Practical Passive Leakage-Abuse Attacks Against Symmetric Searchable Encryption

Matthieu Giraud[1], Alexandre Anzala-Yamajako[2], Olivier Bernard[2], and Pascal Lafourcade[1]

[1] LIMOS, Université Clermont Auvergne
BP 10448, F-63000, Clermont-Ferrand, France,
firstname.lastname@uca.fr
[2] Laboratoire CHiffre, Thales Communication & Security
4 avenue des Louvresses 92622 Gennevilliers,
{alexandre.anzalayamajako,olivier.bernard2}@thalesgroup.com

Abstract. The problem of securely outsourcing client data with search functionality has given rise to efficient solutions called *Symmetric Searchable Encryption* (SSE) schemes. These schemes are provably secure with respect to an explicit leakage profile; however, determining how much information can be inferred in practice from this leakage remains difficult. First, we refine and formalize the leakage hierarchy introduced by Cash et al. in 2015. Second, we further extend the analysis of existing attacks to better understand their real-world efficiency and the practicality of their hypothesis. Finally, we present the first complete practical attacks on L4, L3 and L2 leakage profiles. Our attacks are passive and only assume the very realistic knowledge of a small sample of plaintexts; moreover, we show their devastating effect on real-world datasets.

Keywords: symmetric searchable encryption, leakage, passive attacks.

1 Introduction

With the growing importance of digital data in everyday life, it is necessary to have backups and access from anywhere. Outsourcing this digital data to a cloud provider is enticing; however, some data, such as legal, banking or medical documents, industrial patents or simply our emails can be sensitive or confidential. Though encryption classically provides confidentiality, it prevents any processing of the client data by the cloud provider, thus making access to it cumbersome.

First introduced by Song et al. [18], *Symmetric Searchable Encryption* (SSE) schemes give to the server the ability to perform keyword searches on encrypted data by means of a *protected database*. While the single keyword query is the basic functionality of an SSE scheme, there also exist SSE schemes expanding the expressiveness of search queries to boolean [8], range and substring [12] queries, as well as *dynamic* SSE schemes [7, 16] allowing the client to add new encrypted documents to the protected database while retaining the search capability.

The amount of information leaked by a given SSE scheme to the server is formalized by a *leakage function* [10, 16]. The security of the scheme then relies

on proving that this function does not leak more information than expected. However, it can be used by an honest-but-curious server [13], which dutifully executes the scheme but tries to deduce information on the stored documents. In [6], based on deployed SSE schemes, Cash et al. define four *leakage profiles* L4, L3, L2 and L1, L4 being the most leaky and L1 the least. Commercially available SSE solutions such CipherCloud [2], Bitglass [1] or Skyhigh Networks [3] are L4 or L3 schemes. Assessing the practical impact of each of these profile on the server knowledge of the protected data is critical for real life applications.

Leakage-abuse attacks. The first experimental attack on SSE schemes is the *IKK Attack*, proposed by Islam et al. in [15]. Its goal is to associate search tokens to actual keywords, exploiting the data access pattern revealed by client queries and assuming the adversary has access to a co-occurrence matrix that gives the probability for two keywords to appear in a randomly chosen document. Actually, as noted by Cash et al. in [6], this matrix needs to be so precise for the attack to succeed, that it seems legitimate to suppose the adversary has access to the number of documents in which every keyword appears. With this strong extra knowledge, they mount a more effective attack named the *Count Attack* [6]. Both attacks target leakage profiles beyond L2, but the strength of their assumptions questions their practicality. Additionally, Cash et al. propose in [6] a passive partial document recovery attack for L3-SSE schemes when the adversary knows plaintext-ciphertext pairs. For an *active* adversary able to plant chosen documents in the database, they present a partial document recovery attack on L3- and L2-SSE schemes [6]. With the extra ability to issue selected queries, Zhang et al. [19] mount a query recovery attack that works on any dynamic SSE scheme. These active attacks are very efficient as only few injected files suffice to reveal the association between keywords and search tokens.

Our contributions. We present the first passive attacks on L4, L3 and L2 leakage profiles which only assume the very realistic knowledge of a small sample of plaintexts in addition to the protected database given to the server. In particular, our attacks do not rely on observing search queries, nor do they require the adversary to know any plaintext-ciphertext pairs. Their efficiency and practicality are demonstrated on several real-world datasets such as the mailing-list of *Lucene* Apache project [5]. We also formalize the leakage hierarchy described by Cash et al. in [6] and reintroduce a transitional $\mathsf{L1}^+$ leakage profile between L2 and L1 profiles in order to better classify existing attacks.

Outline. In Section 2, we provide background on SSE schemes and their security. Section 3 details formally the leakage hierarchy of Cash et al. [6] and reintroduces what we call the $\mathsf{L1}^+$ profile; this refined hierarchy is used in Section 4 to put previous attacks into perspective. Section 5 is devoted to the description of our new passive attacks, whose effectiveness is demonstrated in Section 6.

2 Symmetric Searchable Encryption

A Symmetric Searchable Encryption (SSE) scheme is a protocol between a client and a server. The client owns a sensitive dataset but has limited computational power and storage capacity. The server has a large storage space and high processing power, but is not trusted by the client except for executing correctly the search protocol. We introduce some notations, then we formalize SSE schemes and discuss the associated security notion.

2.1 Notations

Sequences, lists and sets. A sequence of elements is defined here as an ordered set where repetitions are allowed. A list is therefore an ordered set where all elements are distinct. Finally, a set is defined as a bunch of distinct elements with no notion of ordering. Sequences are guarded by $(\dots)$, lists are denoted by square brackets $[\dots]$ and sets by braces $\{\dots\}$. The number of elements of a set E (resp. list or sequence) is written $\#E$.

We also denote by $\{t\}$ the set of elements of a list t where the order of the elements is not considered. Similarly, $[s]$ denotes the list composed of the elements of a sequence s in their order of first appearance without repetition.

Documents and keywords. Let $\mathsf{W} = \{w_1, \dots, w_m\}$ be a dictionary composed of m distinct keywords and $\mathsf{DB} = \{d_1, \dots, d_n\}$ a set of n documents made of keywords from W. Each document d_i is a sequence of length ℓ_i, formally $d_i = (w_{i_1}, \dots, w_{i_{\ell_i}}) \in \mathsf{W}^{\ell_i}$. DB is called the *dataset*. We denote by W_i the set of distinct keywords of the document d_i i.e., $\mathsf{W}_i = \big\{[d_i]\big\}$.

The same objects are described server-side by introducing the star superscript. Hence, $\mathsf{W}^* = \{w_1^*, \dots, w_m^*\}$ denotes the set of search tokens associated to the keywords of W. Similarly, $\mathsf{DB}^* = \{d_1^*, \dots, d_n^*\}$ is the set of ciphertexts of DB where d_i^* is the encryption of d_i, and W_i^* is the set of tokens associated to d_i^*. As to emphasize the fact that the association between d_i and d_i^* is not known to the server a priori, an identifier id_i is used to uniquely represent d_i^*. A datastructure EDB is also provided, which contains protected metadata that allows the server to answer search queries.

The list of all the indices i such that $d_i \in \mathsf{DB}$ contains the keyword w is denoted by $\mathsf{DB}(w)$. The notation N denotes the number of pairs (d, w) where $d \in \mathsf{DB}$ and $w \in d$ i.e., $N = \#\big\{\,(d, w) \;\big|\; d \in \mathsf{DB}, w \in d\,\big\}$. Note that, as it corresponds to a lower bound on the size of EDB, N can always be computed by the server. Server-side, the list of the identifiers of all the documents $d_i^* \in \mathsf{DB}^*$ associated to the search token w^* is written $\mathsf{EDB}(w^*)$. We stress that this information is not accessible directly from w^* and DB^*, thus the need of the extra protected metadata structure EDB.

2.2 Static SSE Schemes

Basic SSE schemes are defined by a symmetric encryption scheme together with an algorithm for setup and another for search.

As a first step, the client creates two datastructures DB^* and EDB as introduced above. Both datastructures are then uploaded to the server. As a second step, when the client wants to search for a specific keyword w, he computes the associated search token w^* with his secret key and sends w^* to the server. From w^* and EDB the server is able to return the identifiers of all encrypted documents matching the client's search. With the list of identifiers the client retrieves the encrypted documents, from which he can obtain the plaintext documents. We stress that the server should not be able to learn anything about the client's query or the returned documents during the process. For further details, see the Definition 1 of Appendix A inspired from Curtmola et al. [10].

This formalizes *static* SSE schemes. Static SSE schemes allow the client to initialize a protected database that supports keyword searches but cannot be updated, by opposition to *dynamic* SSE schemes. Dynamic schemes can be attacked at least as well as static schemes, so we do not introduce them formally.

2.3 Security of SSE Schemes

Introduced by Curtmola et al. in [10] and by Kamara et al. in [16], the *leakage function* $\mathcal{L}$ of a SSE scheme is a set of information revealed by the SSE scheme to the server. This leakage function formalizes information that EDB and the client queries reveal to the server.

The SSE scheme is said to be $\mathcal{L}$-secure iff any polynomial-time adversary making a sequence Q of queries (i.e., keywords of W) can successfully tell with only negligible probability whether the protocol is honestly executed or simulated from the leakage function $\mathcal{L}$.

The $\mathcal{L}$-security proves that no information is leaked by the SSE scheme to the server outside of what is exposed by the leakage function. Nevertheless, it does not tell us what is implied *in practice* on the knowledge of the protected data. Answering that question is the primary motivation of this paper.

3 A Leakage Hierarchy

We formally define classes of SSE schemes according to how much information the protected database leaks, as first introduced by Cash et al. in [6].

3.1 L4 Leakage Profile

Without any semantic consideration, a document is characterized by its number of words, their orders and their occurence counts. Moreover, it is possible to determine which words are shared with any other document. L4-SSE schemes reveal all this information, so nothing is lost about the plaintext non-semantic structure. Thus, a SSE scheme of leakage function $\mathcal{L}$ is of class L4 iff:

$$\mathcal{L}(\mathsf{EDB}) = \left\{(w^*_{i_1}, \ldots, w^*_{i_{\ell_i}})\right\}_{1 \leqslant i \leqslant n}.$$

Example 1. We use the following setup as a running example to illustrate the different amounts of leakage revealed to the server. Let $\mathsf{W} = \{\text{as}, \text{call}, \text{i}, \text{if}, \text{me}, \text{possible}, \text{soon}, \text{you}\}$ and d_1 and d_2, two documents defined over W where $d_1 = (\text{call}, \text{me}, \text{as}, \text{soon}, \text{as}, \text{possible})$ and $d_2 = (\text{i}, \text{call}, \text{you}, \text{if}, \text{possible})$. Assume that the search tokens associated to keywords of W are the following:

Keywords (W)	as	call	i	if	me	possible	soon	you
Tokens (W*)	14	76	33	11	25	35	78	10

Under L4 leakage, EDB reveals to the server $(76, 25, 14, 78, 14, 35) \to \mathsf{id}_1$ and $(33, 76, 10, 11, 35) \to \mathsf{id}_2$. The server knows that the document identified by id_1 is of length 6 and has five distinct keywords; it also knows that one keyword, associated to the token 14, is repeated twice. The document identified by id_2 contains exactly five distinct keywords and shares two keywords with the first document, namely those represented by tokens 35 and 76.

3.2 L3 Leakage Profile

For keyword search purposes, it is not necessary to know the occurrence count of each keyword. L3-SSE schemes of leakage function $\mathcal{L}$ do not reveal this, i.e.:

$$\mathcal{L}(\mathsf{EDB}) = \left\{\mathsf{L3}_{\mathsf{EDB}}(\mathsf{id}_i)\right\}_{1\leqslant i\leqslant n}, \quad \text{where } \mathsf{L3}_{\mathsf{EDB}}(\mathsf{id}_i) = \left[(w^*_{i_1}, \ldots, w^*_{i_{\ell_i}})\right].$$

Example 2. Resuming Example 1, the information revealed by an L3-SSE scheme about d_1 and d_2 is: $(76, 25, 14, 78, 35) \to \mathsf{id}_1$ and $(33, 76, 10, 11, 35) \to \mathsf{id}_2$. The server does not know anymore that the token 14 is associated twice to id_1.

3.3 L2 Leakage Profile

SSE schemes of class L2 only reveal the set of tokens of a document. The server can however still determine which documents contain a given token. A SSE scheme of leakage function $\mathcal{L}$ is of class L2 iff:

$$\mathcal{L}(\mathsf{EDB}) = \left\{W^*_i\right\}_{1\leqslant i\leqslant n}.$$

Example 3. Resuming Example 1, an L2-SSE scheme reveals about d_1 and d_2 : $(14, 25, 35, 76, 78) \to \mathsf{id}_1$ and $(10, 11, 33, 35, 76) \to \mathsf{id}_2$. We stress that the token order is not preserved in EDB: we arbitrarily sorted the token in ascending order, thus the server does not know their initial order.

3.4 L1$^+$ Leakage Profile

The L1$^+$ profile mentioned here appears in [6] but is not labeled as a separate profile. We nevertheless reintroduce this profile here for two reasons: first, it might be an interesting performance-security tradeoff, as shown by the work of Tessaro and Cash [9]; second, this allows us to differentiate the IKK attack [15] from the Count attack [6], which are described in Section 4.

With no initial search, SSE schemes of class $\mathsf{L1}^+$ of leakage function $\mathcal{L}$ only reveals the number of documents matching an unknown keyword, i.e.:

$$\mathcal{L}(\mathsf{EDB}) = \left\{\#\mathsf{EDB}(w_i^*)\right\}_{1 \leqslant i \leqslant m}.$$

Example 4. Resuming Example 1, and assuming the client did no query, the following information about d_1 and d_2 is revealed by an $\mathsf{L1}^+$-SSE scheme:

w^*	10	11	14	25	33	35	76	78
$\#\mathsf{EDB}(w^*)$	1	1	1	1	1	2	2	1

The EDB initial state only shows the number of documents sharing a given keyword. Now, if the client searches for tokens 35 and 78, this reveals:

w^*	10	11	14	25	33	35	76	78
$\#\mathsf{EDB}(w^*)$	1	1	1	1	1	2	2	1
Identifiers						id_1 id_2		id_1

Hence, documents identified by id_1 and id_2 share the same keyword of token 35; keywords of tokens 14 and 35 are both in the document identified by id_1.

3.5 L1 Leakage Profile

With no initial search, SSE schemes of class L1 leak the least possible amount of information i.e., the number N of document/keyword pairs of the dataset:

$$\mathcal{L}(\mathsf{EDB}) = \{N\}.$$

Example 5. Resuming Example 1, when no search has been done, the information revealed by an L1-SSE scheme looks like:

w^*	α	β	γ	δ	ϵ	ζ	η	θ	ι	κ
Identifiers	a	b	c	d	e	f	g	h	i	j

Greek (resp. latin) letters represent tokens (resp. identifiers). The server has absolutely no clue about this correspondence, so it only knows $N = 10$. If the client searches for "soon" and "you", this reveals:

w^*	α	β	γ	δ	ϵ	35	35	θ	ι	78
Identifiers	a	b	c	d	e	id_1	id_2	h	i	id_1

The server then learns the same information as for the $\mathsf{L1}^+$ leakage.

3.6 Effect of Queries on the Leakage Profile

We study what can be inferred from the protected database, but it is informative to reflect upon the effect of queries on the amount of information revealed to the server. Namely, at the end of the search protocol the client obtains identifiers of the documents matching its query. Server-side this can be leveraged to associate search tokens $\{w_{i_1}^*, \ldots, w_{i_q}^*\}$ to their matched documents $\{\mathsf{EDB}(w_{i_1}^*), \ldots, \mathsf{EDB}(w_{i_q}^*)\}$, which corresponds precisely to the definition of the L2 leakage profile given above. Actually, if all keywords are queried then the leakage profiles L1 and $\mathsf{L1}^+$ both collapse to L2.

4 Prior Work: Going Beyond L2 Leakage Is Difficult

The efficiency of attacks is often viewed as the percentage of queries that could be correctly paired with plaintext keywords. Actually, efficiency should include the computational cost and the conservativeness of the assumptions. Thus, before presenting our attacks in Section 5 schemes, we discuss the efficiency of the IKK attack from Islam et al. [15] and the Count attack from Cash et al. [6].

4.1 IKK Attack

In [15], Islam et al. present a passive query recovery attack on SSE schemes. It requires access to a co-occurrence matrix C_{W} which represents the probability for two keywords to appear in a randomly chosen document. The attack also requires the observation of queries issued by the client and the responses provided by the server. The adversary is then able to compute for each pair of search tokens, the number of documents which are a match for both. Associating keywords to search tokens boils down to finding the minimum of the function F defined as follows: for observed search tokens $(w_1^*, \dots, w_q^*)$,

$$F(i_1, \dots, i_q) = \sum_{1 \leq s,t \leq q} \left| \frac{\#\big(\mathsf{EDB}(w_s^*) \cap \mathsf{EDB}(w_t^*)\big)}{n} - C_{\mathsf{W}}\big(w_{i_s}, w_{i_t}\big) \right|^2 .$$

Since no assumption are made about the amount of leakage obtained from the SSE scheme, we can classify the IKK attack as an L1 attack with auxiliary information in the form of this co-occurence matrix C_{W}. To the best of our knowledge this is the most generic attack on SSE schemes. Islam et al. justify the access to a co-occurence matrix by implying that it could be computed from a dataset *similar* to the one targeted by the attack. The cost of building C_{W} could then be amortized over several datasets. Unfortunately, practical experiments of [6] show that any kind of success with this attack requires C_{W} to have been computed directly from the plaintext dataset DB. Another constraint is that for the attack to be practical C_{W} cannot be built over the full dictionnary; we must assume that all the search tokens are associated to a keyword represented in C_{W}. Following Section 3.6, we can relax the requirements by considering this attack on an L2 scheme. In this setting the adversary computes the response set intersections directly from EDB without the need for search tokens. Complexity-wise, the IKK attack is costly as minimizing the objective function F requires the use of simulated annealing [15].

4.2 Count Attack

The Count attack from Cash et al. in [6] also aims at passively recovering queries with the help of queries and a C_{W}. However on top of that, it requires to have access for each keyword to the number of plaintext documents that contain it. The adversary is then able to match search tokens to a set of candidate

keywords. Wrong candidates are then eliminated using C_{W}. In Appendix B we give the complete algorithm.

Cash et al. classify Count as an $\mathsf{L1}$ attack but it is worth discussing the impact of the auxiliary information on the actual leakage profile required. Count assumes that the adversary has access to the pairs $\left(w_i, \#\mathsf{DB}(w_i)\right)_{1\leqslant i\leqslant n}$ from which he can compute the set $\left\{\#\mathsf{DB}(w_1), \ldots, \#\mathsf{DB}(w_n)\right\} = \left\{\#\mathsf{EDB}(w^*_{i_1}), \ldots, \#\mathsf{EDB}(w^*_{i_n})\right\}$. One sees that the auxiliary information implies the $\mathsf{L1}^+$ leakage profile given in Section 3.4. Thus we re-classify it as an $\mathsf{L1}^+$ attack with auxiliary information.

In practice the increased amount of auxiliary information compared to the IKK attack allows for a less precise C_{W} : results from [6] show that the IKK attack requires to build C_{W} from the entire plaintext dataset to obtain meaningful results while the Count attack achieves a 40% keyword recovery rate with C_{W} built from 80% of the plaintext dataset.

The use of a co-occurence matrix means that the Count attack shares properties with the IKK attack: namely we assume that the observed search queries correspond to keywords in C_{W} and we do not need search queries anymore if we attack an $\mathsf{L2}$ scheme. Complexity-wise, the Count attack is orders of magnitude faster than the IKK attack since we leverage the extraneous auxiliary information to avoid doing any numerical optimization step.

5 Partial Plaintext Recovery Attacks

Our attacks aim at recovering information on encrypted documents from the sole knowledge of EDB. These attacks are completely passive; the only extra assumption made here is that we know a (small) sample $\mathcal{I}$ of the plaintext documents. We stress that this is in practice a very realistic assumption: for instance, a dataset of books has a very good chance of containing some best-sellers; datasets of mails might as well contain items that have been transferred outside the scope of the SSE scheme.

These attacks proceed in two steps. In the first step, each plaintext of $\mathcal{I}$ is associated to its protected information in EDB. This step is performed using statistical properties that can be computed independently from the plaintexts themselves or from the associated leakage given in EDB. The performance of this association step heavily depends on the statistic capacity to give unique results over the dataset. In the second step, the keywords of the plaintexts are paired with their tokens. Of course, under $\mathsf{L4}$ and $\mathsf{L3}$ leakage profiles, which preserve the order of keywords in EDB, this pairing is completely straightforward.

Finally, correspondences between keywords and tokens obtained from $\mathcal{I}$ can be spread back into EDB, thus recovering partially or totally the content of the encrypted documents. This actually has a devastating effect, giving to the server a massive knowledge of DB it is not supposed to have, as shown in Section 6.

5.1 Mask Attack on $\mathsf{L4}$-SSE Schemes

In order to capture keywords number, order and occurence counts, we introduce the *mask* of a document d_i (resp. id_i), denoted by $\mathsf{mask}(d_i)$ (resp. $\mathsf{mask}(\mathsf{id}_i)$),

as the sequence where all keywords (resp. tokens) are replaced by their position of first appearance. For example, if $d_i = (\text{to}, \text{be}, \text{or}, \text{not}, \text{to}, \text{be})$, then $\mathsf{mask}(d_i) = \mathsf{mask}(\mathsf{id}_i) = (1, 2, 3, 4, 1, 2)$.

The idea of the attack is intuitive: for each plaintext $d \in \mathcal{I}$, the mask of d is computed; this mask is then compared with all masks of corresponding length computed from EDB.

Hopefully, only one mask of EDB is matching the mask of d, leading to a correct association. In practice, this is almost always the case (see Section 6). The entire process is summarized in Algorithm 1.

Input: $\mathsf{EDB}, \mathcal{I} \subseteq \mathsf{DB}$
Output: Set of tokens $\mathsf{W}^*_{\mathsf{rec}} \subseteq \mathsf{W}^*$ associated to their keyword in W

foreach $d \in \mathcal{I}$ **do**
  $A_d = \{\, i \mid \ell_i = \#d \text{ and } \mathsf{mask}(\mathsf{id}_i) = \mathsf{mask}(d) \,\}$;
return $\mathsf{W}^*_{\mathsf{rec}} = \{\, \mathsf{W}^*_i \mid \#A_{d_i} = 1 \,\}$

Algorithm 1: Mask Attack.

5.2 Co-Mask Attack on L3-SSE Schemes

Under $\mathsf{L3}$ leakage the Mask Attack does not apply anymore as the mask of a document d boils down to the sequence $\big(1, \ldots, \#[d]\big)$.

Therefore we introduce the *co-resulting mask* of a pair (d_1, d_2) of documents, denoted by $\mathsf{comask}(d_1, d_2)$. Intuitively, it can be viewed as the mask of the intersection, marking the positions of the keywords of $\mathsf{W}_{1,2} = \mathsf{W}_1 \cap \mathsf{W}_2$. More formally, if $\mathbb{1}_{\mathsf{W}_{1,2}}(\cdot)$ is the indicator function on $\mathsf{W}_{1,2}$, we define:

$$\mathsf{comask}(d_1, d_2) = \Big(\big(\mathbb{1}_{\mathsf{W}_{1,2}}(d_1[i])\big)_{1 \le i \le \#\mathsf{W}_1},\ \big(\mathbb{1}_{\mathsf{W}_{1,2}}(d_2[i])\big)_{1 \le i \le \#\mathsf{W}_2} \Big)\ .$$

We stress that this quantity can be computed directly from every EDB of profile $\mathsf{L3}$; by abuse of notation this is denoted by $\mathsf{comask}(\mathsf{id}_1, \mathsf{id}_2)$.

The general idea of the algorithm is as follows: for each pair in $(d_i, d_j) \in \mathcal{I}^2$, the co-resulting mask of the pair is computed and compared with all co-resulting masks computed from elements of EDB which have length $\#[d_i]$ and $\#[d_j]$.

In practice, this kind of exhaustive search would be particularly inefficient. We instead iteratively construct a set A_t containing all t-tuples of identifiers such that the co-resulting masks of all pairs in the t-uple match the co-resulting masks of the corresponding pairs in $(d_1, \ldots, d_t) \subseteq \mathcal{I}$. More formally:

$$A_t = \Big\{ \big(\mathsf{id}_{i_1}, \ldots, \mathsf{id}_{i_t}\big) \ \Big|\ \forall s, u \le t,\ \mathsf{comask}(\mathsf{id}_{i_s}, \mathsf{id}_{i_u}) = \mathsf{comask}(d_s, d_u) \Big\}\ .$$

To compute A_t from A_{t-1} using d_t, we consider for each induced new pair (d_j, d_t) the set $C_{j,t}$ of pairs of identifiers $(\mathsf{id}_{i_j}, \mathsf{id}_{i_t})$ with matching co-resulting masks, such that both id_{i_j} and id_{i_t} are still marked as compatible. From the $C_{j,t}$'s, it is easy to remove all unconsistent t-tuples from A_t i.e., for each j, those having positions j and t not in $C_{j,t}$. When t reaches $\#\mathcal{I}$, the whole search space has

been explored: each component $A_{\#\mathcal{I}}[k]$ composed of only one element gives the correct association $A_{\#\mathcal{I}}[k] = \mathsf{id}_k$.

It is worth noting that in practice A_2 is almost always reduced to one element, and so is $A_{\#\mathcal{I}}$. In any case, very few identifiers would remain possible for a given document in $A_{\#\mathcal{I}}$. The Co-Mask Attack is summarized in Algorithm 2.

```
Input: EDB, I = (d_1, ..., d_#I) ⊆ DB
Output: Set of tokens W*_rec ⊆ W* associated to their keyword in W
/* Consider the first pair of documents                                  */
A_2 = { (id_i1, id_i2) | #id_i1 = #[d_1], #id_i2 = #[d_2]
                         and comask(id_i1, id_i2) = comask(d_1, d_2) };
/* Construct A_t from A_{t-1} using d_t                                  */
for t = 3 to #I do
    A_t = A_{t-1} × { id | #id = #[d_t] };
    /* A_t will be reduced by considering all new pairs (d_j, d_t)       */
    foreach j < t do
        C_{j,t} = { (id_ij, id_it) | id_ij ∈ A_t[j], id_t ∈ A_t[t]
                                     and comask(id_ij, id_it) = comask(d_j, d_t) };
        A_t = { a ∈ A_t | (a[j], a[t]) ∈ C_{j,t} }; /* Keep consistent t-tuples */
        if #A_t = 1 then break;
return W*_rec = { W*_t | #A_#I[t] = 1 }
```

Algorithm 2: Co-Mask Attack

We stress that this attack could be extended to higher order intersections, in the spirit of the PowerSet Attack presented in the next section. In practice, only considering pairs already gives outstanding results, as shown in Section 6.

5.3 PowerSet Attack on L2-SSE Schemes

As the order of keywords is not preserved anymore under L2 leakage, the co-resulting mask used in the Co-Mask Attack cannot be computed. Worse, even if a document is correctly associated to its identifier, inferring the correct association between each keyword and its token is still a challenge. The PowerSet Attack addresses both issues.

Associating Documents and Identifiers. An L2 leakage still allows to determine which keywords are shared between two documents. To associate documents of $\mathcal{I}$ to their identifiers, it is therefore tempting to run the Co-Mask Attack where the co-resulting mask of a pair of documents is replaced by the cardinal of their intersection. Unfortunately this is not sufficient, since in practice many pairs of identifiers of EDB share the same number of tokens.

We introduce the *power set of order* h of a list of t documents, denoted by $\mathsf{PowerSet}_h^t(d_1, \ldots, d_t)$, and defined as the sequence of the $\binom{t}{h}$ cardinals of all

possible intersections of h elements of the t-uple. More formally:

$$\mathsf{PowerSet}_h^t\big(d_1,\dots,d_t\big) = \Big(\# \bigcap_{1\le j\le h} d_{i_j}\Big)_{1\le i_1<\dots<i_h\le t} .$$

The superscript will be omitted when it is clear from the context. We stress that this sequence can be computed directly from every EDB of profile L2; by abuse of notation this will be denoted by $\mathsf{PowerSet}_h\big(\mathsf{id}_1,\dots,\mathsf{id}_t\big)$.

The algorithm strives to exploit all available information on $\mathcal{I}$ i.e., find sequences of identifiers such that all intersections of all possible subsets match the cardinals of those computed on $\mathcal{I}$. As this is a huge search space, it must be explored with care. Therefore, we iteratively construct a set A_t containing all t-tuples of identifiers such that all power sets of order less than t match the power sets of the corresponding documents in $(d_1,\dots,d_t)\in\mathcal{I}$. When t reaches $\#\mathcal{I}$, all information on $\mathcal{I}$ has been processed and singleton components of $A_{\#\mathcal{I}}$ give a correct association.

Computing A_t starting from A_{t-1} and candidate identifiers for d_t requires to reduce the size of A_t as fast as possible. This is done by considering subset intersections of increasing order, thus squeezing A_t as the combinatorics grow. Let $A_t^{(h)}$ be the set of compatible t-tuples with all power sets of order up to h:

$$A_t^{(h)} = \left\{ \big(\mathsf{id}_{i_1},\dots,\mathsf{id}_{i_t}\big) \;\middle|\; \forall s\le h, \quad \begin{array}{c}\mathsf{PowerSet}_s(d_1,\dots,d_t)\\ = \mathsf{PowerSet}_s(\mathsf{id}_{i_1},\dots,\mathsf{id}_{i_t})\end{array} \right\} .$$

The algorithm then computes the following decreasing sequence, using the procedure Reduce given in Algorithm 3 to go from $A_t^{(h)}$ to $A_t^{(h+1)}$:

$$A_{t-1}\times\big\{\ \mathsf{id}\ \big|\ \#\mathsf{id}=\#\{d_t\}\ \big\} = A_t^{(1)} \supseteq A_t^{(2)} \supseteq A_t^{(3)} \supseteq\dots\supseteq A_t^{(t)} = A_t .$$

```
Input: I_t = (d_1, ..., d_t), h-order candidates A_t^(h)
Output: Set of (h+1)-order candidates A_t^(h+1)
B_t = A_t^(h);
/* Consider each subset of (h+1) elements containing d_t            */
foreach 1 ≤ j_1 < ··· < j_h < t do
    C_{j,t} = { ((id_{i_j}), id_{i_t}) | id_t ∈ B_t[t], (id_{i_j}) ∈ B_t[j],
                                         and #(id_{i_t} ∩ (id_{i_j})) = #(d_t ∩ (d_j)) };
    B_t = { b ∈ B_t | ((b[j]), b[t]) ∈ C_{j,t} };     /* Keep consistent t-tuples */
    if #B_t = 1 then break;
return A_t^(h+1) = B_t
```

Algorithm 3: Reduce procedure: computing $A_t^{(h+1)}$ from $A_t^{(h)}$.

We stress that, by induction, only subsets containing d_t have to be considered. Algorithm 4 summarizes the first phase of the PowerSet Attack.[3]

[3] In our implementation of the PowerSet Attack, we first go through all documents then we increase the size of subsets to avoid phenomena of combinatorial explosion.

Input: EDB, $\mathcal{I} = (d_1, \dots, d_{\#\mathcal{I}}) \subseteq \mathsf{DB}$
Output: Set of documents $\mathcal{I}_0 \subseteq \mathcal{I}$ associated to their identifiers in EDB

```
/* Consider the first pair of documents */
```

$$A_2 = \left\{ (\mathsf{id}_{i_1}, \mathsf{id}_{i_2}) \;\middle|\; \begin{array}{l} \#\mathsf{id}_{i_1} = \#\{d_1\},\ \#\mathsf{id}_{i_2} = \#\{d_2\} \\ \text{and } \mathsf{PowerSet}_2(\mathsf{id}_{i_1}, \mathsf{id}_{i_2}) = \mathsf{PowerSet}_2(d_1, d_2) \end{array} \right\};$$

/* Construct A_t from A_{t-1} using d_t */

for $t = 3$ **to** $\#\mathcal{I}$ **do**

- $A_t^{(1)} = A_{t-1} \times \{\, \mathsf{id} \mid \#\mathsf{id} = \#\{d_t\} \,\}$;
- /* Consider intersections of increasing order h to reduce A_t */
- **for** $h = 2$ **to** t **do**
  - $A_t^{(h)} = \mathsf{Reduce}(A_t^{(h-1)})$;
  - **if** $\#A_t^{(h)} = 1$ **then** set $A_t = A_t^{(h)}$ and break;

return $\mathcal{I}_0 = \{\, d_t \mid \#A_{\#\mathcal{I}}[t] = 1 \,\}$

Algorithm 4: PowerSet Attack: documents-identifiers association.

In practice, computing A_2 is the most costly part of Algorithm 4, as the result is sufficiently small so that adding new documents becomes negligible. Moreover, experiments show that A_t is reduced to one element as soon as $t \geq 4$.

Associating Keywords and Tokens. The previous phase associates each document of $\mathcal{I}_0$ with a set of tokens. Since token ordering is not preserved under L2 leakage, finding the correct keyword-token associations remains non-trivial.

To solve this problem, we construct the *inverted index* of $\mathcal{I}_0$, denoted by $\mathsf{inv}(\mathcal{I}_0)$, which associates the keywords $w \in \mathcal{I}_0$ and to the identifiers of the documents containing w. This inverted index is then ordered by decreasing number of identifiers to form the *ordered inverted index* $\mathsf{inv}_{\geq}(\mathcal{I}_0)$.

Consider first the keyword w_i having the most identifiers, and assume that no following keyword has the same associated identifiers. Hence the intersection of the sets of tokens associated to w_i gives us an unique match w_i^*. Now, if the second line w_j of $\mathsf{inv}_{\geq}(\mathcal{I}_0)$ is also unique, we distinguish two cases: either the intersection of the sets of tokens associated to w_j gives us an unique match w_j^*; or, when identifiers are also associated to the previous keyword w_i, we obtain two tokens. Knowing w_i^* from the first association, we easily deduce the token w_j^* associated to w_j.

Example 6. Let $\mathcal{I}_0 = \{d_1, d_2, d_3\}$ be a set of three documents $d_1 = (w_1, w_2, w_3)$, $d_2 = (w_3, w_2)$ and $d_3 = (w_1, w_2)$. Inverted indexes $\mathsf{inv}(\mathcal{I}_0)$ and $\mathsf{inv}_{\geq}(\mathcal{I}_0)$ are:

$\mathsf{inv}(\mathcal{I}_0)$	
w_1	id_1 id_3
w_2	id_1 id_2 id_3
w_3	id_1 id_2

$\mathsf{inv}_{\geq}(\mathcal{I}_0)$	
w_2	id_1 id_2 id_3
w_3	id_1 id_2
w_1	id_1 id_3

Consider the first line of $\mathsf{inv}_{\geq}(\mathcal{I}_0)$. We know that only w_2 is in d_1, d_2 and d_3. Hence $\mathsf{W}_1^* \cap \mathsf{W}_2^* \cap \mathsf{W}_3^* = \{w_2^*\}$. Now, consider the second keyword of $\mathsf{inv}_{\geq}(\mathcal{I}_0)$ i.e., w_3. This keyword is in d_1 and d_2, but w_2 too. So $\mathsf{W}_1^* \cap \mathsf{W}_2^* = \{w_2^*, w_3^*\}$, but we already know that w_2^* is the token of w_2, hence the token of w_3 is w_3^*.

Unfortunately, several keywords may be associated to the same identifiers i.e., be simultaneously in exactly the same documents. In this case, they are completely undistinguishable and we ignore them when they appear in the following intersections. This keyword/token association process is given in Algorithm 5.

Input: EDB, set $\mathcal{I}_0 \subseteq \mathcal{I}$ of documents associated to their identifiers
Output: Set of tokens $\mathsf{W}^*_{\mathsf{rec}} \subseteq \mathsf{W}^*$ associated to their keyword in W

$\mathsf{W}^*_{\mathsf{ign}} \leftarrow \emptyset$; `/* Contains associated and undistinguishable tokens */`
Compute $\mathsf{inv}_{\geq}(\mathcal{I}_0)$;
foreach $w \in \mathsf{inv}_{\geq}(\mathcal{I}_0)$ *taken in decreasing order* **do**
  $A_w = \left(\bigcap \left\{ \mathsf{W}^*_i \;\middle|\; \mathsf{id}_i \in \mathsf{inv}_{\geq}(\mathcal{I}_0)[w] \right\}\right) \setminus \mathsf{W}^*_{\mathsf{ign}}$;
  $\mathsf{W}^*_{\mathsf{ign}} = \mathsf{W}^*_{\mathsf{ign}} \cup A_w$; `/* Associated (`$\#A_w = 1$`) or undistinguishable */`
return $\mathsf{W}^*_{\mathsf{rec}} = \left\{ A_w \;\middle|\; \#A_w = 1 \right\}$

Algorithm 5: PowerSet Attack: keywords-tokens association.

5.4 Elements of Complexity

Deriving precise complexity bounds for our attacks is difficult, as they heavily depend on statistical properties of the targeted dataset. We nevertheless give some elements allowing to compare the impact of the leakage profiles.

The most relevant data for our attacks is the maximum number of identifiers to consider for a document of a given length i.e., for each leakage profile:

$$\mathsf{M}_{\mathsf{L23}} = \max_{d \in \mathsf{DB}} \#\left\{\mathsf{id} \;\middle|\; \#\mathsf{id} = \#[d]\right\} \quad \text{and} \quad \mathsf{M}_{\mathsf{L4}} = \max_{d \in \mathsf{DB}} \#\left\{\mathsf{id} \;\middle|\; \#\mathsf{id} = \#d\right\}$$

Measurements on our datasets (see Section 6) show that $\sqrt{\#\mathsf{DB}}$ is a good approximation of these values. It is actually a worst-case assumption, as in practice we can consider documents of $\mathcal{I}$ with the least number of candidates first.

Mask Attack. Each document of $\mathcal{I}$ induces the computation of at most M_{L4} masks; hence the total complexity is $\mathcal{O}\big(\#\mathcal{I} \cdot \mathsf{M}_{\mathsf{L4}}\big)$ mask computations.

Co-Mask Attack. Constructing A_2 costs $\mathsf{M}^2_{\mathsf{L23}}$ applications of comask. We heuristically expect the sets A_t to decrease as t grows. Indeed, if $\#A_2 \leq \mathsf{M}_{\mathsf{L23}}$, each association of d_t starts from a smaller set A_{t-1} and imposes greater constraints, thus costing at most $\#A_2 \cdot \mathsf{M}_{\mathsf{L23}}$. In our experiments, A_2 is almost always reduced to one element, so we conjecture a total complexity of $\mathcal{O}\big(\mathsf{M}^2_{\mathsf{L23}} + (1+\epsilon)\#\mathcal{I} \cdot \mathsf{M}_{\mathsf{L23}}\big)$ co-resulting mask computations.

PowerSet Attack. The analysis is much more complex. However, heuristically, the first pair considered drastically reduces the number $\#A_2$ of candidates, and the same reasoning as above leads to a conjectured complexity of $\mathcal{O}\big(\mathsf{M}^2_{\mathsf{L23}} + \#A_2 \cdot \#\mathcal{I} \cdot \mathsf{M}_{\mathsf{L23}}\big)$ intersections cardinals computations.

6 Experimental Results

Real-World Datasets. We implemented and ran the attacks presented in Section 5 on four different real-world datasets to evaluate their practical efficiency.

The first dataset is the email dataset from the *Enron* corporation, available online [11]. Unlike Islam et al. [15] and Cash et al. [6], we consider all 490,369 emails of the datasets, including mails sent from the outside of *Enron*. The second and third datasets are mailing lists from the *Apache* foundation, namely *Apache Commons* [4] (28,997 emails) and *Apache Lucene* [5] (58,884 emails). The last dataset is the 21,602 books of the *Project Gutenberg* [14].

One email message, one article or one book is considered as one document. For each document, stopwords have been removed and keywords processed using the standard Porter stemming algorithm [17].

Efficiency Measures. We ran our attacks for different sizes of $\mathcal{I}$ using steps of 1% until 10% then steps of 10% from 10% to 100%. Here 1% represents 1% of the pairs (d, w) of the dataset; this allows for a fairer comparison between datasets than the usual per-document measure, as knowing a long document do not have the same impact as knowing a short one.

The measured *success rate* is the ratio of keywords-tokens associations over the set of keywords of $\mathcal{I}$. Then, these correspondences are spread back into EDB in order to evaluate their impact on other documents of the dataset. In particular, we measured the rate of documents of the dataset whose keywords are recovered at 70%, 80%, 90% and 100%.

Experimental Results on Lucene. We expose here the results of our attacks on the *Lucene* dataset; the reader is invited to refer to Appendix C for details on other datasets. All timings are measured on a Core i7 using 16 Go RAM.

As shown in Figure 1, our attacks have a huge impact, giving the server a precise knowledge of the entire dataset even when $\mathcal{I}$ is small. Table 1 summarizes the results when the server knows only 1% of the dataset.

Mask Attack. Over 98% of documents have a unique mask in *Lucene* dataset. This translates into over 99% keyword-token association rate over the set $\mathcal{I}$ in all cases. Moreover, knowing only 1% of the dataset already allows the server to recover 70% of the keywords close to all documents; and 3146 of them are completely recovered. Running the Mask Attack on the *Lucene* dataset when 1% of the dataset is known to the server requires 72 seconds.

Co-Mask Attack. Experiments show that despite the loss of the frequency information, it remains as effective as the Mask Attack. However, the computational complexity increases and running this attack when 1% of the dataset is known to the server requires 284 seconds.

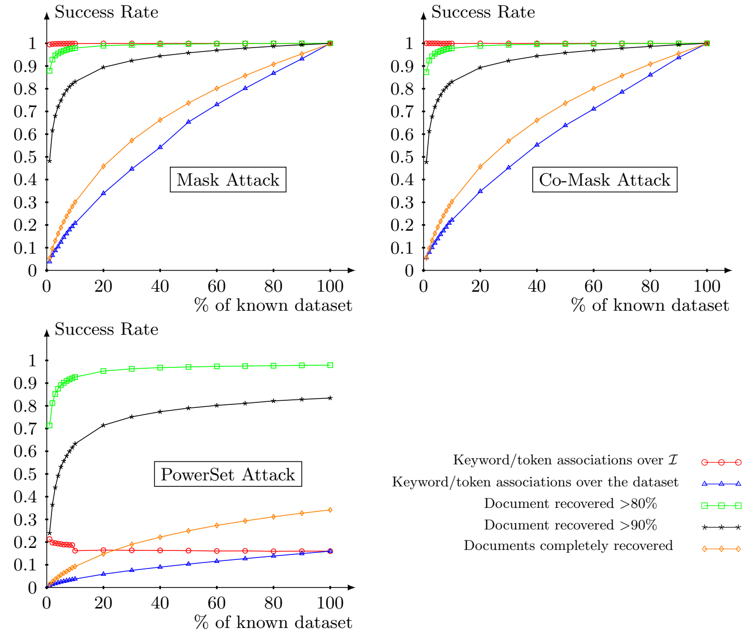


Fig. 1: Efficiency of our attacks on *Lucene* dataset depending on the server's knowledge rate.

PowerSet Attack. It suffers widely from the loss of keyword order. Hence, while the documents-identifiers association performs equally well, the exact association between keywords and tokens plateau around 20%. Still, the knowledge of 1% of the dataset already allows to recover 80% of the keywords of more than 70% of the database documents. This attack is the most costly and runs in 489 seconds when 1% of the dataset is known to the server.

	Success Rate	70%	80%	90%	100%	Time (s)
Mask Attack	.99	.97	.88	.48	.05	72
Co-Mask Attack	.99	.97	.87	.47	.05	284
PowerSet Attack	.21	.93	.71	.24	.01	489

Table 1: Attacks efficiency on Lucene datasets with a 1% plaintext knowledge.

Practical Impact. As noted in [6], this reconstruction allows to reveal sensitive information even if the order of keywords is not preserved. Human inspection of the output of our attacks gives a clear idea of the sense of each document.

7 Conclusion

Prior work [19] taught us that SSE schemes have no hope of being secure in a setting where the adversary can inject chosen files. Additionally, the authors of [6, 15] have shown that passive observation of search tokens reveal the underlying searched keyword when the dataset is completely known.

This paper, focusing on passive attacks of L4, L3 and L2 schemes, adds to this knowledge in several ways. The most glaring conclusion is that our attacks are devastating: regardless of the leakage profile, knowing a mere 1% of the document sets translates into over 90% of documents whose content is revealed over 70%. Moreover, we show that from a security point of view the L4 and L3 leakage profiles are actually equivalent. To conclude, the results in this paper help us to better understand the practical security of SSE schemes and hopefully will help practioners make more secure SSE schemes.

Future work may look into counter-measures to these attacks and their impact on real-world datasets. Another valuable area of research is the study of the behaviour of the degradation from L1 and L1$^+$ to L2 in the presence of queries.

References

1. Bitglass. Security, Compliance, and Encryption. `http://www.bitglass.com/salesforce-security`. Accessed: 2017-01-18.
2. CipherCloud. Cloud Data Encryption. `https://www.ciphercloud.com/technologies/encryption/`. Accessed: 2017-01-18.
3. Skyhigh Networds. Skyhigh for Salesforce. `https://www.skyhighnetworks.com/product/salesforce-security/`. Accessed: 2017-01-18.
4. Apache Commons email dataset. `http://mail-archives.apache.org/mod_mbox/commons-user/`. Accessed: 2016-04.
5. Apache Lucene email dataset. `http://mail-archives.apache.org/mod_mbox/lucene-java-user/`. Accessed: 2016-04.
6. D. Cash, P. Grubbs, J. Perry, and T. Ristenpart. Leakage-Abuse Attacks Against Searchable Encryption. In *Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security*, CCS '15, pages 668–679, New York, NY, USA, 2015. ACM.
7. D. Cash, J. Jaeger, S. Jarecki, C. S. Jutla, H. Krawczyk, M. Rosu, and M. Steiner. Dynamic Searchable Encryption in Very-Large Databases: Data Structures and Implementation. In *21st Annual Network and Distributed System Security Symposium, NDSS 2014, San Diego, California, USA, February 23-26, 2014*, 2014.
8. D. Cash, S. Jarecki, C. Jutla, H. Krawczyk, M.-C. Roşu, and M. Steiner. Highly-Scalable Searchable Symmetric Encryption with Support for Boolean Queries. In *Advances in Cryptology – CRYPTO 2013: 33rd Annual Cryptology Conference, Santa Barbara, CA, USA, August 18-22, 2013. Proceedings, Part I*, pages 353–373, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.
9. D. Cash and S. Tessaro. The Locality of Searchable Symmetric Encryption. In *Advances in Cryptology – EUROCRYPT 2014*, volume 8441 of *Lecture Notes in Computer Science*, pages 351–368. Springer Berlin Heidelberg, 2014.

10. 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 '06, pages 79–88, New York, NY, USA, 2006. ACM.
11. Enron email dataset. `http://www.cs.cmu.edu/~./enron/`. Accessed: 2016-04.
12. S. Faber, S. Jarecki, H. Krawczyk, Q. Nguyen, M. Rosu, and M. Steiner. Rich Queries on Encrypted Data: Beyond Exact Matches. In *Computer Security – ESORICS 2015: 20th European Symposium on Research in Computer Security, Vienna, Austria, September 21-25, 2015, Proceedings, Part II*, pages 123–145. Springer International Publishing, 2015.
13. O. Goldreich. *Secure Multi-party Computation*, 1998. Working Draft.
14. Project Gutenberg. `http://www.gutenberg.org/wiki/Main_Page`. Accessed: 2016-04.
15. M. S. Islam, M. Kuzu, and M. Kantarcioglu. 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.
16. S. Kamara, C. Papamanthou, and T. Roeder. Dynamic Searchable Symmetric Encryption. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, CCS '12, pages 965–976, New York, NY, USA, 2012. ACM.
17. M. F. Porter. An algorithm for suffix striping. *Program*, 14(3):130–137, 1980.
18. D. X. Song, D. Wagner, and A. Perrig. Practical Techniques for Searches on Encrypted Data. In *Proceedings of the 2000 IEEE Symposium on Security and Privacy*, SP '00, pages 44–, Washington, DC, USA, 2000. IEEE Computer Society.
19. Y. Zhang, J. Katz, and C. Papamanthou. All Your Queries Are Belong to Us: The Power of File-Injection Attacks on Searchable Encryption. Cryptology ePrint Archive, Report 2016/172, 2016. `http://eprint.iacr.org/2016/172`.

A Static SSE Scheme

Definition 1 (Static SSE scheme). *Given a symmetric encryption scheme* $(\mathcal{E}.(\cdot), \mathcal{D}.(\cdot))$ *we define a* static SSE scheme *of security parameter* λ *as a quartet of polynomial-time algorithms* $\Pi = (\mathsf{Gen}, \mathsf{Setup}, \mathsf{SearchClient}, \mathsf{SearchServer})$ *by:*

$(K, k) \leftarrow \mathsf{Gen}(1^\lambda)$ *is a probabilistic key generation algorithm that is run by the client. It takes as input a security parameter* λ*, and outputs two symmetric secret keys* K *and* k *which are both kept securely by the client.*

$(\mathsf{EDB}, \mathsf{DB}^*) \leftarrow \mathsf{Setup}(K, k, \mathsf{DB}, \mathcal{E}.(\cdot))$ *is an algorithm that is run by the client to set the scheme up. It takes as input secret keys* K *and* k*, the database* DB *and the algorithm* $\mathcal{E}.(\cdot)$*, and outputs both the protected metadata* EDB *and the encrypted documents* $\mathsf{DB}^* = (\mathcal{E}_k(d_1), \ldots, \mathcal{E}_k(d_n))$*.*

$w^* \leftarrow \mathsf{SearchClient}(K, w)$ *is a deterministic algorithm that is run by the client to send a query to the server. It takes as input the secret key* K *and a keyword queried* $w \in \mathsf{W}$*, and outputs the search token* $w^* \in \mathsf{W}^*$ *associated with* w*. Finally* w^* *is sent to the server.*

$\mathsf{EDB}(w^*) \leftarrow \mathsf{SearchServer}(\mathsf{EDB}, w^*)$ *is a deterministic algorithm that is run by the server to answer a client-query. It takes as input the protected metadata* EDB *and the client-generated search token* w^* *and outputs* $\mathsf{EDB}(w^*)$*: the identifiers of the encrypted documents containing keyword* w*. This list is sent back to the client.*

B Count Attack Algorithm

```
Input: W, T, C_T, C_W
Output: K ⊂ W × T
K ← ∅ ;
Trec ← ∅ ;
WrongPickFlag ← false ;
ProgressFlag ← true ;
min ← 0 ;
foreach t ∈ T do
    L(t) ← {w ∈ W s. t. #DB(w) = #EDB(t)} ;
    if #L(t) < min then
        min ← |L(t)| ;
        t_min ← t ;
foreach w ∈ L(t_min) do
    K ← (t_min, w) ;
    Trec ← t_min ;
    WrongPickFlag ← false ;
    while ProgressFlag do
        if WrongPickFlag = true then
            continue ;
        ProgressFlag ← false ;
        foreach t ∈ T − T_rec do
            foreach w ∈ L(t) do
                foreach (t', w') ∈ K do
                    if C_T(t, t') ≠ C_W(w, w') then
                        L(t) ← L(t) − {w} ;
            if L(t) = {w} then
                K ← K ∪ {(t, w)} ;
                Trec ← Trec ∪ {t} ;
                ProgressFlag ← true;
            if L(t) = ∅ then
                WrongPickFlag ← true ;
                continue ;
    if WrongPickFlag = false then
        return K
```

Algorithm 6: Count attack [6]

C Experimental Results

In Section C.1, we present the results for Mask Attack with datasets *Commons* in Figure 2, *Lucene* in Figure 3, *Enron* in Figure 4 and *Gutenberg* in Figure 5. Section C.2, presents the results for Co-Mask Attack with datasets *Commons* in Figure 6, *Lucene* in Figure 7 and *Gutenberg* in Figure 8. To finish, Section C.3 presents the results for PowerSet Attack with datasets *Commons* in Figure 9 and *Lucene* in Figure 10.

Graphs: Two informations are presented in each graph. The first information is the rate of associations between keywords and search tokens. This rate is calculated compared to the number of keywords present in the sample $\mathcal{I}$, then it is calculated compared to the total number of keywords (#W) in the dataset.

The second information is the number of documents recovered partially or totally. We give the number of documents recovered at least 80%, 90% and 100%.

Tables: The first column of each table is the percentage of plaintext documents known by the server. The second column represent the rate of associations between keywords and search tokens. Other columns present the number of documents recovered at least 70%, 80%, 90% and 100%.

Datasets: We present below some characteristics of datasets that we used. These datasets are all available online and their different nature allow us to verify the efficiency of our attacks on differents contents.

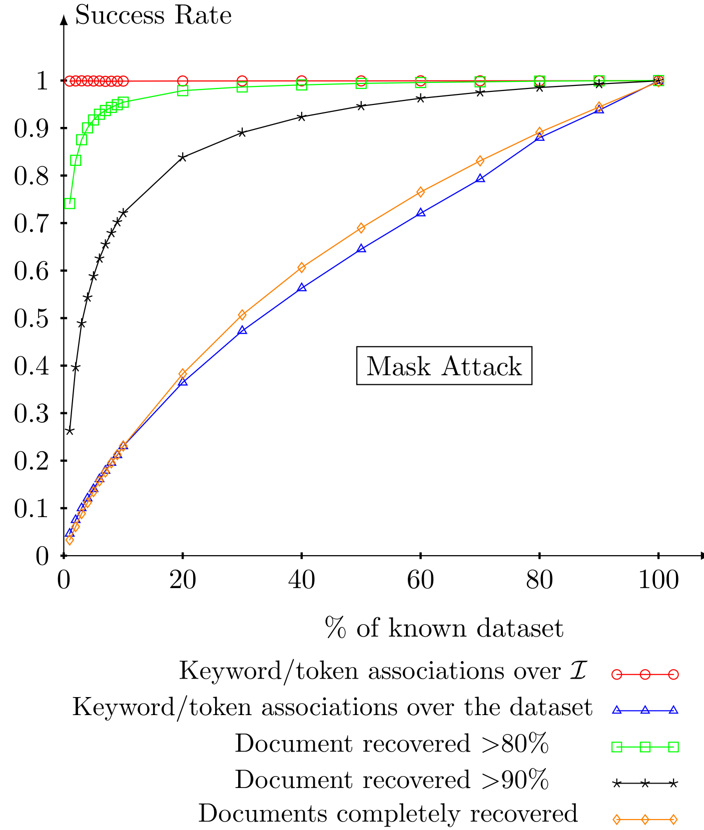
Datasets	Content	#DB	#W	N
Commons	mailing list	28 997	230 893	3 910 562
Enron	emails	490 369	643 818	47 301 160
Gutenberg	books	21 602	2 853 955	91 261 811
Lucene	mailing list	58 884	394 481	7 952 794

Table 2: Characteristics of used datasets.

Discussion: Below, we summarize the efficiency between our three attacks.

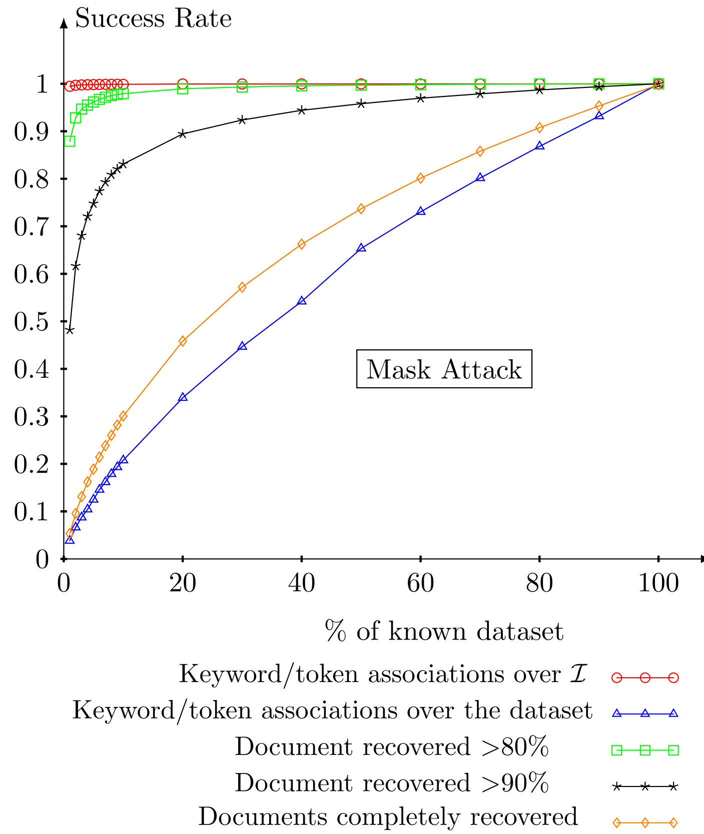
- L4 Due to the amount of leaked information, our attack on L4-SSE schemes are the most efficient. On average, we exploit at 99% information given by the plaintext documents known by the adversary.
- L3 In spite of the lost of information on the frequency of keywords in documents, our attack on L3-SSE schemes are also efficient as the attack on L4-SSE schemes. In fact, we still exploit at 99% information given by the plaintext documents known by the adversary.
- L2 When we do not know the order of keyword in documents, we can now associate on average 20% of them to their corresponding token. However, these 20% of keywords are present in most documents, that is why we can recover a large part of the documents.

C.1 Mask Attack Results for L4 Leakage



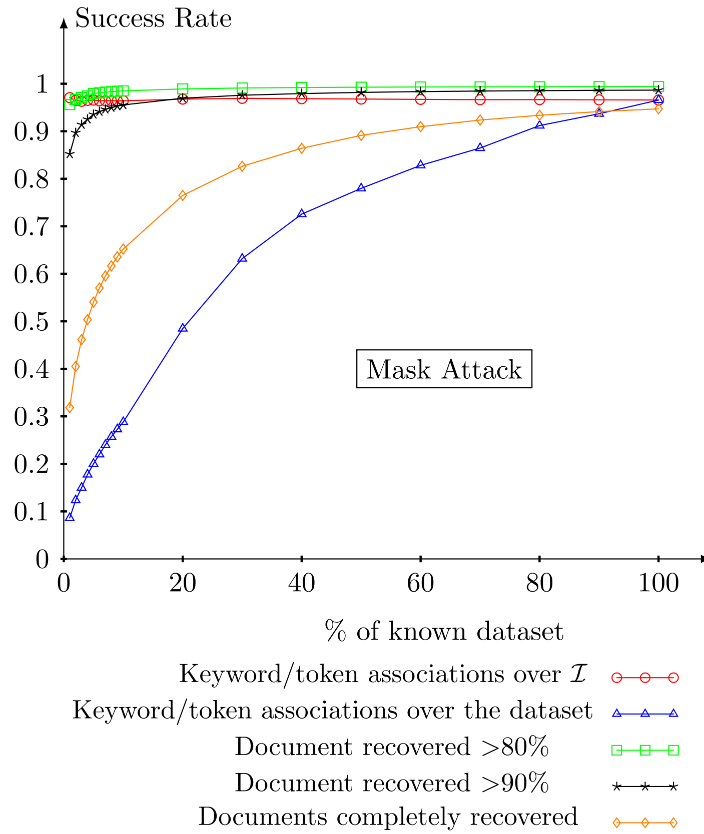
%	Rate	# 70%	# 80%	# 90%	# 100%
1	0.99	27 136	21 488	7 631	961
2	0.99	27 931	24 125	11 503	1 775
3	0.99	28 224	25 381	14 182	2 571
4	0.99	28 403	26 105	15 758	3 242
5	0.99	28 489	26 587	17 053	3 911
10	0.99	28 721	27 672	20 908	6 692
20	0.99	28 862	28 376	24 308	11 095
30	0.99	28 905	28 601	25 819	14 693
40	0.99	28 935	28 724	26 777	17 583
50	0.99	28 955	28 817	27 435	19 994

Fig. 2: Results of the Mask Attack on *Commons* dataset.



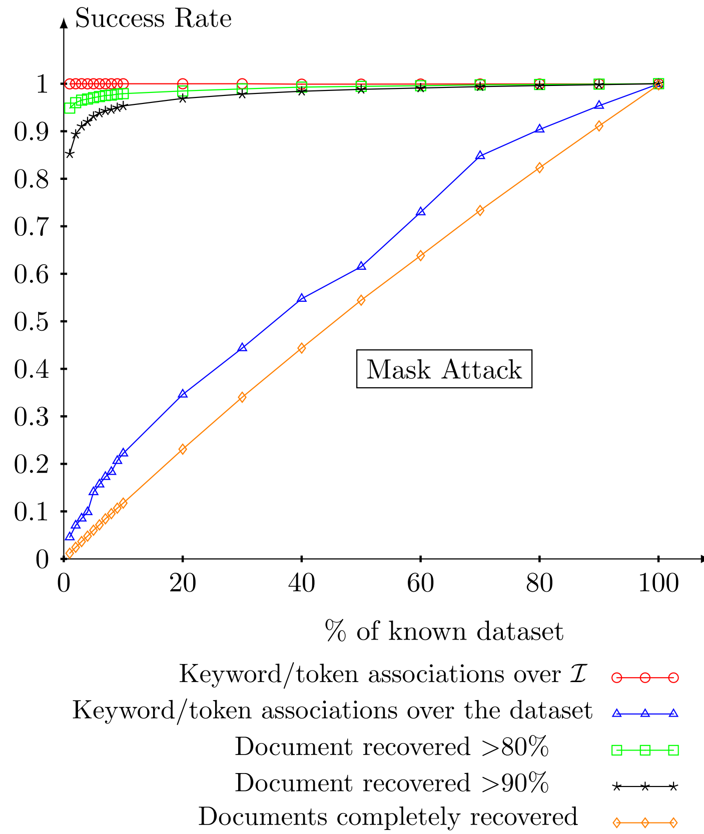
%	Rate	# 70%	# 80%	# 90%	# 100%
1	0.99	57 335	51 752	28 374	3 146
2	0.99	57 928	54 665	36 297	5 639
3	0.99	58 175	55 741	40 068	7 724
4	0.99	58 273	56 246	42 443	9 538
5	0.99	58 348	56 630	44 031	11 120
10	0.99	58 582	57 653	48 922	17 704
20	0.99	58 706	58 261	52 660	26 989
30	0.99	58 764	58 481	54 384	33 670
40	0.99	58 798	58 630	55 586	39 001
50	0.99	58 823	58 704	56 404	43 393

Fig. 3: Results of the Mask Attack on *Lucene* dataset.



%	Rate	# 70%	# 80%	# 90%	# 100%
1	0.97	479 064	468 840	417 832	156 084
2	0.96	481 403	474 111	439 764	198 529
3	0.96	482 285	476 123	448 110	226 283
4	0.96	483 145	477 791	453 622	246 929
5	0.96	484 947	480 232	458 607	264 975
10	0.96	486 237	482 937	468 423	319 751
20	0.96	487 092	484 983	475 371	375 053
30	0.96	487 535	485 904	478 460	405 158
40	0.96	487 744	486 387	480 150	423 732
50	0.96	487 939	486 730	481 363	437 024

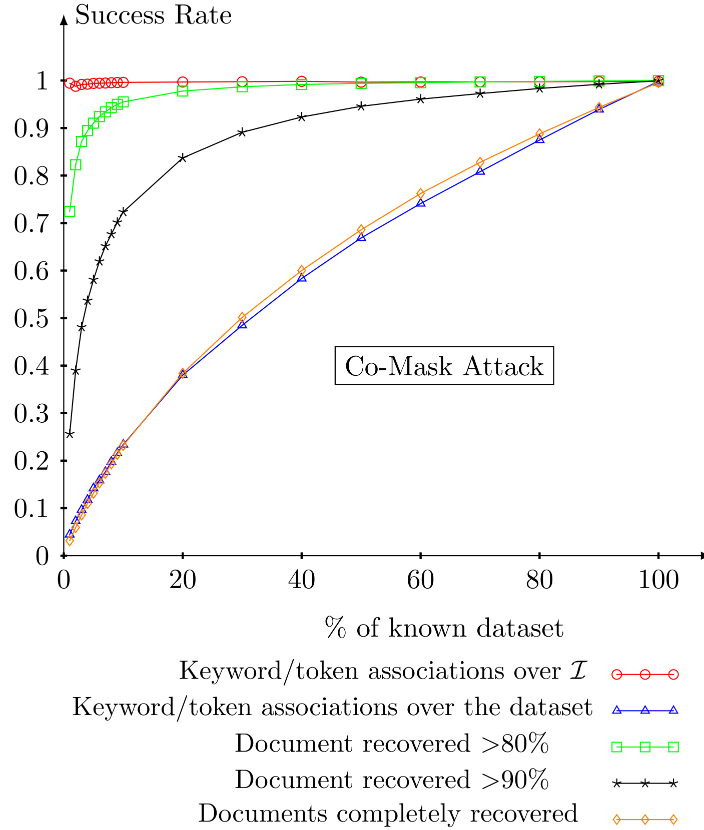
Fig. 4: Results of the Mask Attack on *Enron* dataset.



%	Rate	# 70%	# 80%	# 90%	# 100%
1	0.99	20 966	20 491	18 415	257
2	0.99	21 103	20 738	19 303	527
3	0.99	21 175	20 854	19 664	789
4	0.99	21 202	20 905	19 870	1 036
5	0.99	21 237	20 967	20 117	1 304
10	0.99	21 318	21 149	20 592	2 536
20	0.99	21 407	21 271	20 932	4 988
30	0.99	21 453	21 364	21 127	7 349
40	0.99	21 504	21 446	21 258	9 582
50	0.99	21 527	21 480	21 345	11 760

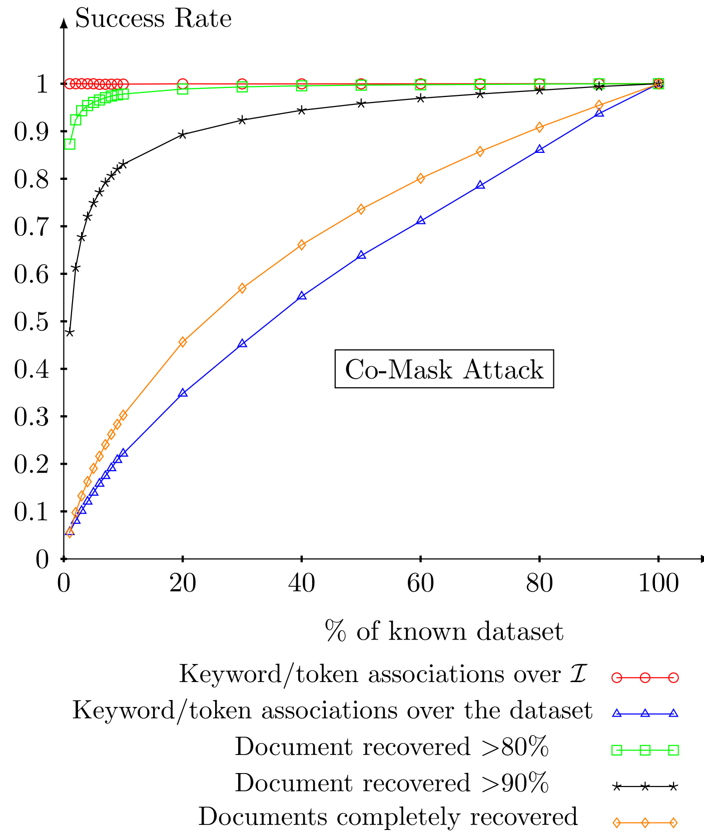
Fig. 5: Results of the Mask Attack on *Gutenberg* dataset.

C.2 Co-Mask Attack Results for L3 Leakage



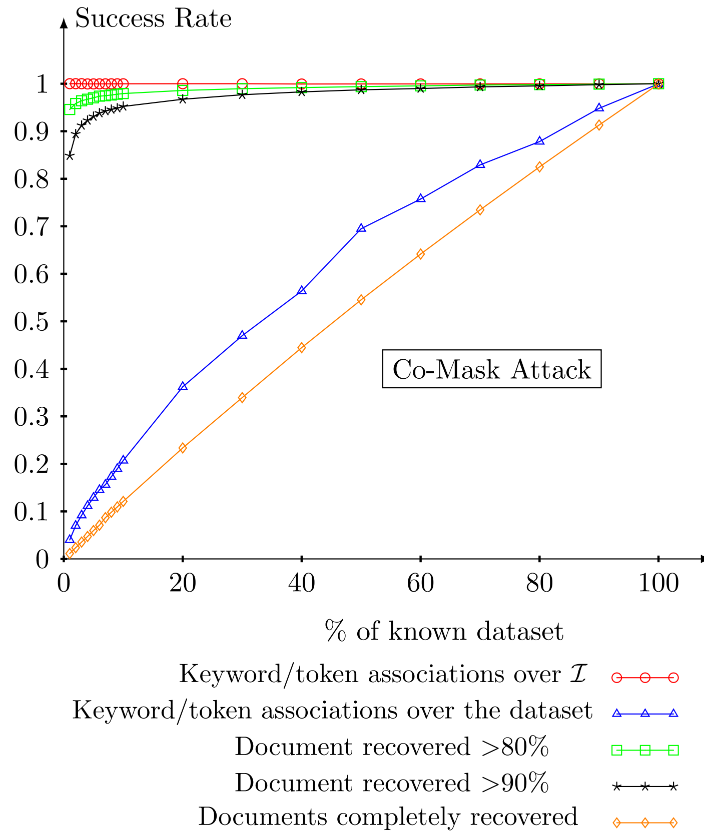
%	Rate	# 70%	# 80%	# 90%	# 100%
1	0.99	26 680	21 009	7 428	917
2	0.99	27 745	23 856	11 299	1 720
3	0.99	28 166	25 273	13 946	2 489
4	0.99	28 329	25 937	15 564	3 190
5	0.99	28 454	26 392	16 840	3 815
10	0.99	28 722	27 689	20 974	6 728
20	0.99	28 853	28 352	24 271	11 129
30	0.99	28 903	28 613	25 835	14 543
40	0.99	28 935	28 751	26 768	17 403
50	0.99	28 952	28 822	27 421	19 886

Fig. 6: Results of the Co-Mask Attack on *Commons* dataset.



%	Rate	# 70%	# 80%	# 90%	# 100%
1	0.99	57 258	51 394	28 086	3 240
2	0.99	57 882	54 405	36 092	5 742
3	0.99	58 081	55 529	39 880	7 810
4	0.99	58 190	56 170	42 419	9 575
5	0.99	58 264	56 567	44 106	11 212
10	0.99	58 524	57 598	48 905	17 811
20	0.99	58 701	58 222	52 586	26 880
30	0.99	58 771	58 489	54 371	33 542
40	0.99	58 806	58 620	55 589	38 913
50	0.99	58 832	58 701	56 430	43 333

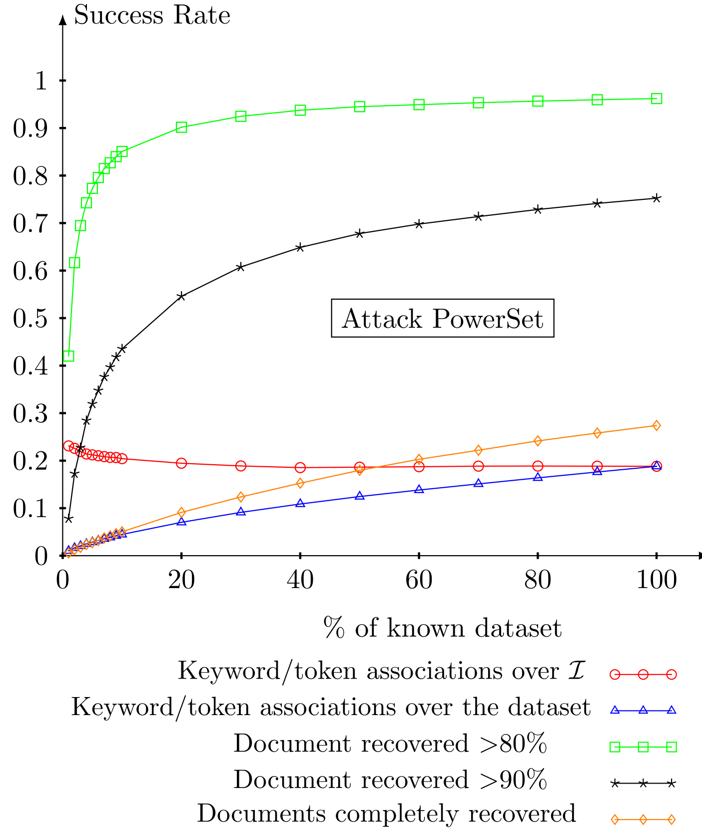
Fig. 7: Results of the Co-Mask Attack on *Lucene* dataset.



%	Rate	# 70%	# 80%	# 90%	# 100%
1	1	20 948	20 430	18 328	244
2	1	21 082	20 701	19 313	509
3	1	21 149	20 826	19 704	766
4	0.99	21 181	20 896	19 932	1 019
5	0.99	21 225	20 959	20 109	1 288
10	0.99	21 317	21 154	20 568	2 613
20	0.99	21 401	21 295	20 894	5 044
30	0.99	21 470	21 375	21 099	7 332
40	0.99	21 505	21 424	21 229	9 603
50	0.99	21 532	21 468	21 323	11 778

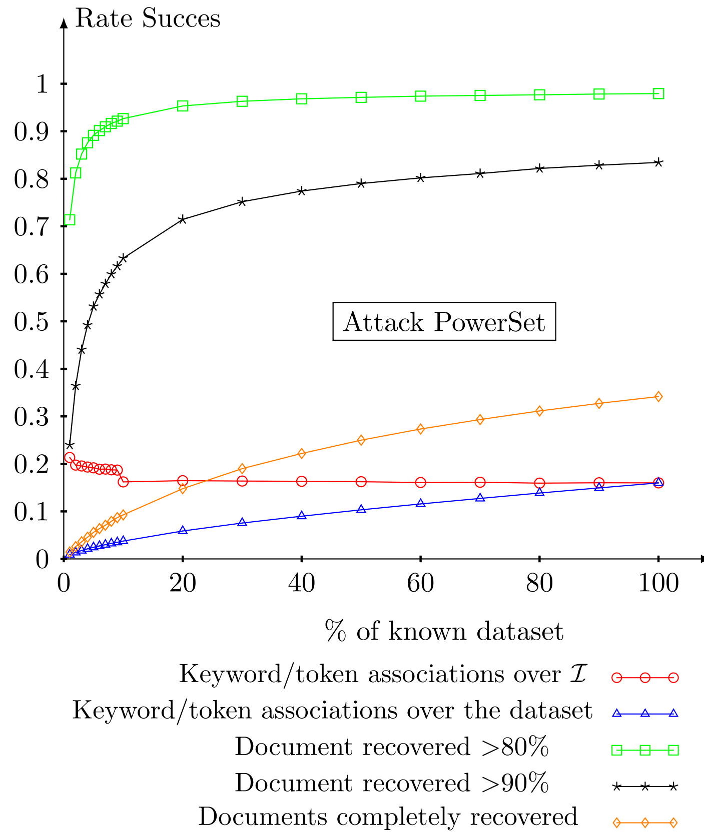
Fig. 8: Results of the Co-Mask Attack on *Gutenberg* dataset.

C.3 PowerSet Attack Results for L2 Leakage



%	Rate	# 70%	# 80%	# 90%	# 100%
1	0.23	21 664	9 516	909	51
2	0.22	25 587	17 878	5 007	363
3	0.21	26 430	20 140	6 598	511
4	0.21	26 927	21 532	8 248	688
5	0.20	27 187	22 460	9 249	794
10	0.19	27 929	24 529	12 472	1 385
20	0.18	28 383	26 187	15 715	2 475
30	0.18	28 508	26 780	17 447	3 326
40	0.18	28 588	27 126	18 573	4 059
50	0.18	28 634	27 354	19 460	4 784

Fig. 9: Results of the PowerSet Attack on *Commons* dataset.



%	Rate	# 70%	# 80%	# 90%	# 100%
1	0.21	54 839	42 016	14 117	855
2	0.20	56 429	47 828	21 453	1 522
3	0.19	57 013	50 163	25 931	2 120
4	0.19	57 306	51 557	28 974	2 691
5	0.19	57 517	52 483	31 285	3 285
10	0.16	57 899	54 556	37 260	5 458
20	0.16	58 217	56 124	42 062	8 704
30	0.16	58 325	56 703	44 257	11 187
40	0.16	58 396	57 008	45 579	13 064
50	0.16	58 438	57 188	46 513	14 713

Fig. 10: Results of the PowerSet Attack on *Lucene* dataset.