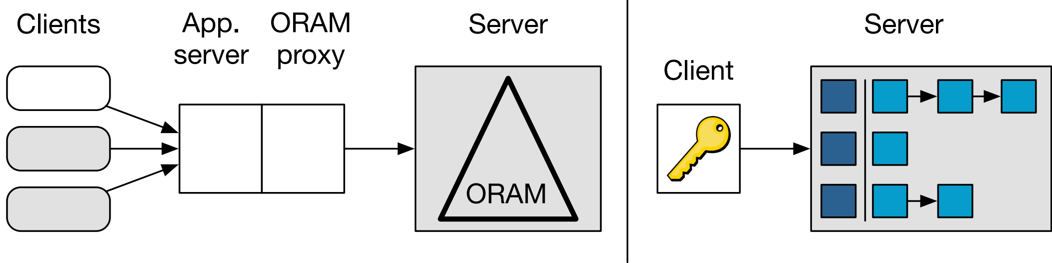


Fig. 1. (Left) ORAM-based system model; both the server and a subset of the clients are untrusted (shaded). (Right) PIR-based model; the server is untrusted (shaded).

PIR-based systems. In the PIR model (Figure 1, right), an untrusted server owns the database and maintains a secondary index on it for fast access. In this case, the server sees all the data, as its role is to maintain and serve publicly available data. There can be multiple clients here too, but each client has an independent interaction with the server so it suffices to focus on a single client. The untrusted server answers user queries in which the requested is private [20]. However, since the data is publicly available, the untrusted server can easily learn the size of the query results.

It is important to note that any sophisticated cryptographic schemes that leak volume are vulnerable to our attacks. ORAM and PIR are just two examples of them. The takeaway is that content encryption is not sufficient to prevent volume leakage, as we've seen in these examples.

Application	Type of queries	File injection strategy	No. of replays observed in 10 minutes
Gmail	Email keywords	Send emails to victim	6
Facebook	Names, post keywords	Create posts in a group page	4
Dropbox	File keywords	Upload files to a shared folder	1
Google Doc	File keywords	Upload files to a shared folder	1
iCloud Mail	Email keywords	Send emails to victim	1
Twitter	Hashtags	Post tweets with hashtags	0
Piazza	Post keywords	Create posts in a class	0
Slack	Names, message keywords	Send messages to a group channel	0
Skype	Names, message keywords	Send messages to victim	0
Yahoo Mail	Email keywords	Send emails to victim	0
Outlook Mail (Hotmail)	Email keywords	Send emails to victim	0

Fig. 2. An empirical assessment of 11 popular web applications. In each case, we list the type of query made by the user, how the attacker can influence the result of this query by application-specific injection, and the number of replays observed. To measure replay, all server responses are dropped for 10 minutes, and we report the number of duplicate queries made within 10 minutes.

3.2 Application model

We assume an application model based on the behavior of actual applications that rely on a secondary index. The model consists of two key assumptions: the ability to inject data into a user's index and the ability to replay a user query without the involvement of the user. We argue that these two assumptions are in fact often inherent to the application. As evidence, we survey a variety of popular web applications that rely on a secondary index and validate both assumptions in 5 out of the 11 surveyed. As an example, Gmail inbox search fits our setting seamlessly, and satisfies both the aforementioned properties. The user can search for all emails that contain a keyword, an attacker can inject data by simply sending the user an email with a specific keyword, and queries are automatically replayed by the application when the server's response is delayed. Finally, we describe how these assumptions together make it difficult to detect our attack.

3.2.1 File injection. First, we find that many applications inherently allow *other* users to *inject* application data into a victim user's index. This property of applications has also been noted in prior work [10, 38, 60]. In our setting, it allows the attacker to potentially influence the volume of results returned by a query. For example, to inject into Gmail inbox search, the attacker sends the victim user an email. Injection is especially easy if there is a secondary index that is shared. To inject an entry for a hashtag in Twitter, the attacker uploads a post with that hashtag; in Slack, the attacker simply sends a message. The ability of the attacker to inject in such applications is fundamental because they are inherently designed for multiple users to interact and share data.

3.2.2 Query replay. Second, we assume that the attacker can replay a user's query a finite number of times. While this assumption is certainly not universal across applications that rely on a secondary index, we find that it is surprisingly common, with many applications replaying queries automatically in the background *without user intervention.* This is because many applications are written to handle transient errors transparently, to put as little burden on the user as possible. In

particular, an application that wants to provide a seamless experience when network connectivity is spotty may retry a query automatically if the response is too slow. Indeed, the HTTP/1.1 RFC [18] specifies that "When an inbound connection is closed prematurely, a client MAY open a new connection and automatically retransmit an aborted sequence of [idempotent] requests." A compromised server can force this behavior by simply dropping its HTTPS responses, triggering an automatic replay.

Examples. To show that these assumptions are realistic, we surveyed 11 applications, including Gmail, Twitter, and Facebook, and tested for the ability to inject data and replay queries for a target user (Figure 2). To test for injection, we examined the application functionality to determine whether the attacker could inject data into an index searchable by the user. To measure the number of query replays, we drop responses from the server without disturbing the original connection, record all network traffic from the application client, and count the number of duplicate queries that appear within 10 minutes. We find that for all applications, injection is possible, although sometimes only if the attacker and the user share some index (*e.g.*, they are both members of a public Facebook group). We also find that 5 of these applications have query replay.

Upon further investigation of these 5 applications, we find that all retry queries automatically, though the rate of retries varies. Two applications, Gmail and Facebook, retry the query repeatedly. The remaining three, Dropbox, Google Drive, and iCloud Mail, retry the query once. The number of retries is important because the greater the number of retries, the easier it is for the attacker to identify the query. Nevertheless, as we show in Section 5.1.2, even a single replay is sufficient for significantly pruning the space of query possibilities, and, in many cases, for mounting the attack feasibly.

Avoiding detection. Because the attack relies on injection visible to the user, one practical concern in launching the attack is avoiding detection. Fortunately, in many settings, our attack is *difficult to detect before it completes*. The reason is that once the user issues a query, the attacker can continue to block the responses from the server, causing the web application to retry queries until the attack completes.

We verified this behavior with Gmail: no results are returned to the user during the attack, and to the user it simply appears that he has a bad network connection. That is, once the user initiates the query, the attack will complete without further actions from the victim user.

It is possible that the user later sees the injected emails and realizes from the synthetic content that he is under attack, but this happens only *after the attack completes*. Further, we note that although services like Gmail may strip suspicious HTML elements during email preprocessing, we can still use style formatting to avoid showing the injected content to the user, to reduce suspicion. The rest of the email could show content that is more user-friendly, *e.g.*, an ad. It is further unlikely for spam filters to detect the injected emails, since the attack targets a specific user. This is just one example, but it illustrates the numerous ways that an attacker could inject data in a way that is difficult to detect before the query is reconstructed.

4 OUR ATTACK

Given the attacker abilities discussed in Section 3.2 in concert with volume leakage, we present and analyze a file-injection attack to recover a user's query on a secondary index. We show that this attack can be launched on the generic encrypted database systems described in Section 3.1, as long as the attacker can view the *number of results* returned.

4.1 Overview

At a high level, our attack works by searching on the keyword universe through multiple rounds of user query replay. By recording the result counts between rounds, the attacker can narrow down the keyword search space by a constant factor per round.

The attacker uses file injection to influence the result count between rounds. During each round, the attacker constructs files from the keyword search space and injects the files into the user's index. The response for each round will then contain some number of injected files. The attacker can use the new result count to determine the number of files injected after the previous round. In this way, the attacker can determine which subset of the search space contains the user's query.

The setup of the attack is as follows: A user queries q_w on a database $\mathcal{D}$, as defined in Section 3.1. The response is the set of matching file contents, $q_w(\mathcal{D}) = \{f_1, \ldots, f_n\}$. The goal of the attack is to recover w, using only $n = |q_w(\mathcal{D})|$, the number of files returned.

4.2 Attack Algorithm

Algorithm 1 provides pseudocode for our attack RecoverQuery. In more detail, the attacker first records the user query's q_w and the number of files returned, n_0. n_0 is the number of files that already matched to w prior to the attack. This enables the attacker to differentiate user-uploaded files from injected ones.

Next, the attacker proceeds in rounds to reduce the keyword search space. He chooses an initial dictionary D_0, a set of words that might contain w, and a parameter k. During each round j, the attacker divides D_j, the current dictionary during round j, into k equal partitions. He injects k files into the server and distributes the words among them as follows: *If a word appears in the i-th partition, he adds the word to exactly i out of the k files.* Hence, if a word appears in the k-th partition, the attacker adds this word to all k files.

The attacker then replays the user's query q_w on the updated database and records the number of files returned, n_j. Assuming that the attacker can block updates to the secondary index, the number of files injected since the previous round is then $i^* = n_j - n_{j-1}$. Thus, w must have been assigned to the i^*-th partition during round j. The attacker repeats this in rounds, each time using the i^*-th partition as the new dictionary, until $|D| = 1$.

This attack converges in a bounded number of rounds since each round is guaranteed to reduce the dictionary size. Furthermore, for a high enough k and a small enough D, the number of rounds, *i.e.*, the number of times the attacker has to replay the user's query, is quite low. We formalize this in the following claim:

Claim 1. *For any dictionary D and for any word $w \in D$, let q_w be a private query for w, and k be the number of partitions. Then,* RecoverQuery(q_w, k) *returns w after $\lceil \log_k |D| \rceil$ rounds.*

Proof. Consider the j-th round of the attack, which searches a dictionary D_j that contains w. w is guaranteed to match to a partition of the dictionary that has size $\leq |D_j|/k$. Thus, round $j+1$ of the attack will search a dictionary of size at most $|D_j|/k$ that also contains w. The algorithm repeats until the dictionary has size one. At this point, RecoverQuery returns the only word in the dictionary, w. Thus, it takes $\lceil \log_k |D| \rceil$ rounds to complete the attack, where D is the initial dictionary. □

Algorithm 1 Pseudocode for the base attack. q_w is a private query for a word w on a database $\mathcal{D}$. Each round of the attack partitions the search space by k.

```
1: procedure RecoverQuery(q_w, k)
2:     𝒟 ← the initial database
3:     D ← keyword universe
4:     n ← |q_w(𝒟)|
5:     while |D| > 1 do
6:         for i in [1, ..., k] do
7:             F_i ← an empty file
8:             D_i ← an empty dictionary
9:         end for
10:        for index in [1, ..., |D|] do
11:            w ← D[index]
12:            i ← ⌊ index / (|D|/k) ⌋
13:            Append w to i unique files in F
14:            Add w to dictionary D_i
15:        end for
16:        𝒟 ← InjectFiles(𝒟, F)
17:        n′ ← |q_w(𝒟)|
18:        i ← n′ − n
19:        D ← D_i
20:        n ← n′
21:    end while
22:    return D[0]
23: end procedure
```

The attacker must also inject a significant number of files. We show that the number of files, along with the file size, measured in number of words, is not too large.

Claim 2. *For any dictionary D and for any word $w \in D$, let q_w be a private query for w and k be the number of partitions. Then, the total number of files injected by* RecoverQuery(q_w, k) *is* $k \lceil \log_k |D| \rceil$.

Proof. During a single round of the attack, the words in the i-th partition of the dictionary must be distributed among i unique files, so that the number of results for the query q_w during the next round will be increased by i if w was in that partition. The maximum value for i is k, the number of partitions. Therefore, each round requires injecting at least k files. There are $\lceil \log_k |D| \rceil$ rounds according to Claim 1, so we require a total of $k \lceil \log_k |D| \rceil$ file injections. □

Claim 3. *For any dictionary D and for any word $w \in D$, let q_w be a private query for w and k be the number of partitions. Then, the total number of words injected by* RecoverQuery(q_w, k) *is* $O(k|D|)$.

Proof. A dictionary D_j is searched during round j of the attack. Each word in partition i of the dictionary appears i times during round j. Each partition has size $\frac{|D_j|}{k}$. Therefore, the total file size injected during this round is $\frac{|D_j|}{k}(1 + 2 + \cdots + k) = O(k|D_j|)$.

Each round reduces the size of the dictionary searched by a factor of k, so $|D_{j+1}| = \frac{|D_j|}{k}$. According to Claim 1, there are $\lceil \log_k |D| \rceil$ many rounds. Then, if the initial dictionary has size $|D|$, the total

Notation	Definition
$\mathcal{D}$	The database, a secondary index mapping words to the files they are associated with.
q_w	A query for the word w, where w is hidden.
D	The dictionary, a set of words probed by the attacker.
k	The number of partitions to search during each round. A higher k means more files injected per round, but fewer rounds total.
n_j	$\lvert q_w(\mathcal{D})\rvert$, or the number of file results for the query on round j. For $j = 0$, this is the user's initial query dictionary.
m	A parameter for the single-round attack. A higher m means more files injected, but higher expected accuracy.
s	A parameter for the noisy data attack. A higher s means more files injected, but a greater possible amount of noise tolerated.

Fig. 3. A table of notation for the attacks described.

file size injected across all rounds is:

$$
\begin{aligned}
&k|D| + k\left(\frac{|D|}{k}\right) + k\left(\frac{|D|}{k^2}\right) + \cdots + k\left(\frac{|D|}{k^{\lceil \log_k |D| \rceil}}\right) \\
&< k|D|\left(1 + \frac{1}{k} + \frac{1}{k^2} + \ldots\right) = O(k|D|)
\end{aligned}
$$

□

By leveraging the three assumptions presented in §3, our attack can recover a user's query on a generic secondary index with perfect accuracy. The number of results returned is indeed the only information we assume. Moreover, we do not require any knowledge of the distribution of queries, file contents and other metadata.

5 EVALUATION

In this section, we evaluate the overheads and accuracy for our attack by simulating an encrypted email database in Section 5.1. We also present a case study on Gmail to evaluate the feasibility of the attacker's capabilities in a real-world application in Section 5.2. We find that a Gmail attacker can perform the necessary injection, replay and file count measurement. In addition, we demonstrate successful attacks on Gmail for a variety of dictionary sizes that complete within a matter of minutes.

5.1 Simulation Using Encrypted Email Databases

5.1.1 Setup. In the following experiments, we use the entire corpus of emails from the Enron email dataset [16] as the queried documents, consisting of ~500K emails belonging to 151 users and ~2.5GB in size. We extracted keywords from this dataset by first stemming the words [48], and then removing 675 stopwords. We next filtered out any words that contained non-alphabetic characters, or were ≥ 20 or ≤ 3 characters long. This gave us a total of ~259K keywords. In our experiments, we only used the top ~123K keywords (*i.e.*, those that appeared in > 3 documents) in order to remove noise from the dataset.

Since an attacker's dictionary may contain words that do not exist in the queried documents, we supplemented the Enron keywords with a corpus of English words [15]. Preprocessing the English words in a similar manner yielded a total of ~257K keywords. The union of both datasets resulted in a universe of ~342K keywords.

5.1.2 Evaluation. Assuming that the queried word is in the initial dictionary chosen by the attacker, our attack achieves perfect query recovery, with strict bounds on the overheads necessary in number of query replays and data injected. Our simulation of the attack in Figures 4 and 5 confirms the theoretical guarantees in Claim 1 and Claim 2.

In this experiment, we build the attacker's dictionary D by randomly sampling keywords from the keyword universe. We pick the keyword queried by the user at random from D in order to stress test the effort required by the attacker—a keyword not in the D would be trivially detected at the end of a single round without requiring further replays. We then report the number of rounds required to guess the keyword with 100% accuracy for different choices of k in Figure 4, and the total number of files injected across rounds in Figure 5. Recall from Section 4.2 that any instance of the attack converges after exactly $\lceil \log_k |D| \rceil$ replays and $k \lceil \log_k |D| \rceil$ files injected, where k is an integer chosen by the attacker. Thus, with a dictionary of fixed size $|D|$, the parameter k represents a tradeoff between the number of query replays required vs. the number of file injections required. The attacker's choice of k then depends on the attacker's ability to replay the query and the rate at which files can be injected for the target application.

We explore this tradeoff with fixed-size dictionaries in Figure 6, which demonstrates how the average number of bytes injected per round increases with k (while the number of rounds decreases). In the worst case where the dictionary comprises the entire keyword universe and $k = 24$, the bytes injected per round is still less than 10MB, demonstrating the feasibility of the attack. We also show the maximum number of bytes injected across any round in Figure 6, equivalent to the number of bytes injected during the first round. The number of bytes that the attacker can inject during a single round must be at least as large as this number. We find that even in the worst case, this is approximately 50MB.

Takeaway. The attack can be mounted easily even when queries are replayed at most once, *i.e.*, the attacker can recover the keyword in merely two rounds without having to inject more than several tens of MBs of data. As an example, Gmail limits the size of emails to a comfortable

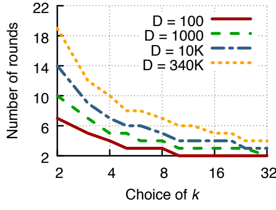


Fig. 4. Number of rounds required to identify a keyword with varying choices of k, across different dictionary sizes.

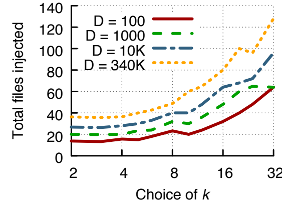


Fig. 5. Total number of files injected with varying choices of k, across different dictionary sizes.

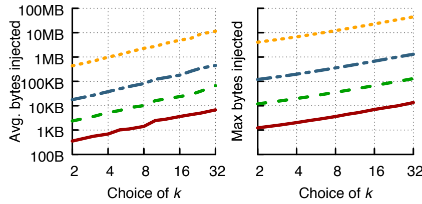


Fig. 6. Number of bytes injected with varying choices of k, across different dictionary sizes: (left) average bytes per round; (right) maximum bytes across rounds.

25MB [23], and the attacker only needs to send 3-4 emails to the victim's inbox in order to identify the query.

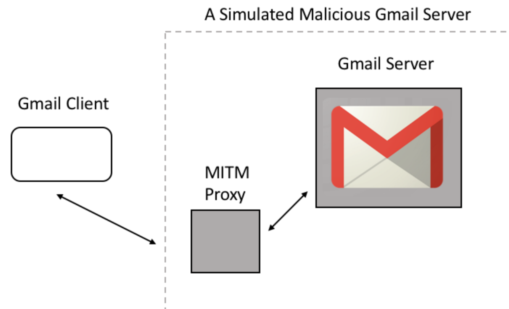


Fig. 7. **(Gmail)** Network setup: The MITM proxy and the real Gmail server together simulates a malicious Gmail server (shaded) who can only learn the result counts of user queries issued by the Gmail client.

5.2 Case Study of Gmail Inbox Search

So far, we have experimentally validated the theoretical performance of our attacks across various parameter choices. In this section, we demonstrate the practical feasibility of our attack in real-world applications by attacking Gmail's inbox search feature.

We attack Gmail by simulating a server-side adversary using a man-in-the-middle proxy (Figure 7), and we assume that the volume of results is leaked to it by the server. We first show that the attacker can indeed meet the two key requirements of file injection and automatic query replay.

Subsequently, we perform an exhaustive experiment across a wide range of dictionary sizes (10 to 100K) to determine the *minimum amount of time* required to mount a successful attack on Gmail. The parameters of our attack are governed by the following constraints: (i) the periodicity of replays in the Gmail client; (ii) the time it takes to inject files into a user's inbox; and (iii) the pagination limit in Gmail (which upper bounds the total number of injections). Despite these constraints, we show that our attack completes within a matter of minutes for the Gmail application. We find that for a small dictionary of size 10, a successful attack can be mounted within 1 minute from start to end; for a large dictionary with 100K words, an attack completes successfully in around 7 minutes (see Figure 9). We now describe our methodology in more detail.

5.2.1 Setup. Since we don't have control over Gmail servers, we simulate a server-side adversary using a man-in-the-middle (MITM) HTTPS proxy [41]. Specifically, we launch the Gmail web client on a browser within a guest virtual machine, and launch the MITM proxy on the host. We reroute all host network traffic through the MITM proxy. Subsequently, we install the proxy's certificate at the client browser in order to simulate a server-side adversary. At this point, all TLS network traffic to and from the client browser passes through the MITM proxy, which it can then examine and manipulate. Though the proxy can now read all plaintext data on the HTTP layer, we make sure that during attack it can only process header information unrelated to the private keywords, *e.g.*,, HTTP method, Origin and Referer fields.

Note that our implementation does not modify any code running on the real Gmail client or the Gmail server. The MITM proxy is the entity who can see the number of matched emails returned from the server, construct and inject special emails, and drop server responses to trigger client replays. Now the proxy and the actual Gmail server together comprises a untrusted Gmail server (the highlighted grey area in Figure 7) who *only* learns about the volume of keyword search results, similar to our model described in §3.

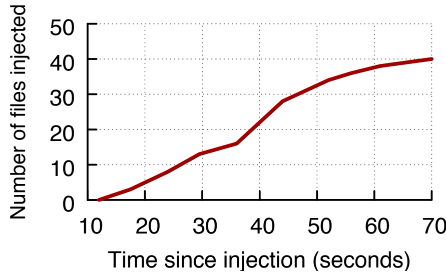


Fig. 8. **(Gmail)** CDF measuring the time it takes for files to be injected into the index for Gmail.

5.2.2 Key Characteristics. We then study Gmail's inbox search features to demonstrate the practical feasibility of our attack.

Query replay. Once a user issues a query, we use the MITM proxy to trigger automatic query replay by simply dropping the HTTP responses returned by the server. After a period of time, the Gmail web client retries the query automatically, without user intervention. Specifically, we find that the client replays the query every 1–3 minutes in the absence of a response. To the user, it simply appears as if the client has a bad network connection.

File injection. File injection in Gmail is simple; the attacker requires a separate Gmail account to send emails to the victim. The attacker must send k emails in each round and also be sure that they are all indexed by the next replay (*i.e.*, at least 60s). We determined the rate at which emails could be injected (Figure 8) to show that it is feasible to index a sufficient number of emails. We found that after injecting 40 emails of size 10KB each, 36 were visible in the user's mailbox 60 seconds later, shown in Figure 8. Thus, within a time window of 60s, the attacker can pick any value less than 36 as a safe option for k.

Volume leakage. In this experiment we assume that the proxy directly obtains the exact result set size from the server, since we simulate a server-side adversary. However, we find that Gmail has a maximum pagination limit of 100, *i.e.*, the server returns at most 100 results in response to a query. The pagination limit constrains the parameter regime of our attack, in that it upper bounds the total number of files that can be injected by the attacker over the duration of the attack.

5.2.3 End-to-end Attack. The aim of the experiment is to minimize the time it takes to launch a successful attack. However, the constraints discussed above—the periodicity of replays, the time it takes to inject files, and the pagination limit—restrict the parameter regime within which an attacker can operate. Therefore, we start by computing the optimal parameters required for mounting a successful attack within the space of possible parameters. Next, we attack Gmail using the computed parameters and report the end-to-end duration of our attack.

Since the Gmail application has a fixed periodicity of replays, the attack duration is directly governed by the number of replays required for the attack. Hence, given a dictionary size $|D|$, our aim is to minimize $\log_k |D|$, where k refers to the number of files that need to be injected per round. However, given the pagination limit of $\ell = 100$, we require that the total number of injected files $k \times \log_k |D|$ be less than ℓ. At the same time, k should be less than 36 given the time it takes to inject files.

We therefore solve the following optimization problem:

$$\begin{aligned} \text{minimize} \quad & \log_k |D| \\ \text{subject to} \quad & k \times \log_k |D| < 100 \\ \text{and} \quad & k < 36 \end{aligned}$$

$\lvert D\rvert$	k	**No. of replays (theoretical)**	**No. of replays (actual)**	**Total injected emails**	**Attack duration**
10	10	1	1	10	1 min
100	10	2	2	20	2 min 5s
1K	32	2	2	63-64	2 min 5s
10K	22	3	3	64	5 min 6s
100K	18	4	5	71	7 min 10s

Fig. 9. **(Gmail)** Attack parameters and duration for Gmail across various dictionary sizes.

Figure 9 summarizes our findings for varying sizes of the attacker's dictionary, 10 to 100K. Note that the total number of injected emails is sometimes marginally less than $k \times \log_k |D|$. This is because $\log_k |D|$ is not always an integer, and therefore files of interest across subsequent rounds may sometimes contain less than k keywords. Additionally, for $|D| = 100K$, our attack requires an extra round of replay because the size of the injected files in the first round were large, increasing the time it took for files to get sent by the client and indexed by the server.

Overall, our experiment demonstrates the feasibility of volume-based attacks on Gmail, which can be successfully completed within a matter of minutes depending on the size of the attacker's dictionary.

In addition, the attack is difficult to detect because during the course of the attack, the user only sees a suspended connection. The user only makes a single query, and the Gmail client automatically replays the query in the background. During this time, emails injected by the attacker are also not delivered to the user's web client, and only modify the server-side index. The user may later see the injected emails, but only after the attack successfully completes.

5.2.4 Other Applications. Besides Gmail, we also use the same setup in Section 5.2.1 to evaluate our attacks on 4 other popular applications: Facebook, Dropbox, Google Doc, and iCloud Mail. Take Facebook as an example, we consider a large group channel, *e.g.*,, free & for sale or course advices, where both the attacker and the user have joined. When the user makes a search within the group, i.e., buying some necessity, the attacker can inject specially-crafted posts to the group to infer what keywords the user searches for. This is problematic because the reconstructed keywords can help the attacker to learn sensitive information about users such as what items they need at the moment or what kinds of academic advices they are seeking for. After conducting a case study on Facebook (similar to the methodology used in Section 5.2.2), we find that Facebook replays group search queries every 1- 4 minutes if not hearing back from the server. Subsequently, we launch our attack with dictionary size D of 10 and partition number k of 10 to demonstrate its practicality. The attack completes in 4 minutes.

6 MITIGATIONS

In this section, we discuss mitigations of our attack. Overall, injection is often fundamental to application functionality, padding is too expensive, and replay could be a legitimate user or application action, as discussed in Section 3.2. Nevertheless, based on our evaluation in §5, we believe that the mitigations proposed below could reduce the extent of our attack by limiting the attacker's abilities (Section 3.2) or making it too expensive to mount.

Note while these methods can mitigate the attack, the vulnerability from result count leakage is difficult to eradicate from a system completely because it relies on very little from the system model and on features inherent to the application model. Rather than trying to reduce system leakage, we believe that the most effective mitigation is actually on the *application* side, although these techniques too may be burdensome because they interfere with application-specific functionality (*e.g.*, disallowing users from sending email).

6.1 Disable File Injection

File injection is arguably the most difficult to defend against, since it is often a part of the target application's functionality. For example, an email inbox search feature is not much use if it could only search from emails that were sent by the user, not to the user. Thus, we believe that the main defense here is rate-limiting and detection. In the email application (Section 3.1), this would require the server to actively filter out suspicious emails. As we found in Section 5.2, applications such as Gmail already rate-limit emails; however, this was not enough to defeat the attack.

6.2 Prevent File Measurement

Reducing the attacker's ability to measure the number of files contained in a response is far more effective. However, padding query responses, while effective in hiding the response size, leads to unacceptable bandwidth overheads for the application. We believe that to varying degrees, this is a property of all padding schemes.

A more effective way to hinder the attacker is to inject some noise in the query responses. This requires little overhead in server-client bandwidth compared to the attacker's overhead: an additive factor of k per query compared to a multiplicative factor of k per attack. This countermeasure is also simple to implement; simply add a random number of dummy files to every response and have the client filter them out.

Another method is to limit the number of results that can be fetched at a time. The user must explicitly request further results if needed. This lowers the feasible dictionary size for the attack, at the cost of user convenience.

6.3 Block Query Replay

The most effective way to prevent our attack is to block query replays. Query replays are an important feature of applications such as Gmail that produce the illusion of a seamless connection during limited network connectivity (Section 5.2). A possible countermeasure is to include a unique query ID for each request, so that the server can detect and filter out duplicate requests.

The main disadvantage of such an approach is that the server would then have to record and replay past responses in order to both prevent the attack and keep the application available. Long-running user sessions would have to be garbage-collected, potentially sacrificing correctness. More crucially, web servers are often replicated for performance and fault tolerance. Ensuring consistency for duplicate queries in such settings is well-known to be expensive, if even possible [22].

7 CONCLUSION

In this work, we proposed a novel generic attack on encrypted databases that only leverages result size leakage. We demonstrated that our attack can reconstruct queries with 100% confidence for a range of realistic settings, jeopardizing the security guarantees of these systems. We showed the effectiveness of our attack via both theoretical bounds and an empirical evaluation, including a demonstration on the Gmail web application.

REFERENCES

[1] Signal. https://signal.org.

[2] M. A. Abdelraheem, T. Andersson, and C. Gehrmann. Inference and Record-Injection Attacks on Searchable Encrypted Relational Databases. Cryptology ePrint Archive, Report 2017/024, 2017. http://eprint.iacr.org/2017/024.

[3] A. Arasu, K. Eguro, R. Kaushik, D. Kossmann, R. Ramamurthy, and R. Venkatesan. A secure coprocessor for database applications. In *FPL*, 2013.

[4] D. Asonov. Querying Databases Privately. In *ISBN 3-540-22441-6 Springer-Verlag Berlin Heidelberg*, 2003.

[5] V. Bindschaedler, M. Naveed, X. Pan, X. Wang, and Y. Huang. Practicing Oblivious Access on Cloud Storage: The Gap, the Fallacy, and the New Way Forward. In *Proceedings of the 22nd ACM Conference on Computer and Communications Security (CCS)*, Denver, CO, 2015.

[6] A. Boldyreva, N. Chenette, Y. Lee, and A. O'Neill. Order-Preserving Symmetric Encryption. In *Proceedings of the 28th Annual International Conference on the Theory and Applications of Cryptographic Techniques (Eurocrypt)*, Cologne, Germany, 2009.

[7] A. Boldyreva, N. Chenette, and A. O'Neill. Order-Preserving Encryption Revisited: Improved Security Analysis and Alternative Solutions. In *Proceedings of the 31st International Cryptology Conference (CRYPTO)*, Santa Barbara, CA, 2011.

[8] R. Bost. $\Sigma o \varphi o \varsigma$: Forward Secure Searchable Encryption. In *Proceedings of the 23rd ACM Conference on Computer and Communications Security (CCS)*, Vienna, Austria, 2016.

[9] D. Cash, P. Grubbs, J. Perry, and T. Ristenpart. Leakage-Abuse Attacks Against Searchable Encryption. In *Proceedings of the 22nd ACM Conference on Computer and Communications Security (CCS)*, Denver, CO, 2015.

[10] D. Cash, J. Jaeger, S. Jarecki, C. Jutla, H. Krawczyk, M. Rosu, and M. Steiner. Dynamic Searchable Encryption in Very-Large Databases: Data Structures and Implementation. In *Proceedings of the 21st Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, 2014.

[11] D. Cash, S. Jarecki, C. Jutla, H. Krawczyk, M. Rosu, and M. Steiner. Highly-Scalable Searchable Symmetric Encryption with Support for Boolean Queries. In *Proceedings of the 33rd International Cryptology Conference (CRYPTO)*, Santa Barbara, CA, 2013.

[12] CipherCloud: Cloud Data Protection Solution, 2017. http://www.ciphercloud.com.

[13] R. Curtmola, J. Garay, S. Kamara, and R. Ostrovsky. Searchable symmetric encryption: improved definitions and efficient constructions. In *Proceedings of the 13th ACM Conference on Computer and Communications Security (CCS)*, Alexandria, VA, 2006.

[14] J. L. Dautrich, Jr. and C. V. Ravishankar. Compromising Privacy in Precise Query Protocols. In *International Conference on Extending Database Technology*, 2013.

[15] English keywords dataset, 2017. https://github.com/dwyl/english-words.

[16] Enron email dataset, 2017. https://www.cs.cmu.edu/~./enron/.

[17] S. Faber, S. Jarecki, H. Krawczyk, Q. Nguyen, M. Rosu, and M. Steiner. Rich Queries on Encrypted Data: Beyond Exact Matches. In *Proceedings of the 20th European Symposium on Research in Computer Security (ESORICS)*, Vienna, Austria, 2015.

[18] R. Fielding and J. Reschke. Hypertext Transfer Protocol (HTTP/1.1): Message Syntax and Routing. RFC 7230, 2014.

[19] B. Fuller, M. Varia, A. Yerukhimovich, E. Shen, A. Hamlin, V. Gadepally, R. Shay, J. D. Mitchell, and R. K. Cunningham. SoK: Cryptographically Protected Database Search. In *Proceedings of the 38th IEEE Symposium on Security and Privacy (IEEE S&P)*, 2017.

[20] W. Gasarch. A survey on private information retrieval. In *The Computational Complexity Column*, 2007.

[21] M. Giaruad, A. Anzala-Yamajako, O. Bernard, and P. Lafourcase. Practical Passive Leakage-Abuse Attacks Against Symmetric Searchable Encryption. Cryptology ePrint Archive, Report 2017/046, 2017. http://eprint.iacr.org/2017/046.

[22] S. Gilbert and N. Lynch. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *ACM SIGACT News*, 33(2):51–59, 2002.

[23] Gmail size limits, 2017. https://support.google.com/mail/answer/6584.

[24] O. Goldreich and R. Ostrovsky. Software Protection and Simulation on Oblivious RAMs. *J. ACM*, pages 431–473, 1996.

[25] P. Grubbs, M.-S. Lacharité, B. Minaud, and K. G. Paterson. Pump up the volume: Practical database reconstruction from volume leakage on range queries. In *Proceedings of the 25th ACM Conference on Computer and Communications Security (CCS)*, 2018.

[26] P. Grubbs, R. McPherson, M. Naveed, T. Ristenpart, and V. Shmatikov. Breaking Web Applications Built On Top of Encrypted Data. In *Proceedings of the 23rd ACM Conference on Computer and Communications Security (CCS)*, Vienna, Austria, 2016.

[27] W. He, D. Akhawe, S. Jain, E. Shi, and D. X. Song. ShadowCrypt: Encrypted Web Applications for Everyone. In *Proceedings of the 21st ACM Conference on Computer and Communications Security (CCS)*, Scottsdale, AZ, 2014.

[28] iQrypt: Encrypt and query your database, 2017. http://iqrypt.com/.

[29] M. S. Islam, M. Kuzu, and M. Kantarcioglu. Access Pattern Disclosure on Searchable Encryption: Ramification, Attack and Mitigation. In *Proceedings of the 19th Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, 2012.

[30] M. S. Islam, M. Kuzu, and M. Kantarcioglu. Inference Attack Against Encrypted Range Queries on Outsourced Databases. In *Proceedings of the 4th ACM Conference on Data and Application Security and Privacy*, San Antonio, TX, 2014.

[31] S. Kamara, C. Papamanthou, and T. Roeder. Dynamic Searchable Symmetric Encryption. In *Proceedings of the 19th ACM Conference on Computer and Communications Security (CCS)*, Raleigh, NC, 2012.

[32] G. Kellaris, G. Kollios, K. Nissim, and A. O'Neill. Generic Attacks on Secure Outsourced Databases. In *Proceedings of the 23rd ACM Conference on Computer and Communications Security (CCS)*, Vienna, Austria, 2016.

[33] F. Kerschbaum and A. Schröpfer. Optimal Average-Complexity Ideal-Security Order-Preserving Encryption. In *Proceedings of the 21st ACM Conference on Computer and Communications Security (CCS)*, Scottsdale, AZ, 2014.

[34] K. Kurosawa. Garbled searchable symmetric encryption. In *Proceedings of the 18th International Conference on Financial Cryptography*, 2014.

[35] M.-S. Lacharité, B. Minaud, and K. G. Paterson. Improved reconstruction attacks on encrypted data using range query leakage. In *Proceedings of the 39th IEEE Symposium on Security and Privacy (IEEE S&P)*, 2018.

[36] B. Lau, S. P. Chung, C. Song, Y. Jang, W. Lee, and A. Boldyreva. Mimesis Aegis: A Mimicry Privacy Shield - A System's Approach to Data Privacy on Public Cloud. In *Proceedings of the 23rd USENIX Security Symposium*, San Diego, CA, 2014.

[37] K. Lewi and D. J. Wu. Order-Revealing Encryption: New Constructions, Applications, and Lower Bounds. In *Proceedings of the 23rd ACM Conference on Computer and Communications Security (CCS)*, Vienna, Austria, 2016.

[38] C. Liu, L. Zhu, M. Wang, and Y.-A. Tan. Search pattern leakage in searchable encryption: Attacks and new construction. *Inf. Sci.*, 265:176–188, 2014.

[39] M. Marlinspike. Technology preview: Private contact discovery for signal. https://signal.org/blog/private-contact-discovery/, 2017.

[40] Microsoft SQL Server: Always Encrypted Database Engine, 2017. https://msdn.microsoft.com/en-us/library/mt163865.aspx.

[41] MITM Proxy, 2018. http://mitmproxy.org/.

[42] M. Naveed. The Fallacy of Composition of Oblivious RAM and Searchable Encryption. Cryptology ePrint Archive, Report 2015/668, 2015. http://eprint.iacr.org/2015/668.

[43] M. Naveed, M. Prabhakaran, and C. A. Gunter. Dynamic Searchable Encryption via Blind Storage. In *Proceedings of the 35th IEEE Symposium on Security and Privacy (IEEE S&P)*, 2014.

[44] W. Ogata, K. Koiwa, A. Kanaoka, and S. Matsuo. Toward Practical Searchable Symmetric Encryption. In *Proceedings of the 8th International Workshop on Security*, 2013.

[45] F. Olumofin and I. Goldberg. Privacy-preserving queries over relational databases. In *Proceedings of the 10th Privacy Enhancing Technologies Symposium (PETS)*, Berlin, Germany, 2010.

[46] R. A. Popa, F. H. Li, and N. Zeldovich. An Ideal-Security Protocol for Order-Preserving Encoding. In *Proceedings of the 34th IEEE Symposium on Security and Privacy (IEEE S&P)*, San Francisco, CA, 2013.

[47] R. A. Popa, C. M. S. Redfield, N. Zeldovich, and H. Balakrishnan. CryptDB: Protecting Confidentiality with Encrypted Query Processing. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP)*, Cascais, Portugal, 2011.

[48] M. F. Porter. An Algorithm for Suffix Stripping. *Readings in Information Retrieval*, pages 313–316, 1997.

[49] D. Pouliot and C. V. Wright. The Shadow Nemesis: Inference Attacks on Efficiently Deployable, Efficiently Searchable Encryption. In *Proceedings of the 23rd ACM Conference on Computer and Communications Security (CCS)*, Vienna, Austria, 2016.

[50] Cloud Threat Intelligence, Skyhigh Cloud Security labs, Skyhigh Networks, 2017. https://www.skyhighnetworks.com/.

[51] D. S. Roche, A. J. Aviv, and S. G. Choi. A Practical Oblivious Map Data Structure with Secure Deletion and History Independence. In *Proceedings of the 37th IEEE Symposium on Security and Privacy (IEEE S&P)*, 2016.

[52] C. Sahin, V. Zakhary, A. El Abbadi, H. Lin, and S. Tessaro. TaoStore: Overcoming Asynchronicity in Oblivious Data Storage. In *Proceedings of the 37th IEEE Symposium on Security and Privacy (IEEE S&P)*, 2016.

[53] N. P. Smart (Editor). Future Directions in Computing on Encrypted Data. In *ECRYPT report*, 2015.

[54] D. X. Song, D. Wagner, and A. Perrig. Practical Techniques for Searches on Encrypted Data. In *Proceedings of the 21st IEEE Symposium on Security and Privacy (IEEE S&P)*, Oakland, CA, 2000.

[55] E. Stefanov, C. Papamanthou, and E. Shi. Practical Dynamic Searchable Encryption with Small Leakage. In *Proceedings of the 21st Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, 2014.

[56] E. Stefanov and E. Shi. ObliviStore: High Performance Oblivious Cloud Storage. In *Proceedings of the 36th IEEE Symposium on Security and Privacy (IEEE S&P)*, 2015.

[57] E. Stefanov, M. van Dijk, E. Shi, C. W. Fletcher, L. Ren, X. Yu, and S. Devadas. Path ORAM: an extremely simple oblivious RAM protocol. In *Proceedings of the 20th ACM Conference on Computer and Communications Security (CCS)*, Berlin, German, 2013.

[58] S. Tu, M. F. Kaashoek, S. Madden, and N. Zeldovich. Processing Analytical Queries over Encrypted Data. In *Proceedings of the 39th International Conference on Very Large Data Bases (VLDB)*, Riva del Garda, Italy, 2013.

[59] S. Wang, D. Agrawal, and A. E. Abbadi. Generalizing PIR for Practical Private Retrieval of Public Data. In *Lecture Notes in Computer Science, volume 6166*, 2010.

[60] Y. Zhang, J. Katz, and C. Papamanthou. All Your Queries Are Belong to Us: The Power of File-Injection Attacks on Searchable Encryption. In *Proceedings of the 25th USENIX Security Symposium*, Austin, TX, 2016.