All Your Queries Are Belong to Us: The Power of File-Injection Attacks on Searchable Encryption

YUPENG ZHANG[*]   JONATHAN KATZ[†]   CHARALAMPOS PAPAMANTHOU[*]

Abstract

The goal of *searchable encryption* (SE) is to enable a client to execute searches over encrypted files stored on an untrusted server while ensuring some measure of privacy for both the encrypted files and the search queries. Research has focused on developing efficient SE schemes at the expense of allowing some small, well-characterized "(information) leakage" to the server about the files and/or the queries. The practical impact of this leakage, however, remains unclear.

We thoroughly study *file-injection attacks*—in which the server sends files to the client that the client then encrypts and stores—on the query privacy of single-keyword and conjunctive SE schemes. We show such attacks can reveal the client's queries in their entirety using very few injected files, even for SE schemes having low leakage. We also demonstrate that natural countermeasures for preventing file-injection attacks can be easily circumvented. Our attacks outperform prior work significantly in terms of their effectiveness as well as in terms of their assumptions about the attacker's prior knowledge.

1 Introduction

The goal of *searchable encryption* (SE) is to enable a client to perform keyword searches over encrypted files stored on an untrusted server while still guaranteeing some measure of privacy for both the files themselves as well as the client's queries. In principle, solutions that leak no information to the server can be constructed based on powerful techniques such as secure two-party computation, fully-homomorphic encryption, and/or oblivious RAM. Such systems, however, would be prohibitively expensive and completely impractical [15].

In light of the above, researchers have focused on the development of novel SE schemes that are much more efficient, at the expense of allowing some information to "leak" to the server [19, 9, 8, 11, 6, 16, 12, 20, 13, 5]. The situation is summarized, e.g., by Cash et al. [6]:

> *The premise of [our] work is that in order to provide truly practical SSE solutions one needs to accept a certain level of leakage; therefore, the aim is to achieve an acceptable balance between leakage and performance.*

The question then becomes: what sort of leakage is acceptable? Roughly speaking, and focusing on single-keyword search for simplicity, current state-of-the-art schemes leak mainly two things: the *query pattern* (i.e., when a query is repeated) and the *file-access pattern* (namely, which files are returned in response to each query); these are collectively called *L1 leakage* in [4]. The prevailing argument is that L1 leakage is inconsequential in practice, and so represents a reasonable sacrifice for obtaining an efficient SE scheme.

In truth, the ramifications of different types of leakage are poorly understood; indeed, characterizing the real-world consequences of the leakage of existing SE schemes was highlighted as an important open question in [6]. Recently, several groups have shown that even seemingly minor leakage can be exploited to learn sensitive information, especially if the attacker has significant prior knowledge about the client's files or the keywords they contain. Islam et al. [10] (IKK12), who initiated this line of work, showed that if the server knows (almost) all the contents of the client's files, then it can determine the client's queries from L1 leakage. Cash et al. [4] (CGPR15) gave an improved attack that works for larger keyword universes while assuming (slightly) less knowledge about the files of the client. They also explored the effects of even greater leakage, and showed how query-recovery attacks could serve as a springboard for learning further information about the client's files.

A different attack for query recovery was given by Liu et al. [14]. The attack assumes a known distribution on the keywords being searched by the client, and works only after the client issues a large number of queries.

1.1 Our Contributions

In this paper, we further investigate the consequences of leakage in SE schemes through the lens of *file-injection attacks*. In such attacks, the server sends files of its choice to the client, who then encrypts and uploads them as dictated by the SE scheme. This attack was introduced by Cash et al. [4], who called it a *known-document attack*. As argued by those authors, it would be quite easy to carry out such attacks: for example, if a client is using an SE scheme for searching email (e.g., Pmail [2]), with incoming emails processed automatically, then the server can inject files by simply sending email to the client (from a spoofed email address, if it wishes to avoid suspicion). We stress that the server otherwise behaves entirely in an "honest-but-curious" fashion.

We show that file-injection attacks are *devastating* for query privacy: that is, a server can learn a very high fraction of the keywords being searched by the client, by injecting a relatively small number of files. As compared to prior work [10, 4], our attacks are both more effective in terms of the fraction of queries recovered and far less demanding in terms of the prior information the server is assumed to know. Our attacks differ in that the server must inject files, but as argued above this would be easy to carry out in practice.

We consider both adaptive and non-adaptive attacks, where adaptivity refers (in part) to whether the server injects files before or after the client's query is made. Our adaptive attacks are more effective, but assume the SE scheme does not satisfy *forward privacy* [7, 20]. (Forward privacy means that the server cannot tell if a newly inserted file matches previous search queries. With the exception of [7, 20], however, all efficient SE schemes supporting updates do not have forward privacy.) Our work thus highlights the importance of forward privacy in any real-world deployment.

1.2 Organization of the paper

We begin by showing a simple, binary-search attack that allows the server to learn 100% of the client's queries with *no* prior knowledge about the client's files. We then propose an easy countermeasure: limiting the number of keywords that are indexed per file. (We show that this idea is viable insofar as it has limited effect on the utility of searchable encryption.) However, our attacks can be suitably modified to defeat this countermeasure, either using a larger number of injected files (but still no prior knowledge about the client's files) or based on limited knowledge—as low as 10%—of the client's files. Our attacks still outperform prior work [10, 4], having a significantly higher recovery rate and requiring a lower fraction of the client's files to be known.

We additionally investigate the effectiveness of padding files with random keywords (suggested in [10, 4]) as another countermeasure against our attacks. We show that the performance of our attacks degrades only slightly when such padding is used, in contrast to prior attacks that fail completely.

Finally, we initiate a study of the implications of leakage on *conjunctive* queries, and show how to extend our attacks to this setting. Our attacks work against SE schemes having "ideal" leakage, but are even more effective against the scheme of Cash et al. [6] (the most efficient SE scheme allowing conjunctive queries), which suffers from larger leakage.

2 Background

For the purposes of this paper, only minimal background about searchable encryption (SE) is needed. At a high level, an SE scheme allows a client to store encrypted versions of its files on a server, such that at a later point in time the client can retrieve all files containing a certain keyword (or collection of keywords). We assume a set of keywords $K = \{k_0, k_1, \ldots\}$ known to an attacker, and for simplicity view a file as an unordered set of keywords. (Although the order and multiplicity of the keywords matter, and a file may contain non-keywords as well, these details are irrelevant for our purposes.)

We assume an SE scheme in which searching for some keyword k is done via the following process (all efficient SE schemes work in this way): first, the client deterministically computes a *token t* corresponding to k and sends t to the server; using t, the server then computes and sends back the *file identifiers* of all files containing keyword k. (These file identifiers need not be "actual" filenames; they can instead simply be pointers to the appropriate encrypted files residing at the server.) The client then downloads the appropriate files.

Because the token is generated deterministically from the keyword, the server can tell when queries repeat and thus learn the *query pattern*; the returned file identifiers reveal the *file-access pattern*. Our attacks rely only on knowledge of the file-access pattern, though we additionally assume that the server can identify when a specific file identifier corresponds to some particular file injected by the server. (The same assumption is made by Cash et al. [4].) This is reasonable to assume, even if file identifiers are chosen randomly by the client, for several reasons: (1) the server can identify the file returned based on its length (even if padding is used to mitigate this, it is impractical to pad every file to the maximum file length); (2) in SE schemes supporting updates, the server can inject a file F and then identify F with the next (encrypted) file uploaded by the client; (3) if the server can influence the queries of the client, or even if it knows some of the client's queries, then the server can use that information to identify specific injected files with particular file identifiers. We postpone further discussion to Section 8.
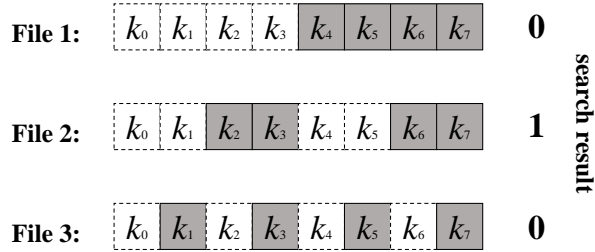
Figure 1: An example of the binary-search attack with $|K| = 8$. Each file injected by the attacker contains 4 keywords, which are shaded in the figure. If file 2 is returned in response to some token, but files 1 and 3 are not, the keyword corresponding to that token is k_2.

In this paper, we focus only on query-recovery attacks, where the server observes various tokens sent by the client and the file identifiers returned, and the server's goal is to determine the keywords corresponding to those tokens. As noted by Cash et al. [4], though, such query-recovery attacks can also be used to learn partial information about the client's files.

3 Binary-Search Attack

In this section, we present a basic query-recovery attack that we call the *binary-search attack*. This attack does not require the server to have any knowledge about the client's files, and recovers all the keywords being searched by the client with 100% accuracy.

3.1 Basic Algorithm

The basic observation is that if the server injects a file F containing exactly half the keywords from the keyword universe K, then by observing whether the token t sent by the client matches that file (i.e., whether F is returned in response to that token), the server learns one bit of information about the keyword corresponding to t. Using a standard non-adaptive version of binary search, the server can thus use $\lceil \log |K| \rceil$ injected files to determine the keyword exactly. The idea is illustrated in Figure 1 for $|K| = 8$.

The attack is described more formally in the pseudocode of Figure 2. We assume for simplicity that $|K|$ is a power of 2, and identify K with the set $\{0, \ldots, |K| - 1\}$ written in binary. The attack begins by having the server generate a set $\mathbf{F}$ of $\log |K|$ files to be injected, where the ith file contains exactly those keywords whose ith most-significant bit is equal to 1. At some point,[1] the server learns, for each injected file, whether it is returned in response to some token t. We let $R = r_1 r_2 \cdots$ denote the search results on the injected files, where $r_i = 1$ if and only if the ith file is returned in response to the token. For this attack, the server can deduce that the keyword corresponding to t is precisely R.

We highlight again that for this attack, the files are generated non-adaptively and independent of the token t. We note further that the *same* injected files can be used to recover the keywords corresponding to *any number of tokens*, i.e., once these files are injected, the server can recover

---

[1]This can occur if the files are injected before the token t is sent, or if the files are injected after t is sent and the SE scheme does not satisfy forward privacy.

Algorithm $\mathbf{F} \leftarrow \mathsf{Inject\_Files}(K)$

  1: **for** $i = 1, \ldots, \log|K|$ **do**

  2:     Generate a file F_i that contains exactly the keywords in K whose ith bit is 1.

  3: Output $\mathbf{F} = \{F_1, \ldots, F_{\log|K|}\}$.

Algorithm $k \leftarrow \mathsf{Recover}(R, K)$

  1: Return R as the keyword from universe K associated with the token.

Figure 2: The binary-search attack. R denotes the search results for the token to be recovered on the injected files.

the keywords corresponding to any future tokens sent by the client. The number of injected files needed for this attack is quite reasonable; with a 10,000-keyword universe, a server who sends only one email per day to the client can inject the necessary files in just 2 weeks.

Small keyword universe. For completeness and future reference, we note that the binary-search attack can be optimized if the hidden keyword is known to lie in some smaller universe of keywords, or if the server only cares about keywords lying in some subset of the entire keyword universe (and gives up on learning the keyword if it lies outside this subset). Specifically, the server can carry out the binary-search attack from Figure 2 based on any subset $K' \subset K$ of the keyword universe using only $\log|K'|$ injected files.

3.2 Threshold Countermeasure

A prominent feature of the binary-search attack is that the files that need to be injected for the attack each contain a large number of keywords, i.e., $|K|/2$ keywords per file. We observe, then, that one possible countermeasure to our attack is to modify the SE scheme so as to limit the number of keywords per indexed file to some threshold $T \ll |K|/2$. This could be done either by simply not indexing files containing more than T keywords (possibly caching such files at the client), or by choosing at most T keywords to index from any file containing more than T keywords.

    The threshold T can be set to some reasonably small value while not significantly impacting the utility of the SE scheme. For example, in the Enron email dataset [1] with roughly 5,000 keywords (see Section 5 for further details), the average number of keywords per email is 90; only 3% of the emails contain more than 200 keywords. Using the threshold countermeasure with $T = 200$ would thus affect only 3% of the honest client's files, but would require the server to inject many more files in order to carry out a naive variant of the binary-search attack. Specifically, the server could replace each file F_i (that contains $|K|/2$ keywords) in the basic attack with a sequence of $|K|/2T$ files $F_{i,1}, \ldots, F_{i,|K|/2T}$ each containing T keywords, such that $\cup_j F_{i,j} = F_i$. If any of these files is returned, this is equivalent to the original file F_i being returned in the basic attack. Note, however, that the server must now inject $|K|/2T \cdot \log|K|$ files. Unfortunately, as we explore in detail in the following section, the threshold countermeasure can be defeated using fewer injected files via more-sophisticated attacks.

    Note also that the threshold countermeasure does not affect the binary-search attack with small keyword universe $K' \subset K$, as long as $|K'| \leq 2T$.

4 Advanced Attacks

In this section, we present more-sophisticated attacks for when the threshold countermeasure introduced in the previous section is used. In Section 4.1 we show an attack that uses fewer injected files than a naive modification of the binary-search attack, still without any knowledge of the client's files. Then, in the following section, we show attacks that reduce the number of injected files even further, but based on the assumption that the server has information about some fraction of the client's files.

4.1 Hierarchical-Search Attack

We noted earlier that the threshold countermeasure does not affect the binary-search attack with small keyword universe $K' \subset K$ if $|K'| \leq 2T$. We can leverage this to learn keywords in the *entire* universe using what we call a *hierarchical search attack*. This attack works by first partitioning the keyword universe into $\lceil |K|/T \rceil$ subsets containing T keywords each. The server injects files containing the keywords in each subset to learn which subset the client's keyword lies in. In addition, it uses the small-universe, binary-search attack on adjacent pairs of these subsets to determine the keyword exactly. The algorithm is presented in Figure 3.
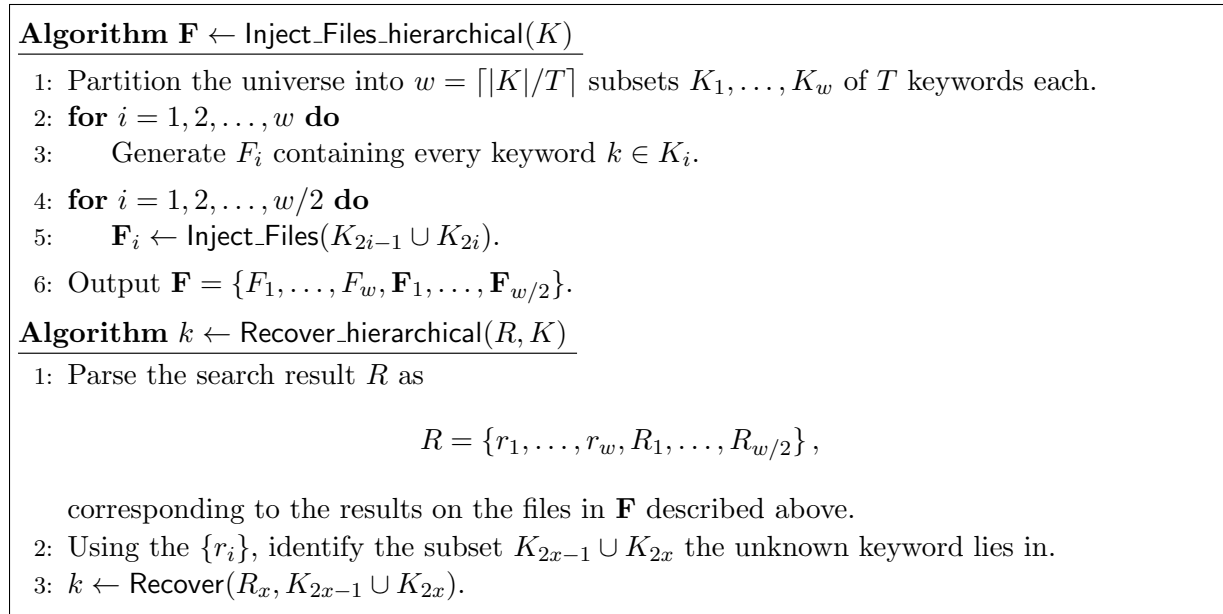

Algorithm $\mathbf{F} \leftarrow$ Inject_Files_hierarchical(K)

1: Partition the universe into $w = \lceil |K|/T \rceil$ subsets $K_1, \ldots, K_w$ of T keywords each.
2: **for** $i = 1, 2, \ldots, w$ **do**
3:     Generate F_i containing every keyword $k \in K_i$.
4: **for** $i = 1, 2, \ldots, w/2$ **do**
5:     $\mathbf{F}_i \leftarrow$ Inject_Files$(K_{2i-1} \cup K_{2i})$.
6: Output $\mathbf{F} = \{F_1, \ldots, F_w, \mathbf{F}_1, \ldots, \mathbf{F}_{w/2}\}$.

Algorithm $k \leftarrow$ Recover_hierarchical(R, K)

1: Parse the search result R as

$$R = \{r_1, \ldots, r_w, R_1, \ldots, R_{w/2}\},$$

    corresponding to the results on the files in $\mathbf{F}$ described above.
2: Using the $\{r_i\}$, identify the subset $K_{2x-1} \cup K_{2x}$ the unknown keyword lies in.
3: $k \leftarrow$ Recover$(R_x, K_{2x-1} \cup K_{2x})$.

Figure 3: The hierarchical-search attack. T is the threshold determining the maximum number of keywords in a file. R denotes the search results on the injected files. Inject_Files and Recover are from Figure 2.

We now calculate the number of injected files required by this attack. In Step 3 of Inject_Files_hierarchical, the server injects $\lceil |K|/T \rceil$ files, and in Step 5 it injects $\lceil |K|/2T \rceil \cdot \lceil \log 2T \rceil$ files. The total number of injected files is therefore at most

$$\lceil |K|/2T \rceil \cdot (\lceil \log 2T \rceil + 2).$$

In fact, for each i the first file in the set $\mathbf{F}_i$ generated by $\mathsf{Inject\_files}(K_{2i-1} \cup K_{2i})$ is the same as F_{2i-1} and the server does not need to inject it again. Also, the server does not need to generate F_w in Step 3 because if the keyword is not in $F_1, \ldots, F_{w-1}$ then the server knows it must be in F_w. So the total number of injected files can be improved to

$$\lceil |K|/2T \rceil \cdot (\lceil \log 2T \rceil + 1) - 1 \, .$$

When the size of the keyword universe is $|K| = 5,000$ and the threshold is $T = 200$, the server needs to inject only 131 files, and the number of injected files grows linearly with the size of the keyword universe. We highlight again that the same injected files can be used to recover the keywords corresponding to any number of tokens; i.e., once these files are injected, the server can recover the keywords of any future searches made by the client.

We remark that an *adaptive* version of the above attack is also possible. Here, the attacker would first inject $\lceil |K|/T \rceil - 1$ files to learn what subset the unknown keyword lies in, and then carry out the small-universe, binary-search attack on a subset of size T. This requires only $\lceil |K|/T \rceil + \log T - 1$ injected files, but has the disadvantage of being adaptive and hence requires the SE scheme to not satisfy forward privacy. This version of the attack also has the disadvantage of targeting one particular search query of the client; additional files may need to be injected to learn the keyword used in some subsequent search query.

4.2 Attacks Using Partial Knowledge

With the goal of further decreasing the number of injected files required to recover a token in presence of the threshold countermeasure, we now explore additional attacks that leverage prior information that the server might have about some of the client's files; we refer to the files known to the server as *leaked files*.[2] A similar assumption is used in prior work showing attacks on SE schemes [10, 4]; previous attacks, however, require the server to know about 90% of the client's files to be effective (see Section 5), whereas our attacks work well even when the server knows a much smaller fraction of the client's files.

Our attacks utilize the frequency of occurrence of the tokens and keywords in the client's files. We define the *frequency* of a token (resp., keyword) as the fraction of the client's files containing this token (resp., keyword). Similarly, we define the *joint frequency* of two tokens (resp., keywords) as the fraction of files containing *both* tokens (resp., keywords). The server learns the exact frequency (resp., joint frequency) of a token (resp., pair of tokens) based on the observed search results. The server obtains an estimate of the frequencies (resp., joint frequencies) of all the keywords based on the client's files that it knows. We let $f(t)$ denote the exact (observed) frequency of token t, and let $f(t_1, t_2)$ be the joint frequency of tokens t_1, t_2. We use $f^*(k)$ to denote the estimated frequency of keyword k, and define $f^*(k_1, k_2)$ analogously. Our attacks use the observation that if the leaked files are representative of all the client's files, then $f(t)$ and $f^*(k)$ are close when t is the token corresponding to keyword k.

4.2.1 Recovering One Keyword

Say the server obtains a token t sent by the client, having observed frequency $f(t)$. The server first constructs a *candidate universe* K' for the keyword corresponding to t consisting of the $2T$

---

[2]We stress that our attacks only rely on the *content* of these leaked files; we do not assume the server can identify the file identifiers corresponding to the leaked files after they have been uploaded to the server.

Algorithm $k \leftarrow$ Inject_Files_Single(t, K)

  1: Let K' be the set of $2T$ keywords with estimated frequencies closest to $f(t)$.
  2: $\mathbf{F} \leftarrow$ Inject_Files(K').

Algorithm $k \leftarrow$ Recover_Single(R, K')

  1: If R contains all 0s, output $\perp$.
  2: Else $k \leftarrow$ Recover(R, K').

Figure 4: Recovering a single keyword using partial file knowledge. T is the threshold determining the maximum number of keywords in a file. R denotes the search results on the injected files. Inject_Files and Recover are from Figure 2.

keywords whose estimated frequencies are closest to $f(t)$. The server then uses the small-universe, binary-search attack to recover the keyword exactly. In this way, the number of injected files is only $\lceil \log 2T \rceil$. The attack is presented in detail in Figure 4.

Differences from attacks in previous sections. The attack just described is *adaptive*, in that it targets a particular token t and injects files whose contents depend on the results of a search using t. This means the attack only applies to SE schemes that do not satisfy forward privacy. It also means that the attack needs to be carried out again in order to learn the keyword corresponding to some other token.

Another difference from our previous attacks is that this attack does not work with certainty. In particular, if the observed and estimated frequencies are far apart, or the number of keywords whose estimated frequencies are close to the observed frequency is larger than $2T$, the server may fail to recover the keyword corresponding to the token. On the other hand, the server can tell whether the attack succeeds or not, so will never associate an incorrect keyword with a token. This also means that if the attack fails, the attacker can re-run the attack with a different candidate universe, or switch to using one of our earlier attacks, in order to learn the correct keyword. (We rely on this feature to design an attack for multiple tokens in the following section.) This is in contrast to earlier attacks [10, 4], where the attacker cannot always tell whether the keyword was recovered correctly.

4.2.2 Recovering Multiple Keywords

To learn the keywords corresponding to m tokens, the server can repeat the attack above for each token, but then the number of injected files will be (in the worst case) $m \cdot \lceil \log 2T \rceil$. A natural way to attempt to reduce the number of injected files is for the server to determine a candidate universe of size $2T$ for each token and then use the union of those candidate universes when injecting the files. In that case, however, the union would almost surely contain more than $2T$ keywords, in which case the number of keywords in the files produce by the binary-search attack will exceed the threshold T.

A second approach would be for the server to make the size of the candidate universe for each token $2T/m$, so the size of their union cannot exceed $2T$ keywords. Here, however, if m is large then the candidate universe for each token is very small and so the probability of the corresponding keyword not lying in its candidate universe increases substantially. Therefore, the recovery rate of

$\mathbf{t} = \{t_1, \ldots, t_m\}$ is the set of m tokens whose keywords we wish to recover.

Algorithm $\mathbf{k} \leftarrow$ Attack_Multiple_Tokens$(\mathbf{t}, K)$

Build ground truth set G.

1: Sort tokens in $\mathbf{t}$ according to their exact frequencies $f(t)$. Let $\mathbf{t}_1$ denote the n tokens with highest observed frequencies.
2: **for** each token t in $\mathbf{t}_1$ **do**
3:     Set its candidate universe K_t as the set of $\frac{2T}{n}$ keywords with estimated frequencies $f^*(k)$ nearest to $f(t)$.
4: Define $K' = \cup_{t \in \mathbf{t}_1} K_t$ and inject files generated by $\mathbf{F}_1 \leftarrow$ Inject_Files(K').
5: **for** each token t in $\mathbf{t}_1$ **do**
6:     Let R_t be the search result of token t on files $\mathbf{F}_1$.
7:     **if** R_t is not all 0s **then**
8:         $k_t \leftarrow$ Recover(R_t, K').
9:         Add (t, k_t) to G.

Recover the remaining tokens, let $\mathbf{t}_2$ be the set of unrecovered tokens.

10: **for** each token $t' \in \mathbf{t}_2$ **do**
11:     Set its candidate universe $K_{t'}$ as the set of $2T$ keywords with estimated frequencies $f^*(k)$ nearest to $f(t')$.
12:     **for** each keyword $k' \in K_{t'}$ **do**
13:         **for** each token/keyword pair $(t, k) \in G$ **do**
14:             If $|f(t, t') - f^*(k, k')| > \delta \cdot f^*(k, k')$, remove k' from candidate universe $K_{t'}$.
15: Set $K'' = \cup_{t' \in \mathbf{t}_2} K_{t'}$.
16: **if** $|K''| \leq 2T$ **then**
17:     $\mathbf{F}_2 \leftarrow$ Inject_Files(K'').
18:     **for** each token $t' \in \mathbf{t}_2$ **do**
19:         Let $R_{t'}$ be the search result of token t' on files $\mathbf{F}_2$.
20:         $k_{t'} \leftarrow$ Recover$(R_{t'}, K'')$
21: **else**
22:     $\mathbf{F}_2 \leftarrow$ Inject_Files_hierarchical(K'').
23:     **for** each token $t' \in \mathbf{t}_2$ **do**
24:         Let $R_{t'}$ be the search result of token t' on files $\mathbf{F}_2$.
25:         $k_{t'} \leftarrow$ Recover_hierarchical$(R_{t'}, K'')$
26: Output $\mathbf{k}$ that includes all recovered keywords.

Figure 5: Recovering multiple keywords using partial file knowledge. T is the threshold determining the maximum number of keywords in a file; δ is a parameter. Inject_Files and Recover are from Figure 2.

this attack would be low.

Instead, we propose a more-complex attack that recovers multiple tokens by taking into account the joint frequencies for tokens and keywords. Our attack has two main steps (see Figure 5):

1. First, we recover the keywords corresponding to a subset of the tokens, namely the $n \ll m$ tokens with the highest observed frequencies. We recover the keywords using the second approach sketched above, which works (with few injected files) because n is small. This gives us as a set of tokens and their associated keywords as "ground truth."

2. Given the ground truth, we recover the keyword associated with some other token t' using the following observation: if k' is the keyword corresponding to t', then the observed joint frequency $f(t, t')$ should be "close" to the estimated joint frequency $f^*(k, k')$ for *all* pairs (t, k) in our ground-truth set, where "closeness" is determined by a parameter δ. By discarding candidate keywords that do not satisfy this property, we are left with a small set K' of candidate keywords for t'. If the candidate universe of keywords for each token is small enough, then even their union will be small. We then use a small-universe, binary-search attack to recover the corresponding keywords exactly.

Note that in the above attack the ability to tell whether a token is recovered correctly when building the ground truth is crucial—otherwise the ground-truth set could contain many incorrect associations.

Parameter selection. Our attack has two parameters: n and δ. A larger value of n means that the ground-truth set can potentially be larger, but if n is too large then there is a risk that the candidate universe K_t (comprising the $2T/n$ keywords with estimated frequencies closest to $f(t)$) will not contain the true keyword corresponding to t. In our experiments, we set n heuristically to a value that achieves good performance.

The value of δ is chosen based on statistical-estimation theory. The estimated joint frequency is an empirical average computed from a collection of leaked files assumed to be sampled uniformly from the set of all files. Thus, we set δ such that if keywords k, k' correspond to tokens t, t', respectively, then the estimated joint frequency $f^*(k, k')$ is within $\pm\delta \cdot f^*(k, k')$ of the true value $f(t, t')$ at least 99% of the time.

Ground-truth set selection. When building the ground-truth set, we recover the keywords associated with those tokens having the highest observed frequencies. We do so because those keywords can be recovered correctly with higher probability, as we explain next.

If the leaked files are chosen uniformly from the set of all files, then using statistical-estimation theory as above the attacker can compute a value δ such that at least 99% of the time it holds that $|f^*(k) - f(t)| \leq \epsilon \cdot f^*(k)$, where k denotes the (unknown) keyword corresponding to t. Thus, if the attacker sets the candidate universe K_t to be the set of *all* keywords whose estimated frequencies are within distance $\epsilon \cdot f^*(k)$ of $f(t)$, the candidate universe will include the keyword corresponding to t at least 99% of the time. The problem with taking this approach, in general, is that the set K_t constructed this way may be too large.

If we assume a Zipfian distribution [3] for the keyword frequencies, however, then the size of K_t as constructed above is smallest when $f(t)$ is largest. (This is a consequence of the fact that the Zipfian distribution places high probability on a few items and low probability on many items.) In particular, then, the set of $2T/n$ keywords with estimated frequencies closest to $f(t)$ (as chosen by our algorithm), will "cover" all keywords within distance $\epsilon \cdot f^*(k)$ of $f(t)$ from $f(t)$—or, equivalently, the candidate universe will contain the true keyword k—with high probability.
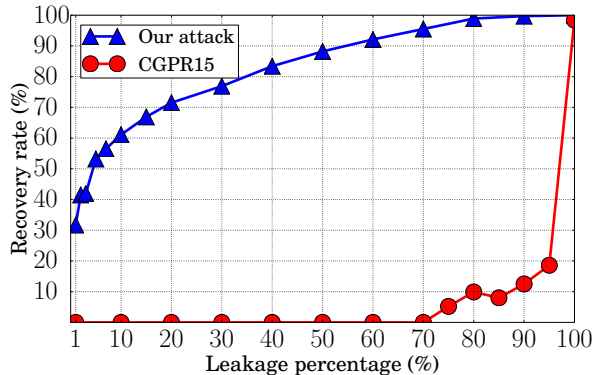
Figure 6: Recovering the keyword corresponding to a single token. Probability of recovering the correct keyword as a function of the percentage of files leaked.

5   Experiments

We simulate the attacks from Section 4. (We do not run any simulations for the binary-search attack described in Section 3, since this attack succeeds with probability 1, injecting a fixed number of emails.) We compare our attacks to our own implementation of the attacks by Cash et al. [4] (CGPR15). We do not compare with the attacks of Islam et al. [10] (IKK12), since their results are strictly dominated by those of CGPR15.

5.1   Setup

For our experiments we use the Enron email dataset [1], consisting of 30,109 emails from the "sent mail" folder of 150 employees of the Enron corporation that were sent between 2000–2002. We extracted keywords from this dataset as in CGPR15: words were first stemmed using the standard Porter stemming algorithm [18], and we then removed 200 stop words such as "to," "a," etc. Doing so results in approximately 77,000 keywords in total. In our experiments, we chose the top 5,000 most frequent keywords as our keyword universe (as in CGPR15).

We assumed the threshold countermeasure with $T = 200$. As discussed earlier, only 3% of the files contained more than this many keywords.

We could not find real-world query datasets for email. Therefore, in our experiments we choose the client's queries uniformly from the keyword universe, as in CGPR15. (However, our attacks do not use any information about the distribution of the queries.) Leaked files are chosen uniformly from the base set of 30,109 emails, and the percentage of leaked files was varied from 1% to 100%. For each value of the file-leakage percentage, we repeat the attack on 100 uniform sets of queries (containing either one token or 100 tokens) and 10 uniformly sampled sets of leaked files of the appropriate size; we report the average. We do not include error bars in our figures, but have observed that the standard deviation in our experiments is very small (less than 3% of the average).

5.2   Recovery of a Single Token

The performance of our attack for recovering the keyword associated with a single token (described in Section 4.2.1) is displayed in Figure 6. The server only needs to inject $\lceil \log 2T \rceil = 9$ files in order
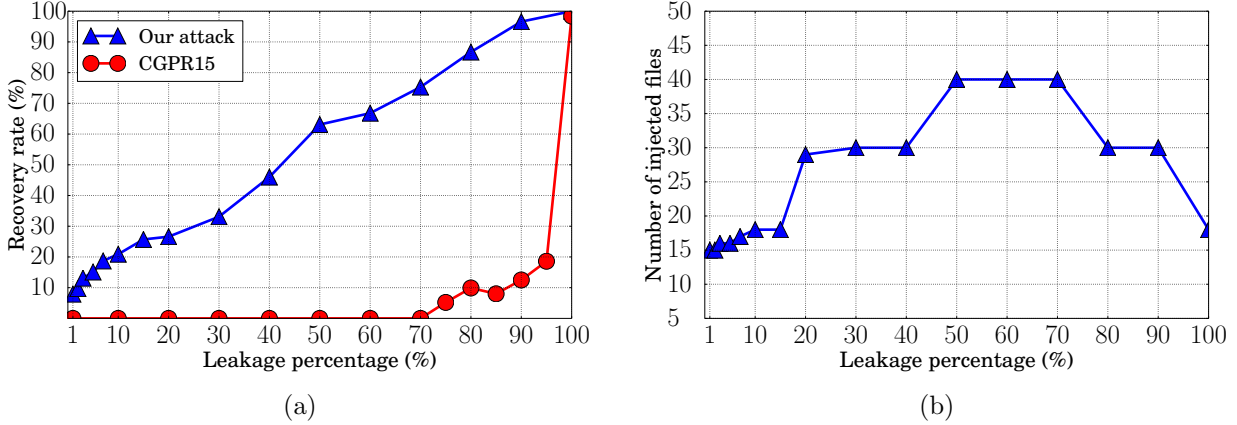
11

Figure 7: Recovering the keywords corresponding to 100 tokens. (a) Fraction of keywords recovered and (b) number of files injected as a function of the percentage of files leaked.

to carry out the attack. It can be observed that our attack performs quite well even with only a small fraction of leaked files, e.g., recovering the keyword about 70% of the time once only 20% of the files are leaked, and achieving 30% recovery rate even when given only 1% of the files.

Neither the IKK12 attack nor the CGPR15 attack applies when the server is given the search results of only a single token. To provide a comparison with our results, we run the CGPR15 attack by giving it the search results of 100 tokens (corresponding to uniformly chosen keywords) and then measure the fraction of keywords recovered. As shown in Figure 6, the CGPR15 attack recovers a keyword with probability less than 20% even when 95% of the client's files are leaked. Of course, our attack model is stronger than the one considered in CGPR15.

5.3 Recovery of Multiple Tokens

We have also implemented our attack from Section 4.2.2 which can be used to recover the keywords corresponding to multiple tokens. In our experiments, we target the recovery of the keywords associated with $m = 100$ tokens; we choose $n = 10$, and set δ as described in Section 4.2.2.

Figure 7a tabulates the fraction of keywords recovered by our attack, and compares it to the fraction recovered by the CGPR15 attack. (As noted in the previous section, the CGPR15 attack inherently requires search results for multiple tokens; this explains why the results for the CGPR15 attack in Figure 7a are almost identical to the results for their attack in Figure 6.) Both attacks do well when the fraction of leaked files is large, however the recovery rate of the CFPR15 attack drops dramatically as the fraction of leaked files decreases. In contrast, our attack continues to perform well, recovering 65% of the keywords given access to 50% of the client's files, and still recovering 20% of the keywords when only 10% of the client's files have been leaked. We stress that in our attack the server knows which keywords have been recovered correctly and which have not, something that is not the case for prior attacks.

Figure 7b shows the number of files that need to be injected in order to carry out our attack. The number of files injected never exceeds 40, and in many cases it is even less than that. We also highlight that the number of files injected to recover the keywords associated with 100 tokens is more than an order-of-magnitude smaller than 100× the number of files injected to recover the keyword associated with a single token in the previous section.
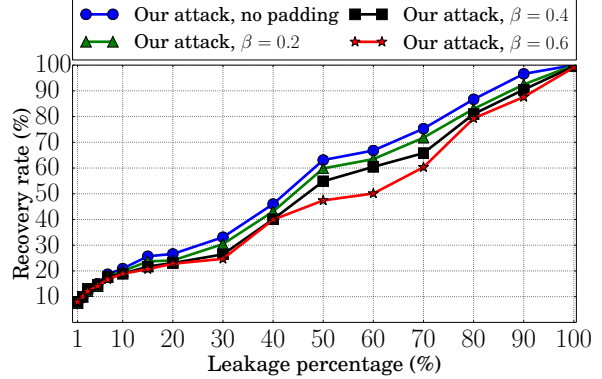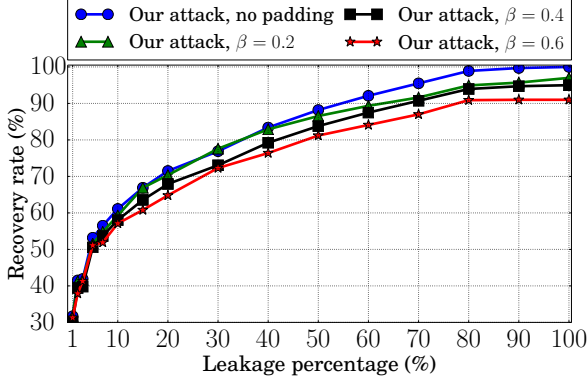
12

Figure 8: Recovering the keyword corresponding to a single token when keyword padding is used.

Figure 9: Recovering the keywords corresponding to 100 tokens when keyword padding is used.

The number of files injected by our attack first increases with the fraction of leaked files, and then decreases; we briefly explain why. The number of files injected in step 1 of our attack is independent of the fraction of leaked files. The number of files injected in step 2 of the attack depends on both the number of unrecovered tokens (i.e., the size of $\mathbf{t}_2$) and the average size of the candidate universe for each unrecovered token t' (i.e., the size of $K_{t'}$). When the fraction of leaked files is very small, the estimated joint frequencies are far from the true frequencies and, in particular, most estimated joint frequencies are 0; thus, many keywords are removed from $K_{t'}$ and hence the size of $K_{t'}$ is low. The net result is that the recovery rate is small, but so is the number of injected files. As the fraction of leaked files increases, more keywords are included in $K_{t'}$, leading to higher recovery rate but also more injected files. When the fraction of leaked files becomes very high, however, the estimated frequencies are very close to the true frequencies and so more keywords are recovered in step 1 of the attack. This leaves fewer unrecovered tokens in step 2, leading to fewer injected files overall even as the recovery rate remains high.

6   Ineffectiveness of Keyword Padding

Prior work [10, 4] suggests *keyword padding* as another potential countermeasure for attacks that exploit the file-access pattern. The basic idea is to distort the real frequency of each keyword k by randomly associating files that do not contain that keyword with k; this is done at setup time, when the client uploads its encrypted files to the server. One version of the countermeasure [4] ensures that the number of files returned in response to any search result is a multiple of an integer λ. A stronger version of the countermeasure [10] involves performing the padding in such a way that for any keyword k there are at least $\alpha - 1$ other keywords having the same frequency. These countermeasures defeat the attacks in prior work, but we show that they have little effect on our attacks.

We remark that keyword padding seems difficult to apply in the dynamic setting, where new files are uploaded after the initial setup done by the client. The dynamic case is not discussed in [10, 4].

6.1 Binary-/Hierarchical-Search Attacks

Even when keyword padding is used, our binary-search and hierarchical-search attacks will recover the keyword k corresponding to some token t unless one of the injected files that does not contain k is returned in response to the search using t. We show that the probability of this bad event is small, focusing on the binary-search attack for concreteness. Say ℓ of the files contain k and that, after keyword padding, an additional $\beta \cdot \ell$ random and independently chosen files (in expectation) that do not contain k are returned in response to the search using t. (By setting parameters appropriately, this roughly encompasses both the countermeasures described above.) Now consider some file injected as part of the binary-search attack that does not contain k. The probability that this file is chosen as one of the spurious files returned in response to the search using t is $\beta\ell/(F-\ell)$, where F is the total number of files (including the injected files). Since $\lceil \log|K| \rceil$ files are injected, the overall probability that the bad event occurs is at most

$$1 - \left(1 - \frac{\beta\ell}{(F-\ell)} \right)^{\lceil \log|K| \rceil}.$$

In fact, this is an over-estimate since if k is uniform then on average only half the injected files contain k.

For the Enron dataset with $|K| = 5,000$, $F = 30,109$, $\ell = 560$, and $\beta = 0.6$, and assuming half the injected files contain the keyword in question, the probability that the binary-search attack succeeds is 0.93. (In fact, $\beta = 0.6$ is quite high, as this means that more than $1/3$ of the files returned in response to a query do not actually contain the searched keyword.) With $\beta = 0.6$ the IKK12 and CGPR15 attacks recover no keywords at all.

6.2 Attacks with Partial File Leakage

Although our attacks with partial file leakage use information about keyword frequencies and joint frequencies, they are still not significantly affected by the padding countermeasures. The reason is that although the padding ensures that a given frequency no longer suffices to uniquely identity a keyword, the frequency of any particular keyword doesn't change very much. Thus, the exact frequency and the estimated frequency of any keyword remain close even after the padding is done, and the underlying keyword is still likely to be included in the candidate universe of a target token. As long as this occurs, the search step recovers the token with high probability as discussed in the previous section. This is even more so the case with regard to joint frequencies, since these do not change unless two keywords are both associated with the same random file that contains neither of those keywords, something that happens with low probability.

To validate our argument, we implement the padding countermeasure proposed in [4] and repeat the experiments using our attacks. As shown in Figures 8 and 9, the recovery rate of our attacks degrades only slightly when keyword padding is used.

Figure 10 compares the effectiveness of our attack to the CGPR15 attack when keyword padding is used. The recovery rate of the CGPR15 attack drops dramatically in the presence of this countermeasure. In particular, it recovers only 57% of the tokens even with 100% file leakage when $\beta = 0.2$, and recovers nothing even with 100% file leakage when $\beta = 0.6$. In contrast, our attack still recovers almost the same number of keywords as when no padding is used.
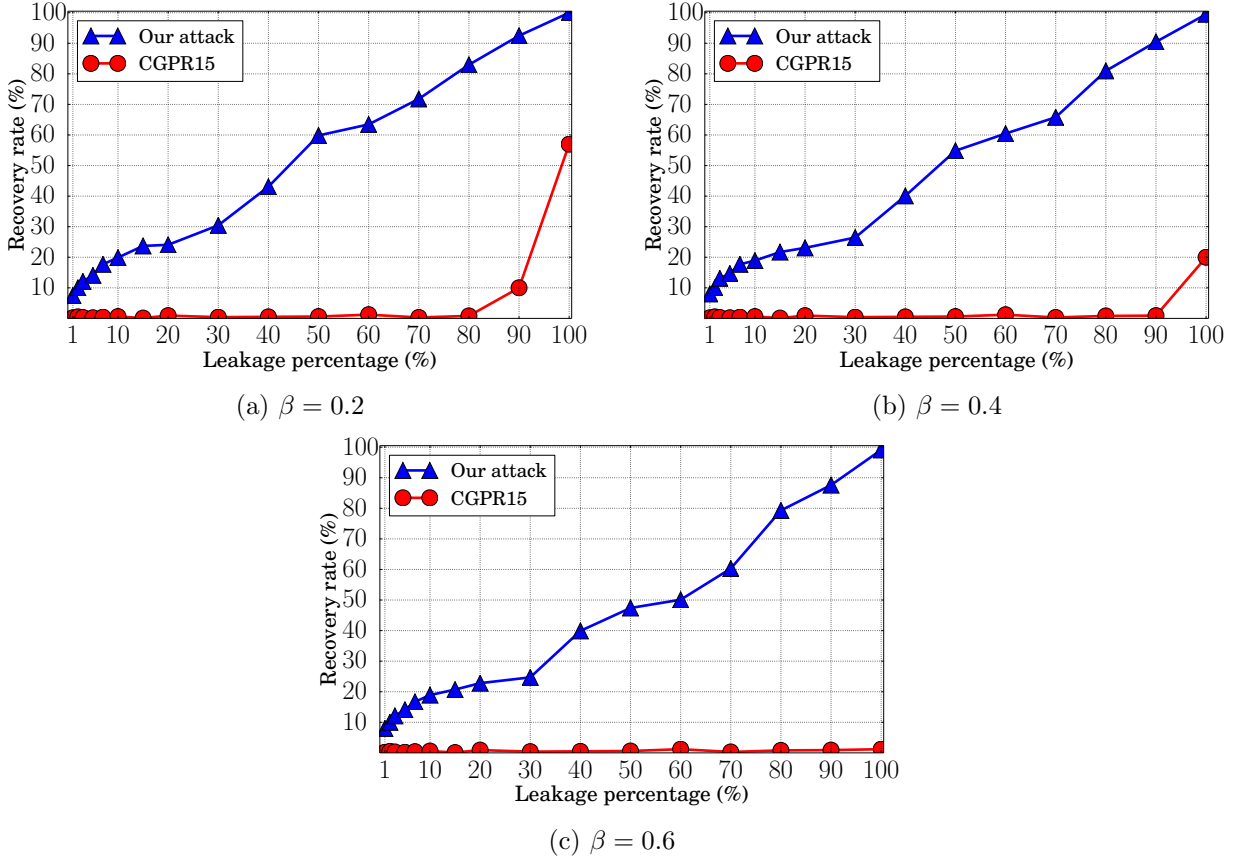
(a) $\beta = 0.2$

(b) $\beta = 0.4$

(c) $\beta = 0.6$

Figure 10: Recovering the keywords corresponding to 100 tokens when keyword padding is used, plotted for different β.

7 Extensions to Conjunctive SE

SE schemes supporting conjunctive queries allow the client to request all files containing some collection of keywords $k_1, k_2, \ldots, k_d$. The naive way to support conjunctive queries is to simply have the client issue queries for each of these keywords individually; the server can compute the set of file identifiers S_i containing each keyword k_i and then take their intersection to give the final result. Such an approach leaks more information than necessary: specifically, it leaks each of $S_1, \ldots, S_d$ rather than the final result $\cap S_i$ alone. We refer to $\cap S_i$ as the *ideal access-pattern leakage* for a conjunctive query, and show attacks based only on such ideal leakage. We remark, however, that no known efficient SE scheme achieves ideal leakage. For example, the scheme by Cash et al. [6] leaks $S_1, S_1 \cap S_2, S_1 \cap S_3, \ldots, S_1 \cap S_d$. Such additional leakage can only benefit our attacks.

Throughout this section, we assume the threshold countermeasure is not used and so injected files can contain any number of keywords. (Our attacks here could be generalized as done previously in case the threshold countermeasure is used.)

15

7.1 Queries with Two Keywords

We first present a non-adaptive attack to recover the keywords used in a conjunctive query involving two keywords. As in the non-adaptive attacks in prior sections, the attacker can recover the keywords corresponding to any future queries after injecting some initial set of files.

The idea is the following. Say the conjunctive search query involves keywords k_1 and k_2, and we can partition the universe of keywords into two sets K_1 and K_2 with $k_1 \in K_1$ and $k_2 \in K_2$. We can then use a variant of the binary-search attack in which we inject files generated by Inject_Files(K_1), where we additionally include all keywords in K_2. Since these files always contain k_2, the search results of the conjunctive query on these injected files is exactly the same as the search results of k_1 on these files, and we can thus recover k_1 as before. We can proceed analogously to recover k_2.

The problem with the above is that we do not know, *a priori*, how to partition K into sets K_1, K_2 as required. Instead, we generate a sequence of $\log|K|$ partitions $\{(K_1^i, K_2^i)\}$ such that for some partition i it holds that $k_1 \in K_1^i$ and $k_2 \in K_2^i$. This is done by simply letting K_1^i be the set of all keywords whose ith bit is 0, and K_2^i be the complement. Since k_1 and k_2 are distinct, they must differ on at least one position, say i, and satisfy the desired separation property on the ith partition. By repeating the attack described earlier for each partition, we obtain an attack using $\log^2|K| + \log|K|$ injected files (after removing duplicates). The attack is described in detail in Figure 11.

Let q be a conjunctive query with two keywords.

Algorithm F ← Inject_Files_Disjoint(K_1, K_2)

1: **F** ← Inject_Files(K_1).
2: Include all keywords in K_2 in every file in **F**.

Algorithm F ← Inject_Files_Conjunctive(K)

1: **for** $i = 1, 2, \ldots, \log|K|$ **do**
2:    Let K_1^i contain keywords whose ith bit is 0, and let $K_2^i = K \setminus K_1^i$.
3:    Generate file F_1^i that contains all keywords in K_1^i and file F_2^i that contains all keywords in K_2^i.
4:    $\mathbf{F}_1^i$ ← Inject_Files_Disjoint(K_1^i, K_2^i).
5:    $\mathbf{F}_2^i$ ← Inject_Files_Disjoint(K_2^i, K_1^i).
6: Output $\mathbf{F} = \{F_1^i, F_2^i, \mathbf{F}_1^i, \mathbf{F}_2^i,$ for all $i\}$.

Algorithm k ← Recover_Conjunctive$(q, K, \mathbf{F})$

1: Let $R_q = \{r_1^i, r_2^i, R_1^i, R_2^i\}$ for $i = 1, \ldots, \log|K|$ be the search result of query q on the files **F** described above.
2: Find i such that neither F_1^i nor F_2^i is in the search result (i.e., $r_1^i = r_2^i = 0$).
3: k_1 ← Recover(R_1^i, K_1^i).
4: k_2 ← Recover(R_2^i, K_2^i).
5: Output (k_1, k_2).

Figure 11: Non-adaptive attack for a conjunctive query involving two keywords.

Given ideal access-pattern leakage, the above attack above only works for conjunctive queries involving two keywords. For conjunctive searches using the SE scheme of Cash et al. [6], though,

the above attack can be extended to work for conjunctive queries involving any number of keywords since the pairwise intersections are leaked as described earlier.

7.2 Queries with Multiple Keywords

The attack in Section 7.1 only works for conjunctive queries involving two keywords, and uses $O(\log^2 |K|)$ injected files. Here we present a non-adaptive attack that can recover conjunctive queries involving any number of keywords using only $O(\log |K|)$ injected files, and still assuming only ideal access-pattern leakage. In contrast to the previous attack, however, this attack does not always succeed.

Consider a conjunctive query q involving d keywords. The basic idea is to inject n files, each containing L keywords selected uniformly and independently from the keyword universe. If parameters are set appropriately, the search result on q will include some of the injected files with high probability. By definition, each of those files contains all d keywords involved in the query, and hence the intersection of those files also contains all those keywords. We claim that when parameters are set appropriately, the intersection contains no additional keywords. Thus, the server recovers precisely the d keywords involved in the query by simply taking the intersection of the injected files returned in response to the query. The following theorem formalizes this idea.

Theorem 1. Let $L = (\frac{1}{2})^{1/d}|K|$ and $n = (2 + \varepsilon)d\log|K|$ with $\varepsilon \geq 0$. Then the success probability of the attack is roughly $e^{-1/|K|^{\varepsilon/4}}$.

Proof. Fix some conjunctive query q involving d keywords. The probability that any particular injected file matches the query is approximately $(L/|K|)^d = 1/2$ since each of the d keywords is included in the file with probability roughly $L/|K|$. Since each file is generated independently, the expected number of files that match the query is $n/2$; moreover, the number n' of files that match the query follows a binomial distribution and so the Chernoff bound implies

$$\Pr\left[\left|n' - \frac{n}{2}\right| \geq \theta\frac{\sqrt{n}}{2}\right] \leq e^{-\theta^2/2}.$$

Setting $\theta = \frac{\varepsilon\sqrt{n}}{2(2+\varepsilon)}$, we have

$$\Pr\left[n' \leq \left(1 + \frac{\varepsilon}{4}\right)d\log|K|\right] \leq e^{-\frac{\theta^2}{2}}.$$

Thus, $n' > (1 + \frac{\varepsilon}{4})d\log|K|$ with overwhelming probability.

The probability that any *other* keyword is in all these n' files is extremely low. Specifically, for any fixed keyword not involved in the query, the probability that it lies in all n' files is $(L/|K|)^{n'}$. Thus, the probability that *no* other keyword lies in all n' files is

$$\left(1 - \left(\frac{L}{|K|}\right)^{n'}\right)^{|K|-d} \approx \left(1 - \frac{1}{|K|^{1+\varepsilon/4}}\right)^{|K|}$$

(assuming $d \ll |K|$). The above simplifies to $e^{-1/|K|^{\varepsilon/4}}$. $\qquad\square$

Note that for any $\varepsilon > 0$ the bound given by the theorem approaches 1 as $|K|$ tends to infinity. We experimentally verified the bound in the theorem for $|K| = 5,000$ and $d = 3$. For example,

setting $\varepsilon = 1$ we obtain an attack in which the server injects $n = 110$ files with $L = 3,969$ keywords each, and recovers all keywords involved in the conjunctive query with probability 0.97. For completeness, we remark that the server can tell whether it correctly recovers all the keywords or not, assuming d is known.

7.3   An Adaptive Attack

We can further reduce the number of injected files using an adaptive attack. The idea is to recover the keywords involved in the query one-by-one, starting with the lexicographically largest, using an adaptive binary search for each keyword. The server first injects a file containing the first $|K|/2$ keywords. There are two possibilities:

1. If this file is in the search result for the query, the server learns that all the keywords involved in the query have index at most $|K|/2$. It will next inject a file containing the first $|K|/4$ keywords.

2. If this file is not in the search result for the query, the server learns that at least one keyword involved in the query has index greater than $|K|/2$. It will next inject a file containing the first $3|K|/4$ keywords.

Proceeding in this way, the server learns the lexicographically largest keyword using $\log |K|$ injected files. Once that keyword k_d is recovered, the server repeats this attack but with k_d always included in the injected files to learn the next keyword, and so on. See Figure 12.

The number of injected files is $d \log |K|$. We remark that d need not be known in advance, since the attacker can determine d during the course of the attack. It is also worth observing that the number of injected files is essentially optimal for a deterministic attack with success probability 1, because the search results on $d \log |K|$ files contain at most $d \log |K|$ bits of information, which is roughly the entropy of a conjunctive search involving d keywords from a universe of size $|K|$.

8   Additional (Potential) Countermeasures

In this section, we briefly discuss some other potential countermeasures against our attacks.

Semantic filtering. One may be tempted to think that the files injected by our attacks will not "look like" normal English text, and can therefore be filtered easily by the client. We argue that such an approach is unlikely to prevent our attacks. First, although as described our attacks inject files containing arbitrary sets of keywords, the server actually has some flexibility in the choice of keywords; e.g., the binary-search attack could be modified to group sets of keywords that appear naturally together. Second, within each injected file, the server can decide the order and number of occurrences of the keywords, can choose variants of the keywords (adding "-ed" or "-s," for example), and can freely include non-keywords ("a," "the," etc.) There are several tools (e.g., [21]) that can potentially be adapted to generate grammatically correct text from a given set of keywords by ordering keywords based on *n-grams* trained from leaked files and simple grammatical rules. A detailed exploration is beyond the scope of our paper.

Batching updates. As mentioned in Section 2, even if the client shuffles the file identifiers and pads all files to the same length, the server can identify an injected file based on the time at which it is inserted by the client. This suggests a (partial) countermeasure that can be used in dynamic

Let q be a conjunctive query with keyword $k_1, \ldots, k_d$.

Algorithm $\mathbf{k} \leftarrow$ Attack_Conjunctive(q, K)

1: Initialize $\mathbf{k} = \emptyset$.
2: **for** $i = d, \ldots, 1$ **do**
3:     Set $K_i = K \setminus \mathbf{k}$, set $b = |K_i|/2$.
4:     **for** $j = 2, \ldots, \log |K_i|$ **do**
5:         Inject F that contains the first b keywords
            in K_i and all keywords in $\mathbf{k}$.
6:         Let R_q be the search result of query q on F.
7:         **if** $R_q = 1$ **then**
8:             $b = b - |K_i|/2^j$.
9:         **else**
10:             $b = b + |K_i|/2^j$.
11:     Inject F that contains the first b keywords in K_i and all keywords in $\mathbf{k}$.
12:     Let R_q be the search result of query q on F.
13:     **if** $R_q = 1$ **then**
14:         Recover k_i as the bth keyword in K_i.
15:     **else**
16:         Recover k_i as the $(b+1)$th keyword in K_i.
17:     $\mathbf{k} = \mathbf{k} \cup \{k_i\}$.

Figure 12: An adaptive attack for conjunctive queries involving d keywords.

SE schemes that support updates: rather than uploading each new file as it arrives, the client should wait until there are several (say, B) new files and then upload this "batch" of B files at once. Assuming only one of those files was injected by the server, this means the server only learns that the injected file corresponds to one of B possibilities.

This countermeasure can be trivially circumvented if the server can inject B files before any other new files arrive. (If the server additionally has the ability to mount chosen-query attacks—something we have not otherwise considered in this paper—then the total number of injected files remains the same.) Even if the server can inject only $B' < B$ identical[3] files into a single "batch," the server knows that if fewer than B' files from this batch are returned in response to some query, then the injected files do not match that query. Finally, even if the server can only inject a single file per "batch," the server can inject the same file repeatedly and with high confidence determine based on the search results whether the file matches some query. We leave a more complete analysis of this countermeasure for future work.

9 Conclusions

Our paper shows that file-injection attacks are devastating for query privacy in searchable encryption schemes that leak file-access patterns. This calls into question the utility of searchable encryption, and raises doubts as to whether existing SE schemes represent a satisfactory tradeoff

---

[3]The files need not be identical; they only need to contain an identical set of keywords.

between their efficiency and the leakage they allow. Nevertheless, we briefly argue that searchable encryption may still be useful in scenarios where file-injection attacks are not a concern, and then suggest directions for future research.

We have argued that file-injection attacks would be easy to carry out in the context of searching email. But in "closed systems," where a client is searching over records generated via some other process, file-injection attacks may not be possible or may be much more difficult to carry out. Additionally, there may be settings—e.g., when all files have the same length because they share some particular format—where even though the server can inject files, it may not be able to associate file identifiers with specific files it has injected. It is worth noting also that there may be applications of SE in which the server is trusted (and so, in particular, can be assumed not to carry out file-injection attacks), and the threat being defended against is an external attacker who compromises the server.[4]

Our work and previous work [10, 4] demonstrate that leaking file-access patterns in their entirety is dangerous, and can be exploited by an attacker to learn a significant amount of sensitive information. We suggest, therefore, that future research on searchable encryption focus on reducing or eliminating this leakage rather than accepting it as the default. Our work also highlights the need to design efficient schemes satisfying forward privacy. Addressing these challenges may require exploring new directions, such as interactive protocols [17] or multiple servers. It would also be of interest to explore lower bounds on the efficiency that searchable encryption can achieve as a function of how much about the file-access pattern is leaked.

References

[1] Enron email dataset. `https://www.cs.cmu.edu/~./enron/`. Accessed: 2015-12-14.

[2] Pmail. `https://github.com/tonypr/Pmail`, 2014.

[3] ADAMIC, L. A., AND HUBERMAN, B. A. Zipfs law and the internet. *Glottometrics 3*, 1 (2002), 143–150.

[4] CASH, D., GRUBBS, P., PERRY, J., AND RISTENPART, T. Leakage-abuse attacks against searchable encryption. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security* (2015), ACM, pp. 668–679.

[5] CASH, D., JAEGER, J., JARECKI, S., JUTLA, C. S., KRAWCZYK, H., ROSU, M.-C., AND STEINER, M. Dynamic searchable encryption in very-large databases: Data structures and implementation. *IACR Cryptology ePrint Archive 2014* (2014), 853.

[6] CASH, D., JARECKI, S., JUTLA, C., KRAWCZYK, H., ROŞU, M.-C., AND STEINER, M. Highly-scalable searchable symmetric encryption with support for boolean queries. In *Advances in Cryptology–CRYPTO 2013*. Springer, 2013, pp. 353–373. Full version available at `http://eprint.iacr.org`.

[7] CHANG, Y.-C., AND MITZENMACHER, M. Privacy preserving keyword searches on remote encrypted data. In *Applied Cryptography and Network Security* (2005), Springer, pp. 442–455.

[8] CURTMOLA, R., GARAY, J., KAMARA, S., AND OSTROVSKY, R. Searchable symmetric encryption: improved definitions and efficient constructions. In *Proceedings of the 13th ACM conference on Computer and communications security* (2006), ACM, pp. 79–88.

[9] GOH, E.-J., ET AL. Secure indexes. *IACR Cryptology ePrint Archive 2003* (2003), 216.

[10] ISLAM, M. S., KUZU, M., AND KANTARCIOGLU, M. Access pattern disclosure on searchable encryption: Ramification, attack and mitigation. In *19th Annual Network and Distributed System Security Symposium, NDSS 2012, San Diego, California, USA, February 5-8, 2012* (2012).

---

[4]Though even here one must be careful since an external attacker might have the ability to inject files, and/or be able to learn file-access patterns from the client-server communication (e.g., based on file lengths) without compromising the server.

[11] KAMARA, S., AND PAPAMANTHOU, C. Parallel and dynamic searchable symmetric encryption. In *Financial cryptography and data security.* Springer, 2013, pp. 258–274.

[12] KAMARA, S., PAPAMANTHOU, C., AND ROEDER, T. Dynamic searchable symmetric encryption. In *the ACM Conference on Computer and Communications Security, CCS'12, Raleigh, NC, USA, October 16-18, 2012* (2012), pp. 965–976.

[13] LAU, B., CHUNG, S., SONG, C., JANG, Y., LEE, W., AND BOLDYREVA, A. Mimesis aegis: A mimicry privacy shield–a systems approach to data privacy on public cloud. In *23rd USENIX Security Symposium (USENIX Security 14)* (2014), pp. 33–48.

[14] LIU, C., ZHU, L., WANG, M., AND TAN, Y.-A. Search pattern leakage in searchable encryption: Attacks and new construction. *Information Sciences 265* (2014), 176–188.

[15] NAVEED, M. The fallacy of composition of oblivious ram and searchable encryption. Tech. rep., Cryptology ePrint Archive, Report 2015/668, 2015.

[16] NAVEED, M., PRABHAKARAN, M., AND GUNTER, C. A. Dynamic searchable encryption via blind storage. In *Security and Privacy (SP), 2014 IEEE Symposium on* (2014), IEEE, pp. 639–654.

[17] POPA, R. A., LI, F. H., AND ZELDOVICH, N. An ideal-security protocol for order-preserving encoding. In *Security and Privacy (SP), 2013 IEEE Symposium on* (2013), IEEE, pp. 463–477.

[18] PORTER, M. F. An algorithm for suffix stripping. *Program 14,* 3 (1980), 130–137.

[19] SONG, D. X., WAGNER, D., AND PERRIG, A. Practical techniques for searches on encrypted data. In *Security and Privacy, 2000. S&P 2000. Proceedings. 2000 IEEE Symposium on* (2000), IEEE, pp. 44–55.

[20] STEFANOV, E., PAPAMANTHOU, C., AND SHI, E. Practical dynamic searchable encryption with small leakage. In *NDSS* (2014), vol. 14, pp. 23–26.

[21] UCHIMOTO, K., ISAHARA, H., AND SEKINE, S. Text generation from keywords. In *Proceedings of the 19th international conference on Computational linguistics-Volume 1* (2002), Association for Computational Linguistics, pp. 1–7.