Similar Data is Powerful: Enhancing Inference Attacks on SSE with Volume Leakages

Björn Ho, Huanhuan Chen(✉), Zeshun Shi(✉), and Kaitai Liang

Delft University of Technology, 2628 CD Delft, The Netherlands
b.i.y.l.ho@student.tudelft.nl
{h.chen-2, z.shi-2, kaitai.liang}@tudelft.nl

Abstract. Searchable symmetric encryption (SSE) schemes provide users with the ability to perform keyword searches on encrypted databases without the need for decryption. While this functionality is advantageous, it introduces the potential for inadvertent information disclosure, thereby exposing SSE systems to various types of attacks. In this work, we introduce a new inference attack aimed at enhancing the query recovery accuracy of RefScore (presented at USENIX 2021). The proposed approach capitalizes on both similar data knowledge and an additional volume leakage as auxiliary information, facilitating the extraction of keyword matches from leaked data. Empirical evaluations conducted on multiple real-world datasets demonstrate a notable enhancement in query recovery accuracy, up to 19.5%. We also analyze the performance of the proposed attack in the presence of diverse countermeasures.

Keywords: Searchable Symmetric Encryption · Leakage · Inference Attack · Volume Pattern.

1 Introduction

Outsourcing data on cloud introduces a significant challenge: the loss of control over data access. This challenge becomes particularly concerning when dealing with sensitive personal information. Ensuring confidentiality becomes paramount to safeguarding data from unauthorized access. Rather than storing data in plain text, encryption is employed to obfuscate the data prior to its upload to the cloud, thereby enhancing its security.

Encrypting all data is a fundamental strategy for ensuring the security of data stored on the cloud, provided that encryption methodologies are appropriately implemented. This approach inherently entails trade-offs. For instance, a fully encryption over a database introduces complexities for data retrieval and access - a naive approach involves downloading all encrypted data from the cloud server and subsequently decrypting it locally to access the data. However, this proves impractical, particularly when dealing with extensive data sizes, especially if the client just would like to retrieve a small subset of documents.

Utilizing a *Searchable Symmetric Encryption* (SSE) is a practical approach for facilitating search functionalities on encrypted data. SSE, as proposed by

Song et al. [14], provides this capability by enabling clients to perform keyword searches on encrypted data stored in cloud environments using a *trapdoor*, interchangeably referred to as a *query*. As a result, the server is able to retrieve and return encrypted files containing the queried keyword to clients. This methodology upholds confidentiality but also empowers clients with the capacity to search for specific documents within the encrypted dataset. While the design of keyword search on encrypted data can conceal the underlying information of keyword, the interactions of sending queries and retrieving data inadvertently leaks useful information, e.g., frequency of keywords. For instance, observing the encrypted files returned during a search allows attackers to discern valuable insights related to say size of files. This type of leakage easily poses a potential threat to SSE systems. We note that in this paper we mainly focus on an observation type of attack, i.e. passive attack, against the current SSE systems.
Related Work. The IKK attack [6] laid the groundwork in SSE, exploiting full document knowledge and access pattern leakage to recover queries, and optimizing the query-keyword mapping through simulated annealing. Building on this, the Count attack [2] included result length patterns, improving efficiency and accuracy, but still requiring extensive document knowledge. In contrast, the Search attack [8] utilized known search patterns for keyword assignment, dependent on a large number of client queries for precision. The Graph Matching attack [12] catered to EDESE schemes using co-occurrence and reduced the challenge to weighted graph matching. Volume-centric attacks like Volan and its advancement, SelVolan [1], harnessed volumetric leakage and result length patterns but necessitated almost complete document knowledge. The Subgraph attack [1] introduced atomic patterns and delivered better accuracy even with low known data rates. LEAP [11] employed partial document knowledge and matrix mappings for accurate keyword recovery without false positives. Lastly, Score and RefScore attacks [4] employed co-occurrence patterns, with RefScore enhancing precision through ongoing knowledge updates. The effectiveness of each attack is a balance between knowledge requirements, leakage exploitation, and practical applicability. We note that RefScore [4] abuses the similar data knowledge but does not fully exploit its matching techniques. Surprisingly, we also find out that a combination of volume pattern enables attackers to match more queries. Please note that there have been various interesting works concerning SSE attacks and the countermeasures proposed in the literature, such as [9,10,17,16]. As they are not closely connected to this work, we refer interested readers to them.

Table 1 shows an overview of SOTA and the proposed attacks. They have different auxiliary knowledge requirements and exploited patterns. Full or partial document knowledge is a common requirement for known data attacks, and many attacks utilize the co-occurrence pattern. We focus on the state-of-art inference attacks: Score and RefScore [4]. We note that inference attacks only require the information of similar documents instead of partial documents from the real dataset; therefore, they are more practical than other types of attacks. We therefore propose three new attacks in Section 2, which are highlighted in

Table 1: Overview of SSE Attacks

SSE Attacks	Type	Document Knowledge	Query Knowledge	Exploited Patterns
IKK[6]	Known data	Full	None	Co-occ
Count[2]	Known data	Partial	None	Co-occ, rlen
Search[8]	Inference	Keywords	Query search patterns	Search
Graph matching[12]	Inference	Similar	None	Co-occ
Volan[1]	Known data	Partial	None	Tvol
SelVolan[1]	Known data	Partial	None	Tvol, rlen
Subgraph[1]	Known data	Partial	None	Rid or Vol
Score[4]	Inference	Similar	Partial	Co-occ
RefScore[4]	Inference	Similar	Partial	Co-occ
VolScore [Sect. 2.2]	Inference	Similar	Partial	Co-occ, vol
RefVolScore [Sect. 2.3]	Inference	Similar	Partial	Co-occ, vol
ClusterVolScore [Sect. 2.4]	Inference	Similar	Partial	Co-occ, vol

green in Table 1. The main idea in the proposed attacks is to utilize a technique from an existing attack and to explore combining an additional exploited pattern to improve the accuracy. The intuition is that an additional exploited pattern would increase the query recovery accuracy since the attacker has more information. We consider improving the co-occurrence pattern attacks with additional leakages in the volume pattern. A similar technique can be seen from the work of Lambregts et al. [7], which improves the accuracy of assess pattern with volume pattern.

Revisit.

• The Score attack employs known queries and similar documents to extract the keyword vocabulary set K_{sim} from the similar document D_{sim}, and then compute the keyword co-occurrence submatrix C^s_{kw}. The trapdoor co-occurrence submatrices C^s_{td} are generated from observed trapdoors. Central to this approach is the scoring mechanism. It involves comparing co-occurrence vectors $C^s_{kw}[kw]$ for keywords from K_{sim} and $C^s_{td}[td]$ for trapdoors from comprising observed queries Q. Their similarity indicates potential matches. Damie et al. [4] introduced a scoring method that employs the Euclidean norm and a negative logarithm to simplify small distances into more interpretable scores. The algorithm, therefore, selects the highest-scoring keyword for each trapdoor, iterating through all trapdoors.

• Building on the Score Attack, RefScore introduces an additional parameter: RefSpeed (refinement speed), which balances accuracy and runtime. The algorithm's first phase involves identifying unknown queries for each refinement cycle. Unlike the Score Attack, it employs certainty — the score gap between the top two candidates — to guide predictions, with low certainty indicating closely scored top candidates. Subsequently, the algorithm decides whether to continue refining or stop, based on the number of unknown queries and RefSpeed. During refinement, only predictions with the highest certainty are added as known queries, followed by updating the co-occurrence submatrices for the next iteration. Note that [4] demonstrates that RefScore significantly surpasses the base Score Attack in performance, given the same amount of known queries.

Refine. To enhance the RefScore attack, we leverage the followings:

- *Clustering.* Instead of using a static RefSpeed parameter, we consider dynamic adjustments based on certainty levels within clusters. Larger, high-certainty clusters would have a higher RefSpeed, while low certainty would reduce RefSpeed, limiting wrong predictions. This idea became central to the new attack strategy.
- *Volume Pattern.* The intuition is that providing the co-occurrence pattern with more knowledge by merging the volume pattern, therefore improving the accuracy.

In summary, we delve into the effects of integrating supplementary leakage information into SSE attacks. The main contributions are as follows.

• This work expands upon the inference attack by incorporating the volume pattern, achieving significant improvements over the existing attack methodology [4] to enhance query recovery rate, particularly in scenarios where the attacker has access to a limited number of known queries, and to improve stability in result spread due to the utilization of the additional volume leakage pattern.

• We conduct a comprehensive evaluation of the proposed attack approach, comparing its performance against the original attack across various datasets. We also investigate the effectiveness of countermeasures in mitigating the impact of the enhanced attack strategy.

2 The Proposed Attacks

We outline the novel inference attacks on SSE. The approach rests on several key assumptions typical in SSE attacks. First, post-query, both the co-occurrence and volume patterns are leaked to the server. Second, we presume two types of leakages: co-occurrence (access pattern) and volume pattern, each revealing specific document identifiers and their volumes for every query. Third, the attacker is assumed to have access to a dataset similar to the server's, including a shared keyword distribution and the same keyword extraction method, with a focus on the most frequent keywords. Finally, the attacker possesses known queries, comprising keywords from both their and the client's vocabularies. The commonly used notions are shown in Table 3.

2.1 Intuition

The main intuition behind the design is that an attacker has more attack power with more knowledge. In detail, we increase the knowledge of the adversary by utilizing the `volume pattern` and designed the VolScore attack while keeping many elements and the core matching technique the same as RefScore from Damie et al. [4]. The goal of this initial attack is not to achieve better results than RefScore, instead, it is a way to obtain additional knowledge and initial query to keyword predictions. We combine the results obtained via the `volume pattern` and `co-occurrence pattern` by utilizing VolScore, with results via the `co-occurrence pattern` that are obtained from RefScore.

RefScore itself creates query-to-keyword predictions by using a number of known queries, but the adversary does not know which predictions are assigned correctly, and which are assigned incorrectly. By using the additional predictions that are obtained from VolScore, and intersecting with the predictions from RefScore, the attacker becomes more certain about some of the predictions, or in other words the attacker has acquired more knowledge. The attacker can now start a fresh new attack, but with an increased amount of known queries, and with more known queries, the accuracy of the attack is improved.

In Figure 1 we visualized this concept. The chain of attacks that consists of running VolScore, RefScore and another RefScore with increased knowledge is called RefVolScore. We can also replace the last RefScore attack with a modified version that utilizes clustering, and that chain is called ClusterVolScore. We explain each of these attacks in more detail in the following sections.

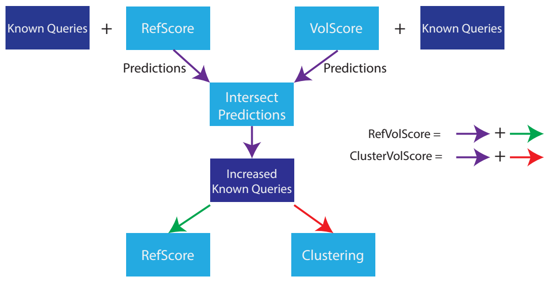


Fig. 1: Volscore Overview

2.2 VolScore

The first attack VolScore is shown below as Algorithm 1, which works by utilizing the volume pattern in the RefScore algorithm. Compared to RefScore, the main difference is that we apply this technique to a covolume matrix, which is created by volume pattern, instead of a co-occurrence matrix. This change has consequences for the accuracy. The prediction accuracy is much lower compared to RefScore, but it does contain a small portion of useful predictions. As mentioned before, we do not aim here to have a higher accuracy here than RefScore, instead, we aim to acquire additional knowledge with this small part of useful predictions.

The algorithm has three important phases. In the first phase, we need to have covolume submatrices prepared and extract the remaining unknown queries for which a prediction needs to be made. In the second phase, the scoring mechanism is applied and we make a prediction for each unknown query. And in the last phase, the algorithm either stops and returns its predictions, or expands the new knowledge into the covolume submatrices for the next iteration. We will discuss each phase in the following sections.

Algorithm 1 VolScore

```
1: Input: $K_{sim}, V^s_{kw}, Q, V^s_{td}$, KnownQ, RefSpeed
2: Output: $final\_pred$
3:
4: $final\_pred \leftarrow []$
5: unknownQ $\leftarrow Q$
6: while unknown $Q \neq \emptyset$ do
7:     // Extract the remaining unknown queries
8:     unknownQ $\leftarrow \{td : (td \in Q) \wedge (\nexists kw \in K_{sim} : (td, kw) \in KnownQ)\}$
9:     $temp\_pred \leftarrow []$
10:
11:     // Propose a prediction for each unknown query
12:     for all $td \in$ unknownQ do
13:         $cand \leftarrow []$                                  ▷ The candidates for trapdoor $td$
14:         for all $kw \in K_{sim}$ do
15:             $s = -\ln(||V^s_{kw}[kw] - V^s_{td}[td]||)$
16:             append {"kw": $kw$, "score":$s$} to $cand$
17:         end for
18:         Sort $cand$ in descending order according to the score.
19:         certainty $\leftarrow$ score($cand$[0]) - score($cand$[1])
20:         append ($td$, kw($cand$[0]), certainty) to $temp\_pred$
21:     end for
22:
23:     // Either stop the algorithm or keep refining.
24:     if | unknownQ | < RefSpeed then
25:         $final\_pred \leftarrow$ KnownQ $\cup$ $temp\_pred$
26:         unknownQ $\leftarrow \emptyset$
27:     else
28:         Append the RefSpeed most certain predictions from $temp\_pred$ to KnownQ
29:         Add the columns corresponding to the new known queries to $V^s_{kw}$ and $V^s_{td}$
30:     end if
31: end while
32: return $final\_pred$
```

Preparing Covolume Matrices. In the RefScore attack, the access pattern is leaked. The co-occurrence pattern is the backbone of this attack. The attacker is able to learn which document identifiers are accessed on single keywords. And when more queries/trapdoors are issued, the attacker is able to learn information about documents that contain both keywords. The co-occurrence is calculated and utilized for the attack.

In the VolScore attack, we use a similar approach but with an additional volume pattern. From each query that is issued, we learn the document identifiers (access pattern/co-occurrence pattern) as well as the volume of each returned document (volume pattern). From each pair of queries, the attacker learns about document identifiers that contain both keywords, as well as the volume of those documents. The sum of the returned document volumes that contain both keywords (or queries) divided by the number of documents is the covolume. The details to compute the covolume are shown below in Algorithm 2.

The parameter `kq_set` is a set of keywords or queries. This is because the covolume needs to be calculated for each keyword pair, as well as each trapdoor pair. The `inv_index` is the inverted index that contains a mapping from keyword to documents, or from query to document identifiers. The volume of each document is stored in `vol_array`. The parameter `nr_docs` is the amount of documents from the similar auxiliary document knowledge when computing

Algorithm 2 Compute covolume

```
1: Input: kq_set, inv_index, vol_array, nr_docs
2: Output: results
3:
4: results ← []
5: for all $kw_i$ ∈ kq_set do
6:     $kw_i$_docs ← inv_index[$kw_i$]
7:     for all $kw_j$ ∈ kq_set do
8:         $kw_j$_docs ← inv_index[$kw_j$]
9:         co_docs ← intersect($kw_i$_docs, $kw_j$_docs)
10:    end for
11:    if length(co_docs) > 0 then
12:        tvol ← 0
13:        for all doc ∈ co_docs do
14:            tvol ← tvol + vol_array[doc]
15:        end for
16:        co_vol_result ← tvol / nr_docs
17:    else
18:        co_vol_result ← 0
19:    end if
20:    append(co_vol_result) to results
21: end for
22: return results
```

for keywords, or the predicted amount of documents stored on the server when computing for queries.

We predict this number by using known queries in the input for RefScore. A known query is a pair (td_i, kw_i) with a known trapdoor and keyword by the attacker. Due to the access pattern leakage, the amount of documents that contain trapdoor td_i is known and can be divided by the amount of documents that contain kw_i from the auxiliary dataset. If we repeat this calculation for every known query and compute the mean, we obtain an estimated ratio of real documents stored on the server.

In the algorithm, we compute co_docs that are documents that contain both kw_i and kw_j in line 8. And then, the covolume is computed by dividing the total volume of documents that contains both keywords by nr_docs in line 15.

In Figure 2 we show an example of two covolume matrices V_{kw} and V_{td} that are covolume matrices for keyword and trapdoors respectively. The diagonal consists of zeroes since we are not interested in covolumes that consist of the same keywords.

$$\text{Covol matrix } V_{kw} = \begin{array}{c|cccc} & kw_1 & kw_2 & \cdots & kw_n \\ \hline kw_1 & 0 & 20 & \cdots & 80 \\ kw_2 & 20 & 0 & \cdots & 40 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ kw_n & 80 & 40 & \cdots & 0 \end{array} \qquad \text{Covol matrix } V_{td} = \begin{array}{c|cccc} & td_1 & td_2 & \cdots & td_l \\ \hline td_1 & 0 & 40 & \cdots & 80 \\ td_2 & 40 & 0 & \cdots & 20 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ td_l & 80 & 20 & \cdots & 0 \end{array}$$

Fig. 2: Covolume matrix example

From the covol matrices that we created, we need to use known queries to create two submatrices. Let's assume that the attacker has two known queries

(td_l, kw_1) and (td_2, kw_2). To create the keyword covol submatrix, we extract columns that consist of keywords from the known queries from the original covol submatrix V_{kw}, in this case, it is kw_1 and kw_2. In Figure 3 the column extraction is shown, and the new covol submatrix V^s_{kw} is created.

$$\text{Covol matrix } V_{kw} = \begin{array}{c} \\ kw_1 \\ kw_2 \\ \vdots \\ kw_n \end{array} \!\!\begin{array}{c} \begin{array}{cccc} kw_1 & kw_2 & \cdots & kw_n \end{array} \\ \begin{pmatrix} 0 & 20 & \cdots & 80 \\ 20 & 0 & \cdots & 40 \\ \vdots & \vdots & \ddots & \vdots \\ 80 & 40 & \cdots & 0 \end{pmatrix} \end{array} \xrightarrow{\text{Extract columns}} \text{Covol submatrix } V^s_{kw} = \begin{array}{c} \\ kw_1 \\ kw_2 \\ \vdots \\ kw_n \end{array} \!\!\begin{array}{c} \begin{array}{cc} kw_1 & kw_2 \end{array} \\ \begin{pmatrix} 0 & 20 \\ 20 & 0 \\ \vdots & \vdots \\ 80 & 40 \end{pmatrix} \end{array}$$

Fig. 3: Covolume submatrix for keywords

We create another covol submatrix V^s_{td} for the trapdoors as shown in Figure 4 but now we extract columns based on the known trapdoors instead of keywords. We also have to reorder the columns in the same order as V^s_{kw} such that the submatrices can be compared.

$$\text{Covol matrix } V_{td} = \begin{array}{c} \\ td_1 \\ td_2 \\ \vdots \\ td_l \end{array} \!\!\begin{array}{c} \begin{array}{cccc} td_1 & td_2 & \cdots & td_l \end{array} \\ \begin{pmatrix} 0 & 40 & \cdots & 80 \\ 40 & 0 & \cdots & 20 \\ \vdots & \vdots & \ddots & \vdots \\ 80 & 20 & \cdots & 0 \end{pmatrix} \end{array} \xrightarrow{\text{Extract columns + reorder}} \text{Covol submatrix } V^s_{td} = \begin{array}{c} \\ td_1 \\ td_2 \\ \vdots \\ td_l \end{array} \!\!\begin{array}{c} \begin{array}{cc} td_l & td_2 \end{array} \\ \begin{pmatrix} 80 & 40 \\ 20 & 0 \\ \vdots & \vdots \\ 0 & 20 \end{pmatrix} \end{array}$$

Fig. 4: Covolume submatrix for trapdoors

Scoring Mechanism. In the second phase of the algorithm, we need to make a prediction for each unknown query and apply a scoring mechanism. For all keywords that are in the similar dataset a score number is assigned for the current unknown trapdoor. From the previous example td_1 is an unknown trapdoor, and the trapdoor vector from covol submatrix V^s_{td} is row td_1. This trapdoor vector is compared with each possible keyword vector that comes from each row from covol submatrix V^s_{kw} by calculating the Euclidean distance between the two vectors, followed by applying the negative natural logarithm on the result. This transforms the score numbers so that the focus is on the order of magnitude and readability, especially when score numbers could be very small and close to zero.

$$\text{Covol submatrix } V^s_{td} = \begin{array}{c} \\ td_1 \\ td_2 \\ \vdots \\ td_l \end{array} \!\!\begin{array}{c} \begin{array}{cc} td_l & td_2 \end{array} \\ \begin{pmatrix} 80 & 40 \\ 20 & 0 \\ \vdots & \vdots \\ 0 & 20 \end{pmatrix} \end{array} \xleftrightarrow{\text{Compare row vectors}} \text{Covol submatrix } V^s_{kw} = \begin{array}{c} \\ kw_1 \\ kw_2 \\ \vdots \\ kw_n \end{array} \!\!\begin{array}{c} \begin{array}{cc} kw_1 & kw_2 \end{array} \\ \begin{pmatrix} 0 & 20 \\ 20 & 0 \\ \vdots & \vdots \\ 80 & 40 \end{pmatrix} \end{array}$$

Fig. 5: Computing the score

In Figure 5 trapdoor vector [80, 40] from V_{td}^s is compared with each row vector: [0, 20], [20, 0], ..., [80, 40] from V_{kw}^s. The last row for keyword kw_n has an Euclidean distance of exactly zero and is the closest to td_1. Usually, this does not occur and we omitted this check in Algorithm 1, but in the implementation the score is set to infinite. In the end, all scores from each keyword are calculated and sorted in descending order. The certainty is the score between the highest and the second highest candidate keyword. In this example, kw_n has the highest score of infinite, so the certainty will also be very high. So this candidate keyword, together with the current trapdoor td_1 and its certainty will be added to a list of temporary predictions `temp_pred`. The algorithm proceeds to do the same process but now with the remaining unknown trapdoors and fills the temporary predictions list, where each entry contains a trapdoor, a candidate keyword, and a certainty. The actual decision is made in the last phase of the algorithm which we discuss next.

Decision Making. In the last phase of the algorithm, the algorithm either stops or keeps refining. If the algorithm keeps refining, a RefSpeed amount of predictions from `temp_pred` is added to the list of known queries. These are the predictions with the highest certainties. The algorithm has acquired more known queries now, so the covolume submatrices V_{kw}^s and V_{td}^s can expand its columns with new known queries. So for the next refinement, a larger portion of the original covolume matrices will be used as submatrices.

In the previous examples, we originally had two known queries (td_l, kw_1) and (td_2, kw_2). For simplicity, let's assume that we have only one new known query: (td_1, kw_n). In Figure 6 the old covol submatrices are expanded by a green column, which reflects the newly added known query. After expansion, the algorithm repeats itself and uses the new covol submatrices until the stop condition is met. The algorithm stops if the number of unknown queries is less than the refinement speed. If so, the stopping criteria are set and the known queries with the current `temp_pred` are returned as the final prediction.

$$\text{New covol submatrix } V_{td}^s = \begin{array}{c} \\ td_1 \\ td_2 \\ \vdots \\ td_l \end{array} \begin{array}{c} \begin{array}{ccc} td_l & td_2 & td_1 \end{array} \\ \begin{pmatrix} 80 & 40 & 0 \\ 20 & 0 & 40 \\ \vdots & \vdots & \vdots \\ 0 & 20 & 80 \end{pmatrix} \end{array} \qquad \text{New covol submatrix } V_{kw}^s = \begin{array}{c} \\ kw_1 \\ kw_2 \\ \vdots \\ kw_n \end{array} \begin{array}{c} \begin{array}{ccc} kw_1 & kw_2 & kw_n \end{array} \\ \begin{pmatrix} 0 & 20 & 80 \\ 20 & 0 & 40 \\ \vdots & \vdots & \vdots \\ 80 & 40 & 0 \end{pmatrix} \end{array}$$

Fig. 6: Covol submatrices expansion

2.3 RefVolScore

Previously, we discussed the RefScore and VolScore attacks. We combined both attacks and call this the RefVolScore attack as shown in Algorithm 4. The idea behind this attack is that the algorithm first runs the VolScore attack which uses the `volume pattern` with `co-occurrence pattern` to obtain results, as well as running the RefScore attack to obtain another result. From both results, we find predictions that are found in both results and are not already known. This means by using two different methods we have found the same new trapdoor to keyword assignments. Whereas if we only run VolScore or RefScore, the attacker does not know which prediction is correct. Since the same predictions are made with two different methods, it is highly likely that this is a correct prediction. These new known queries are then appended to the original known queries list. Then a fresh run of RefScore is run using these updated known queries to obtain a higher accuracy than was previously possible with RefScore by itself.

2.4 ClusterVolScore

We then introduce ClusterVolScore, an enhanced attack strategy in the sequence of attacks after RefScore and VolScore. This method incorporates clustering to dynamically adjust refinement speed based on certainty levels, rather than a fixed speed. High refinement speeds can lead to inaccurate predictions due to adding multiple low-certainty predictions simultaneously. On the other hand, a slow refinement speed might delay the process and risk accuracy when adding only a few predictions, which might be incorrect.

ClusterVolScore's approach is exemplified in Figure 7. Here, temporary predictions with associated certainties are shown. These certainties reflect the confidence in trapdoor-to-keyword assignments. Unlike RefScore, which adds a fixed number of elements (e.g., 10) to known queries, ClusterVolScore computes candidate clusters from sizes 1 to a maximum (`Max_Ref_Speed`, 10 in this case). It selects the cluster size with the largest difference between the cluster's last element and the next external element, thereby adding only those high-certainty elements to known queries.

Temp_pred with certainties $= \left[\mathbf{8.2}\ \mathbf{8}\ \mathbf{7.9}\ \mathbf{3}\ \mathbf{2.5}\ \mathbf{2.4}\ \mathbf{2.3}\ \mathbf{2}\ \mathbf{1.9}\ \mathbf{1.5}\ \mathbf{1.4} \cdots \right]$
Max_Ref_Speed = **10**

Best candidate clustering:
Cluster_size_1 = [8.2], diff = 8.2 - 8 = 0.2
Cluster_size_2 = [8.2, 8], diff = 8 - 7.9 = 0.1
Cluster_size_3 = [8.2, 8, 7.9], diff = 7.9 - 3 = 4.9
...
Cluster_size_Max_RefSpeed = [8.2, 8, $\cdots$, 1.5], diff = 0.1

Fig. 7: A clustering example. Chosen elements are highlighted.

In Algorithm 3, we detail the process of calculating the index with the largest certainty difference. In the given example, if indexing starts at zero, the index with the largest certainty difference is 2. Therefore, we add elements from index 0 to 2 as known queries. This approach differs from the one in [4]. While they focus on identifying the best candidate cluster from a score set linked to a trapdoor, the method centers on finding the best index from a list of certainties representing all trapdoor to keyword assignments. The design goal is to pinpoint the index marking the cluster with the greatest certainty difference. This index helps us adjust the refinement speed, adding only those predictions associated with a single keyword to trapdoor assignment.

The clustering algorithm is presented in Algorithm 5. Unlike static refinement speeds, we employ a maximum refinement speed, updated dynamically to the index with the largest certainty difference from Algorithm 3. We omit the detailed ClusterVolScore algorithm as it's a minor variation of RefVolScore (Algorithm 4). The key difference is on line 33, where instead of initiating RefScore, we call the clustering algorithm with a predefined maximum refinement speed.

Algorithm 3 index_max_diff

```
Input: sorted_tuples, max_ref_speed
Output: ind_max_diff

// Take a subset of all sorted tuples.
// We add 1 element so that we can make a comparison for the last element.
sub_tuples ← sorted_tuples[:(max_ref_speed + 1)]
diff_list ← []
current_index ← 0
// Loop through tuples but without the additional element we added before.
for all tuple ∈ sub_tuples[:-1] do
    // Calculates difference of certainties for current_index
    append (current_index, tuple[2] - sub_tuples[current_index + 1][2]) to diff_list
    current_index ← current_index + 1
end for
// Get max diff and its index
ind_max_diff ← 0
if len(diff_list) > 0 then
    // Max based on diff value and retrieve index
    ind_max_diff ← get_index(max(diff_list))
else
    // Only 1 element, so index is zero.
    ind_max_diff ← 0
end if
// Returns the index that has the largest difference of certainties
return ind_max_diff
```

3 Experimental Evaluation

We evaluate the performance of the new attacks over different datasets, which are listed as follows.

- **Enron dataset** [3]: It contains approximately 500,000 real emails and represents a realistic scenario of encrypted email storage and retrieval. We use 30,109 emails from the `_sent_mail` folder of each user for the presented experiments.

- **Apache Lucene** [5]: We used the "java-user" mailing list from Apache Lucene [5] between 2002 and 2011, following the setup from the Score attack [4]. The dataset consists of .mbox files for each month and year, and we extracted emails using a self-made script.

Table 2: Dataset Comparison After Pre-processing.

	Enron	**Apache**	**Wikipedia**
Total amount of documents	30.109	50.564	30.000
Total number of unique keywords	63.029	92.402	162.074
Number of unique volumes	4940	7094	4811
Avg. amount of keywords/document	57,37	77,99	65,0

- **Wikipedia** [15]: The simplified Wikipedia dataset from October 1st, 2023 [15], was processed using PlainTextWikipedia [13] into plaintext files. A more detailed comparison of datasets is shown in Table 2.

3.1 Methodology

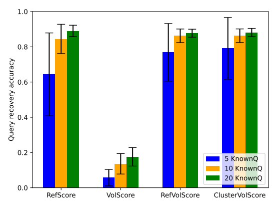
We conducted experiments on the previously mentioned datasets, repeating each experiment 20 times for statistical significance. Each experiment involved adjusting specific parameters and comparing the outcomes with the baseline results from RefScore. We initially divided the dataset into two parts: a 'similar' dataset (40%) representing auxiliary knowledge for the attacker and a 'real' dataset (60%) stored on the server. Keywords were extracted from datasets using NLTK in Python, excluding common words and email-specific terms like 'from', 'to', etc. The most frequent keywords formed the vocabulary for both datasets.
- **Keyword Extraction.** NLTK's PorterStemmer was used to extract keywords. The process included parsing documents, extracting, and then stemming keywords, followed by matching them with a stopword list to exclude common words. The selection of keywords for the datasets was based on their frequency of occurrence.
- **Query Selection.** Queries are assumed to be uniformly distributed. For each experiment, a random sample from the server's keyword space is selected, with each keyword having an equal chance of being chosen.
- **Adversary's Knowledge.** The attacker, possessing a similar dataset, can observe client-server communication. We assume leakage of both access and volume patterns, along with some known queries randomly selected from the common keywords.
- **Metrics.** Attack performance is measured by query recovery accuracy, following the definition in Damie et al. [4]. That is: $QR_{acc} = \frac{|correctPred(UnknownQ)|}{|Q|-|KnownQ|}$. We compare the average attack accuracy of RefScore and various VolScore attacks across different scenarios and parameter changes.
- **Testbed.** Experiments were run on an Arch Linux laptop with an AMD Ryzen 7 5800H CPU and 16GB RAM, using Python 3.11.

3.2 Results

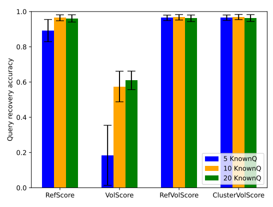
RefScore and VolScore. We evaluate the attacks using 5, 10, and 20 known queries across three datasets. Figure 8a demonstrates that VolScore's accuracy is consistently lower for all query sets. This is attributed to covolume's higher variability and lower uniqueness compared to co-occurrence, making RefScore more likely to correctly predict despite similar query-keyword distances. For 20 queries, RefScore, RefVolScore, and ClusterVolScore exhibit comparable performance. However, with 10 queries, RefVolScore and ClusterVolScore slightly surpass RefScore. Notably, with 5 queries, RefVolScore and ClusterVolScore significantly outperform RefScore, averaging 0.76 and 0.79 against RefScore's 0.64. This indicates that with fewer queries, the additional information from VolScore in RefVolScore and ClusterVolScore is beneficial.

Similar patterns are observed in the Apache and Wikipedia datasets (refer to Figure 8b and Figure 8c). High query counts yield similar attack efficiencies across methods, barring VolScore, suggesting a threshold beyond which additional queries do not significantly enhance accuracy. Conversely, at lower query counts, leveraging volume patterns in RefVolScore and ClusterVolScore yields more precise results than RefScore alone.

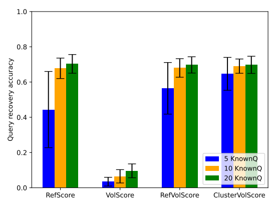
Apache exhibits the highest query recovery accuracy, while Wikipedia is the least accurate. This discrepancy is likely due to dataset characteristics (see Table 2). Wikipedia's challenge stems from its high keyword count balanced by a similar document count as Enron, complicating accurate predictions. Conversely, Apache's superior performance can be credited to its higher average keywords per document, greater document availability, and unique volume count.



(a) Enron



(b) Apache



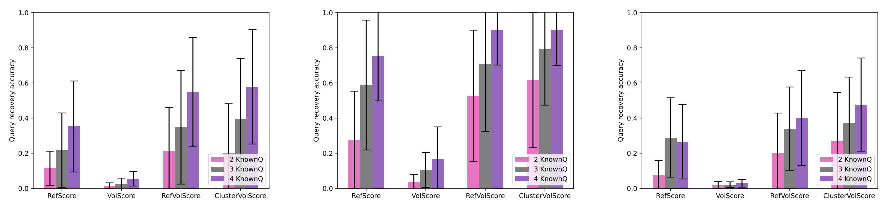
(c) Wiki

Fig. 8: Accuracy comparison between RefScore and VolScore. (a) Enron with $|D_{sim}| = 12K, |D_{real}| = 18K$. (b) Apache with $|D_{sim}| = 20K, |D_{real}| = 30K$. (c) Wiki with $|D_{sim}| = 12K, |D_{real}| = 18K$. $m_{sim} = 1.2K, m_{real} = 1K, |Q| = 150$, RefSpeed = 10 = MaxRefSpeed.

Low Known Queries Comparison. In experiments with numerous known queries, accuracy differences between attacks are minimal. Consequently, subsequent experiments focus on fewer known queries to evaluate performance against RefScore. Figure 9a presents results for 2, 3, and 4 known queries in Enron.

We note that RefVolScore and ClusterVolScore outshine RefScore with fewer known queries. This indicates that more known queries generally enhance performance up to a point. With fewer queries, RefVolScore and ClusterVolScore exploit additional queries sourced through VolScore, achieving greater accuracy than RefScore. Despite VolScore's lower standalone accuracy, its contribution of at least one extra known query noticeably boosts query recovery accuracy. This is evident as RefScore's mean accuracy with 3 known queries is comparable to RefVolScore's with 2. Similar patterns emerge in Apache and Wikipedia (Figures 9b and 9c). While Apache maintains higher overall accuracy, Wikipedia displays the lowest, yet RefVolScore and ClusterVolScore still surpass RefScore. Across all datasets, accuracy variability is pronounced with fewer known queries, indicating greater error potential and difficulty in stabilizing accuracy.

Table 4 consolidates these findings, linking VolScore accuracy to the discovery of new known queries. Higher VolScore accuracy correlates with identifying more known queries. The 'Total KnownQ Accuracy' metric reflects the proportion of accurately identified total known queries, including original and additional ones identified through VolScore. Key factors affecting accuracy and its stabilization include the number of available known queries, the discovery of new known queries through VolScore, the overall correctness of these known queries, and VolScore's accuracy. The Apache dataset, particularly with 4 known queries, exhibits the most stable accuracy spread. This stability is attributed to Apache's inherent positive attributes, leading to high VolScore accuracy and the identification of numerous new known queries with high overall accuracy.



(a) Enron (b) Apache (c) Wiki

Fig. 9: Low known queries comparison. (a) Enron with $|D_{sim}| = 12K, |D_{real}| = 18K$. (b) Apache with $|D_{sim}| = 20K, |D_{real}| = 30K$. (c) Wiki with $|D_{sim}| = 12K, |D_{real}| = 18K$. $m_{sim} = 1.2K, m_{real} = 1K, |Q| = 150$, RefSpeed = 10 = MaxRefSpeed.

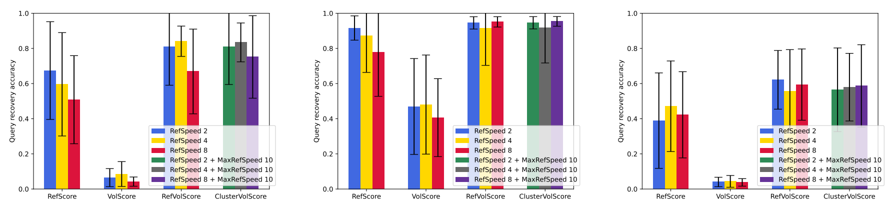
RefSpeed Comparison. We explore how RefSpeed affects attack accuracy, particularly focusing on ClusterVolScore. RefSpeed inversely affects accuracy and speed: lower speeds increase precision but are slower, while higher speeds are faster but less accurate. This experiment used four known queries.

Figure 10a and Figure 10b reveal that lower RefSpeeds improve RefScore's accuracy due to cautious knowledge integration, despite slower performance. In contrast, Wikipedia's RefScore accuracy remains low and less impacted by RefSpeed (Figure 10c). RefVolScore and ClusterVolScore exhibit similar accuracy trends within datasets. In Enron, RefSpeed 2 outperforms RefSpeed 8, but in

Apache, the performances are similar due to VolScore's high accuracy. Wikipedia shows negligible differences between RefSpeeds 2 and 8 (Figure 10c).

The impact of RefSpeed varies depending on the attack and dataset. For RefScore, lower speeds ensure precision. For RefVolScore and ClusterVolScore, the effect is dependent on dataset characteristics. Apache's ample documents and high keyword count minimize RefSpeed's impact, similar to Wikipedia's fewer documents and large keyword pool. Enron, with its lower unique keyword count, is more sensitive to RefSpeed changes.

Despite similarities in RefVolScore and ClusterVolScore performance, Table 5 shows ClusterVolScore underperforms. Its runtime is longer and mean dynamic RefSpeed lower, making it less efficient than RefVolScore. The dynamic RefSpeed based on certainty differences is less effective than expected, as it often leads to adding fewer, not more, predictions. Thus, RefVolScore emerges as a more effective attack method due to its better accuracy and efficiency.



(a) Enron (b) Apache (c) Wiki

Fig. 10: RefSpeed comparison. (a) Enron with $|D_{sim}| = 12K, |D_{real}| = 18K$. (b) Apache with $|D_{sim}| = 20K, |D_{real}| = 30K$. (c) Wiki with $|D_{sim}| = 12K, |D_{real}| = 18K$. $m_{sim} = 1.2K, m_{real} = 1K, |Q| = 150, KnownQ = 4, MaxRefSpeed = 10$.

Against Countermeasures. This experiment assesses the impact of countermeasures on RefScore and RefVolScore. We focus solely on these attacks, omitting other VolScore variants, and test under a worst-case scenario with more known queries available to the attacker than in previous experiments. Additionally, we vary the vocabulary size, a deviation from previous fixed-size approaches, and set parameters as per the original authors' countermeasure setup.

In Enron (Figure 11a), increasing vocabulary size challenges query recovery. With 2000 vocabulary size and no countermeasures, RefScore remains effective. Volume hiding does not affect RefScore, as it does not rely on volume patterns. Padding, particularly with large vocabulary sizes, effectively counters RefScore. For RefVolScore (Figure 11d), volume hiding is again minimally impactful. Although it targets volume pattern leakage, RefVolScore's reliance is not solely on this pattern. Padding significantly impairs RefVolScore, given its dependence on RefScore. A vocabulary size of 2000 with padding successfully neutralizes the attack. In Apache (Figure 11b and Figure 11e), attackers can recover most queries without countermeasures or with volume hiding, regardless of vocabulary size. This indicates that merely increasing vocabulary size is ineffective without countermeasures. Padding's effectiveness is also evident in Apache. A vocabulary size of 1000 with padding is not sufficient against both attacks, but 2000 with padding is. Results for Wikipedia (Figure 11c and Figure 11f) mirror those of Enron due to similar document counts, contrasting with Apache's higher count.

Conclusively, an attack's success varies with dataset characteristics. Datasets with many documents and high average keywords per document, like Apache, are more vulnerable to attacks. In contrast, datasets with fewer documents and lower average keywords, like Enron and Wikipedia, still face considerable query recovery rates at lower vocabulary sizes. An effective defense involves padding coupled with increased vocabulary size.

4 Conclusion

We explore the impact of additional leakage knowledge on SSE attacks, particularly focusing on combining the co-occurrence and volume patterns to create the Refined Score Attack, yielding improved query recovery accuracy. Regarding countermeasures, padding can decrease query recovery accuracy of the proposed attacks, while volume hiding alone is insufficient. We conclude that an improved inference attack can be designed by integrating more knowledge and patterns, and it can be mitigated using padding and large keyword vocabularies. Dataset properties play a crucial role in attack performance and should be considered during evaluation. This research contributes to understanding and improving SSE attack methods and defense mechanisms.

Acknowledgments

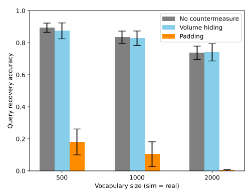
We thank the anonymous reviewers for their valuable comments. This work was partly supported by the EU Horizon Europe Research and Innovation Programme under Grant No. 101073920 (TENSOR), No. 101070627 (REWIRE), and No. 101070052 (TANGO).

References

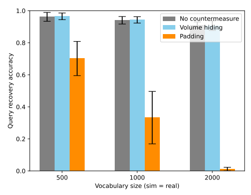
1. Blackstone, L., Kamara, S., Moataz, T.: Revisiting leakage abuse attacks. Cryptology ePrint Archive (2019)
2. Cash, D., Grubbs, P., Perry, J., Ristenpart, T.: Leakage-abuse attacks against searchable encryption. In: CCS '15. p. 668–679. Association for Computing Machinery, New York, NY, USA (2015). https://doi.org/10.1145/2810103.2813700
3. Cohen, W.W.: Enron email dataset (2015), https://www.cs.cmu.edu/~enron/
4. Damie, M., Hahn, F., Peter, A.: A highly accurate Query-Recovery attack against searchable encryption using Non-Indexed documents. In: 30th USENIX Security Symposium (USENIX Security 21). pp. 143–160. USENIX Association (Aug 2021), https://www.usenix.org/conference/usenixsecurity21/presentation/damie

5. Foundation, A.S.: Apache lucene emails (2002), https://lists.apache.org/list?java-user@lucene.apache.org
6. Islam, M.S., Kuzu, M., Kantarcioglu, M.: Access pattern disclosure on searchable encryption: Ramification, attack and mitigation. In: NDSS. The Internet Society (2012), http://dblp.uni-trier.de/db/conf/ndss/ndss2012.html#IslamKK12
7. Lambregts, S., Chen, H., Ning, J., Liang, K.: Val: Volume and access pattern leakage-abuse attack with leaked documents. In: Atluri, V., Di Pietro, R., Jensen, C.D., Meng, W. (eds.) Computer Security – ESORICS 2022. pp. 653–676. Springer International Publishing, Cham (2022)
8. Liu, C., Zhu, L., Wang, M., an Tan, Y.: Search pattern leakage in searchable encryption: Attacks and new construction. Information Sciences **265**, 176–188 (2014). https://doi.org/https://doi.org/10.1016/j.ins.2013.11.021
9. Liu, D., Wang, W., Xu, P., Yang, L., Luo, B., Liang, K.: d-dse: Distinct dynamic searchable encryption resisting volume leakage in encrypted databases (2024). https://doi.org/arXiv preprint arXiv:2403.01182
10. Nie, H., Wang, W., Xu, P., Zhang, X., Yang, L., Liang, K.: Query recovery from easy to hard: Jigsaw attack against sse (2024). https://doi.org/arXiv preprint arXiv:2403.01155
11. Ning, J., Huang, X., Poh, G.S., Yuan, J., Li, Y., Weng, J., Deng, R.H.: Leap: Leakage-abuse attack on efficiently deployable, efficiently searchable encryption with partially known dataset. In: CCS '21. p. 2307–2320. Association for Computing Machinery, New York, NY, USA (2021). https://doi.org/10.1145/3460120.3484540
12. Pouliot, D., Wright, C.V.: The shadow nemesis: Inference attacks on efficiently deployable, efficiently searchable encryption. In: CCS '16. p. 1341–1352. Association for Computing Machinery, New York, NY, USA (2016). https://doi.org/10.1145/2976749.2978401
13. Shapiro, D.: Github plaintextwikipedia (2020), https://github.com/daveshap/PlainTextWikipedia
14. Song, D.X., Wagner, D., Perrig, A.: Practical techniques for searches on encrypted data. In: Proceeding 2000 IEEE Symposium on Security and Privacy. S&P 2000. pp. 44–55 (2000). https://doi.org/10.1109/SECPRI.2000.848445
15. Wikimedia: Wikipedia dumps (2023), https://dumps.wikimedia.org/simplewiki/
16. Zhang, M., Shi, Z., Chen, H., Liang, K.: Inject less, recover more: Unlocking the potential of document recovery in injection attacks against sse (2024)
17. Zhang, X., Wang, W., Xu, P., Yang, L.T., Liang, K.: High recovery with fewer injections: practical binary volumetric injection attacks against dynamic searchable encryption. pp. 5953–5970 (2023). https://doi.org/arXiv preprint arXiv:2403.01155

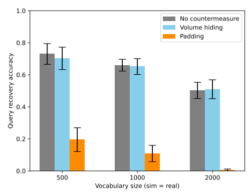
Appendix A



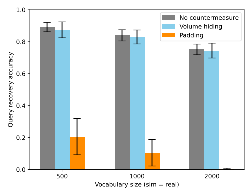
(a) Enron - RefScore



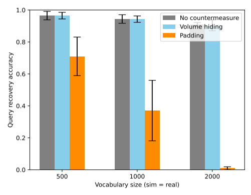
(b) Apache - RefScore



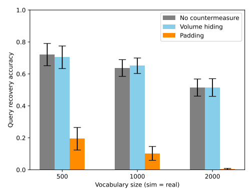
(c) Wiki - RefScore



(d) Enron - RefVolScore



(e) Apache - RefVolScore



(f) Wiki - RefVolScore

Fig. 11: Countermeasures comparison. $|D_{sim}| = 12K, |D_{real}| = 18K, m_{sim} = m_{real}, |Q| = 0.15 * m_{real}$, KnownQ = 15, RefSpeed = $0.05 * |Q|$, $n_{pad} = 500$.

Table 3: Summary of Notations

Notation	Description
D	Document collection $D = (d_1, d_2, ..., d_n)$
ED	Encrypted documents $ED = (ed_1, ed_2, ..., ed_n)$
D_{sim}	Similar document set, available to attacker
$\|D_{sim}\|$	Amount of similar documents
K_{sim}	Keyword vocabulary, extracted from D_{sim}
m_{sim}	Amount of keywords in similar vocabulary
D_{real}	Real document set, stored on server
$\|D_{real}\|$	Amount of real documents
K_{real}	Real keyword set which are usable by the client
m_{real}	Amount of keywords in real vocabulary
Q	Queries that are sent by the client and observed by attacker
$\|Q\|$	Amount of queries
KnownQ	Known queries (td, kw) pairs
—KnownQ—	Amount of known queries
UnknownQ	The unknown queries for the attacker, i.e. Q - KnownQ
C^s_{kw}	Co-occurrence submatrix for keywords
C^s_{td}	Co-occurrence submatrix for trapdoors
V^s_{kw}	Covolume submatrix for keywords
V^s_{td}	Covolume submatrix for trapdoors
K	Abbreviation for 1000, for example 1.2K = 1200
td	Abbreviation for trapdoor
kw	Abbreviation for keyword
kq_set	Set of keywords or queries
inv_index	Mapping from keywords to documents or query to document identifiers
vol_array	An array that consists of the volume of each document
nr_docs	Count of locally stored or server-estimated documents
RefSpeed	The refinement speed
MaxRefSpeed	The maximum refinement speed

Table 4: Comparison on Low Known Queries

Dataset	No. KnownQ	KnownQ Accuracy (%)	Newly Found KnownQ	Total KnownQ Accuracy (%)	RefScore Accuracy (%)	VolScore Accuracy (%)	RefVolScore Accuracy (%)	ClusterVolScore Accuracy (%)
Enron	2	100	2.15	73.85	11.39	1.55	21.39	20.07
	3	100	4.4	74.31	21.77	2.55	34.73	39.52
	4	100	5.1	87.77	35.21	5.45	54.76	57.84
Apache	2	100	2.75	81.44	27.47	3.34	52.64	61.59
	3	100	13.1	86.4	58.78	10.48	70.75	79.29
	4	100	23.2	92.95	75.55	16.82	89.97	90.07
Wikipedia	2	100	2.15	74.04	7.47	1.99	19.83	27.06
	3	100	2.75	80.51	28.81	2.14	33.98	37.11
	4	100	3.4	85.36	26.54	2.88	40.07	47.67

Table 5: Comparison on RefSpeed

Dataset	Attacks	RefSpeed	Mean Dynamic RefSpeed	Runtime (s)	Accuracy (%)
Enron	RefVolScore	2	N/A	40.93	81.03
	ClusterVolScore	2	3.18	25.35	80.89
	RefVolScore	4	N/A	20.97	84.08
	ClusterVolScore	4	3.07	25.62	83.42
	RefVolScore	8	N/A	11.39	66.88
	ClusterVolScore	8	3.18	25.93	75.14
Apache	RefVolScore	2	N/A	22.82	94.52
	ClusterVolScore	2	3.94	11.39	94.59
	RefVolScore	4	N/A	11.61	91.51
	ClusterVolScore	4	3.9	11.06	91.82
	RefVolScore	8	N/A	7.35	95.1
	ClusterVolScore	8	3.77	13.74	95.38
Wikipedia	RefVolScore	2	N/A	37.98	62.12
	ClusterVolScore	2	2.76	26.68	56.44
	RefVolScore	4	N/A	19.19	55.65
	ClusterVolScore	4	2.85	25.59	57.95
	RefVolScore	8	N/A	10.28	59.42
	ClusterVolScore	8	2.86	26.37	58.63

Algorithm 4 RefVolScore

```
1: Input: requirements $VolScore$, requirements $RefScore$
2: Output: results_ref_vol_score
3:
4: // Runs VolScore attack
5: results_vol_score ← $VolScore$
6: // Runs Refined Score attack
7: results_ref_score ← $RefScore$
8: // Retrieves the list of trapdoors from the {td : kw} predictions
9: tds1 ← get keys from results_vol_score
10: tds2 ← get keys from results_ref_score
11: // Intersecting trapdoors between the two results
12: intersect_tds ← intersect(tds1, tds2)
13: // Initialize new known queries
14: new_known_queries ← { }
15: // Find new known queries
16: for all $td \in$ intersect_tds do
17:     // Check if same keyword assigned and not used before
18:     if results_ref_score[td] == results_vol_score[td] and
19:             results_ref_score[td] not in new_known_queries and
20:             not in KnownQ then
21:         append ($td$ : results_ref_score[td]) to new_known_queries
22:     end if
23: end for
24: // Add the new known queries to the original known queries list
25: update(KnownQ, new_known_queries)
26: // Run fresh run of RefScore, but with new known queries
27: results_ref_vol_score ← $RefScore$
28: return results_ref_vol_score
```

Algorithm 5 Clustering

```
1: Input: requirements $RefScore$, max_ref_speed
2: Output: final_pred
3:
4: $final\_pred$ ← []
5: unknownQ ← $Q$
6: while unknown $Q \neq \emptyset$ do
7:     // Extract the remaining unknown queries
8:     unknownQ ← $\{td : (td \in Q) \wedge (\nexists kw \in K_{sim} : (td, kw) \in KnownQ)\}$
9:     $temp\_pred$ ← []
10:    // Propose a prediction for each unknown query
11:    for all $td \in$ unknownQ do
12:        $cand$ ← []                                   ▷ The candidates for trapdoor $td$
13:        for all $kw \in K_{sim}$ do
14:            $s = -\ln(||C^s_{kw}[kw] - C^s_{td}[td]||)$
15:            append {"kw": $kw$, "score":$s$} to $cand$
16:        end for
17:        Sort $cand$ in descending order according to the score.
18:        certainty ← score($cand$[0]) - score($cand$[1])
19:        append ($td$, kw($cand$[0]), certainty) to $temp\_pred$
20:    end for
21:    // Sort temp_pred on certainties in descending order, and call index_max_diff algorithm.
22:    // We add 1 for correct array splicing, and if index is 0 means new_ref_speed becomes 1.
23:    new_ref_speed ← index_max_diff($temp\_pred$, max_ref_speed) + 1
24:    // Either stop the algorithm or keep refining.
25:    if | unknownQ | < max_ref_speed then
26:        $final\_pred$ ← KnownQ ∪ $temp\_pred$
27:        unknownQ ← ∅
28:    else
29:        Append the new_ref_speed most certain predictions from $temp\_pred$ to KnownQ
30:        Add the columns corresponding to the new known queries to $C^s_{kw}$ and $C^s_{td}$
31:    end if
32: end while
33: return $final\_pred$
```