# High Recovery with Fewer Injections: Practical Binary Volumetric Injection Attacks against Dynamic Searchable Encryption

Xianglong Zhang and Wei Wang, *Huazhong University of Science and Technology;* Peng Xu, *Huazhong University of Science and Technology and Hubei Key Laboratory of Distributed System Security;* Laurence T. Yang, *Huazhong University of Science and Technology and St. Francis Xavier University;* Kaitai Liang, *Delft University of Technology*

https://www.usenix.org/conference/usenixsecurity23/presentation/zhang-xianglong

This paper is included in the Proceedings of the 32nd USENIX Security Symposium.

August 9–11, 2023 • Anaheim, CA, USA

978-1-939133-37-3

# High Recovery with Fewer Injections: Practical Binary Volumetric Injection Attacks against Dynamic Searchable Encryption

Xianglong Zhang[1], Wei Wang[1, ✉], Peng Xu[1,2, ✉], Laurence T. Yang[1,3], and Kaitai Liang[4]

[1]*Huazhong University of Science and Technology*
[2]*Hubei Key Laboratory of Distributed System Security, School of Cyber Science and Engineering*
[3]*St. Francis Xavier University*
[4]*Delft University of Technology*
✉*Corresponding authors: viviawangwei@hust.edu.cn, xupeng@hust.edu.cn*

## Abstract

Searchable symmetric encryption enables private queries over an encrypted database, but it can also result in information leakages. Adversaries can exploit these leakages to launch injection attacks (Zhang et al., USENIX Security'16) to recover the underlying keywords from queries. The performance of the existing injection attacks is strongly dependent on the amount of leaked information or injection. In this work, we propose two new injection attacks, namely BVA and BVMA, by leveraging a *binary volumetric* approach. We enable adversaries to inject fewer files than the existing volumetric attacks by using the known keywords and reveal the queries by observing the volume of the query results. Our attacks can thwart well-studied defenses (e.g., threshold countermeasure, padding) without exploiting the distribution of target queries and client databases. We evaluate the proposed attacks empirically in real-world datasets with practical queries. The results show that our attacks can obtain a high recovery rate ($> 80\%$) in the best-case scenario and a roughly 60% recovery even under a large-scale dataset with a small number of injections ($< 20$ files).

## 1 Introduction

Song et al. [42] proposed the notion of searchable symmetric encryption (SSE) that enables secure search over an encrypted database. Following the notion, researchers have presented various SSE schemes to balance practicability and security [6, 9, 21, 27, 36, 45]. In a standard SSE scheme, a search query is a series of interactions between a client and a server. The server can perform matching operations for the query, such as paring a search token given by the client with ciphertexts, and return the query results. Unfortunately, the statistics on the responses leak some patterns of the encrypted database. These patterns seem natural and harmless to data privacy, as they do not directly reveal the files' contents. Adversaries can still exploit them to recover the client's query. Leakage abuse attacks (LAA) [7, 25, 31, 34, 35, 39], a classic type of passive attacks on SSE, enable adversaries to observe the leakage patterns for a continuous query period and then combine the obtained information with prior knowledge to recover the target queries.

Unlike passive attacks, injection attacks [3, 7, 38, 51] enable adversaries to actively inject files into the encrypted database. An adversary can generate a certain number of injected files containing *known keywords* for the client who packs these encrypted files and sends them to the server. It can then recover the keywords with a high probability by observing the leakage patterns of the target queries. Surprisingly, many real-world applications are incapable of preventing file injection [38].

Cash et al. [7] introduced a seminal active attack against property-preserving encryption [1, 4], allowing the adversary to "implant" files into the client's database and then reconstruct partial plaintexts. Following a similar philosophy, Zhang et al. [51] proposed a concrete file injection attack (ZKP) to SSE. The attack enables the adversary to generate and inject files $F_1, \ldots, F_{\log n}$ by using $n$ known keywords, such that each keyword is in a unique subset of the files. After the client has made queries, the adversary must identify the exact set of the returned injected files so as to reveal the queries. The adversary *must* know the access injection pattern, i.e., the set of injected files matching the queries.

The SSE schemes protecting the access (injection) pattern, such as those built on ORAM [15, 19], can successfully resist ZKP [51] and other access pattern based attacks [7, 25, 34]. However, they still leak the *volume pattern (vp)* from the search results. We say that the *vp* includes (1) the number of response files, referred to as the *response length pattern (rlp)*, and (2) the word count of returned files, referred to as the *response size pattern (rsp)*. In this work, we will exploit these patterns in volumetric injection attacks (VIAs)[1].

Poddar et al. [38] proposed injection-based *multiple-round* and *single-round* attacks by only exploiting the *rlp*. In each round of the *multiple-round* attack, the adversary divides the candidate keywords into $k$ partitions and generates the same

[1]Note recent VIAs naturally leverage either the *rlp* or *rsp*.

Table 1: Comparison with leakage abuse (passive) and injection attacks. $\#W$ is the number of the known keywords. We use $k$ ($k \geq 2$) to represent the count of keyword partitions for the multiple-round attack, and $m$ ($m \geq 1$) refers to the injection constant of the single-round attack. We denote $\mathit{offset}, \gamma$ (generally, $\mathit{offset} \gg \#W$ and $\gamma \geq \#W/2$) as the basic size of the injected files for the decoding attack and BVA, respectively. The last column shows the minimum rounds required to restore all the observed queries. Other notions and definitions are given in Table 5, Appendix A.

| Attack | Leakage | Type | | Injection volume | | Round[2] |
|---|---|---|---|---|---|---|
| | | Passive | Injection | Length | Size | |
| IKK [25] | ap | ✓ | × | – | – | – |
| Count [7] | ap,rlp | ✓ | × | – | – | – |
| SelVolAn, Subgraph [3] | ap,vp | ✓ | × | – | – | – |
| LEAP [34] | ap | ✓ | × | – | – | – |
| SAP [35] | sp,rlp | ✓ | × | – | – | – |
| ZKP [51] | aip | × | ✓ | $O(\log \#W)$ | $O(\#W \log \#W)$ | 1 |
| Multiple-round[1][38] | rlp | × | ✓ | $O(k\#W \log_k \#W)$ | $O(k\#W^2)$ | $\#W \log_k \#W$ |
| Single-round [38] | rlp | × | ✓ | $O(m\#W)$ | $O(m\#W^2)$ | 1 |
| Decoding [3] | rsp | × | ✓ | $O(\#W)$ | $O(\mathit{offset} \cdot \#W^2)$ | 1 |
| Search[1][3] | rsp | × | ✓ | $O(\#W \log \#W)$ | $O(\#W^2)$ | $\#W \log \#W$ |
| BVA | rsp | × | ✓ | $O(\log \#W)$ | $O(\gamma\#W)$ | 1 |
| BVMA | vp, sp[3] | × | ✓ | $O(\log \#W)$ | $O(\#W \log \#W)$ | 1 |

[1] Unlike those schemes easily restoring multiple queries, search [3] and multiple-round [38] attacks can only recover a keyword at a time by running many attack rounds. This means that the client should make many queries, and the queries must include the target query at each attack round. The multiple-round is a strategy that depends on query replay, requiring the adversary to evoke the same query repeatedly by controlling the client. These two attacks commit more rounds and injected files (than others in the table) to recover multiple queries.

[2] We here say that an attack round consists of (1) an active and complete injection and then (2) an observation within a specific period. We will present a formal definition in Appendix C.

[3] BVMA mainly investigates the vp to recover the queries. Thus, the sp is an optional and non-essential leakage for the attack (see Appendix E).

number of empty files. For a keyword in the $i$th partition, the adversary adds the keyword to $i$ files as injection. If there are $i$ more response files to the target query than before (i.e., a previous round), the adversary can narrow the search space of the candidate keyword to the $i$th partition. The adversary must run $\log_k n$ rounds of injections and observations to recover a query, given $n$ known keywords. The attack works properly *as long as* the adversary can make the client constantly repeat the same query. This restriction works in some scenarios, e.g., websites using HTTP/1.1 RFC [16]. The *single-round* attack selects a specific $m$ and enables the adversary to inject $m \cdot n$ files for $n$ known keywords in which a keyword $w_i$ is included in $m \cdot i$ files. Upon observing a query with a response of $l$ files, the adversary can recover the query as $w_{\lfloor \frac{l}{m} \rfloor}$. The $m$ should be much larger than the number of the client's files containing $w_i$ so that $\lfloor \frac{l}{m} \rfloor$ is equal to $i$ and the adversary can recover the query. Compared to the multiple-round technique, this attack reduces the number of interactive rounds and recovers multiple queries. But its injection amount is still substantial.

Blackstone et al. [3] leveraged the *rsp* to propose a *decoding* attack and a *search* attack for multi-query and one query recovery, respectively. For each keyword $w_i$, the *decoding* attack generates and injects a file with a size of $i \cdot \mathit{offset}$. Given a query, if the difference of its response sizes before and after injection exceeds $i \cdot \mathit{offset}$ (i.e., observing $i \cdot \mathit{offset}$ more after injection), the adversary can infer that the underlying keyword is $w_i$. A drawback of this attack is that the adversary needs to consume a significant amount of resources in calculating *offset* and injections (which are linear w.r.t. the number of keywords) to perform well in query recovery. The *search* attack, recovering one keyword at a time, uses a binary search method. The adversary can inject a file containing half of the candidate keywords at an attack round. It can determine if the injected file is associated with the target query by a subsequent observation. Using this inject-and-observe approach in the following rounds, the adversary can halve the list of candidate keywords until there is only one keyword left. The attack can only recover a single keyword through massive file injections and attack rounds.

Existing VIAs are constrained by round complexity and injection amount. An interesting question thus arises:

*Could we propose practical injection attacks that achieve a high recovery rate with fewer injections and can also circumvent commonly used defenses?*

**Our contributions.** We present an affirmative answer to the above question by proposing two new injection attacks for dynamic SSE schemes and particular defense mechanisms (e.g., ORAM and padding). Our attacks leverage a dynamic binary injection approach that requires fewer injections than prior works. We show a comparison of the attacks in Table 1. The main contributions are summarized as follows.

• *Practical binary volumetric attacks.* We propose two practical injection attacks (see Section 3) that provide comparable performances to prior attacks, e.g., [3, 38]. First, we present the binary variable-parameter attack (BVA) by exploiting the *rsp*. We use a dynamic injection parameter $\gamma$ to balance a trade-off between injection size and recovery rate. Second, we develop the binary volumetric matching attack (BVMA), which is the first injection attack combining the leakage of the *rlp* and *rsp*. For any queries, the BVMA can filter incorrect keywords, with a small amount of injection, by observing the difference in the response volume before and after injection. Note that we can leverage other leakage information (e.g., query frequency) to enhance the recovery. We also present a generic method that can transform current VIAs ( [3, 38] and ours) to counter the threshold countermeasure (TC)[2] [51].

• *Comprehensive evaluations.* We compare our attacks with BKM [3], PWLP [38], and ZKP [51] in three real-world datasets (see Section 4). We generate queries by obtaining keyword trends from Google trend [22] and the Pageviews tool [33]. Experimental results show that our attacks can provide a comparable level of recovery rate (e.g., $> 80\%$ on average in Enron and Lucene) as the single-round (with $m = \#W$) and decoding attacks while requiring fewer injections (e.g., saving $> 99\%$ of injection costs given the keyword universe). Our attacks can also practically apply to large-scale datasets such as Wikipedia, with approximately 60% recovery.

We evaluate our attacks against various defenses (e.g., TC, padding) and client's active updates. Under TC, our attacks require relatively "lightweight" injections (e.g., $< 10^3$ files injected by the BVMA in Enron and Lucene). In contrast, the single-round and decoding attacks require a significant number of injections, with injection sizes respectively over $10^4\times$ and $10^6\times$ the cost of the BVMA (particularly in Wikipedia). We demonstrate that the static padding cannot effectively resist our attacks (with $> 60\%$ recovery rate on average against SEAL [15]). For dynamic padding (ShieldDB [46]), we show an optimization of our attacks that can yield a high recovery rate against ShieldDB. For example, it can maintain $> 80\%$ recovery by injecting around 600 files in Enron. We also demonstrate that our modified attack from BVA can perform well under client's active updates. Even if the client commits 100% updates for the Enron dataset, the attack can achieve $> 50\%$ recovery rate by increasing the injection size (e.g., $O(32 \cdot \#W)$) while remaining an $O(\log \#W)$ injection length.

## 2 Model Definitions

A dynamic SSE scheme (see Appendix B) should not directly leak any other information to adversaries except those that can be inferred from setup leakage $\mathcal{L}_{St}$, query leakage $\mathcal{L}_{Qr}$, and update leakage $\mathcal{L}_{Up}$, throughout interactions between the client and the server. It captures the adaptive security if adversaries can choose the target queries and the corresponding

[2]Note TC enables SSE schemes to constrain each (encrypted) file's word-count to a small threshold in order to defend against injection attacks.

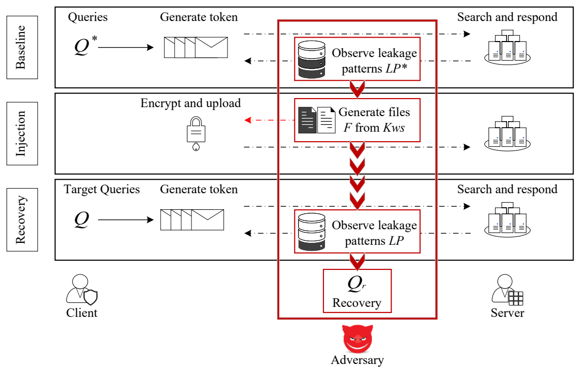


Figure 1: Attack model.

operations adaptively [11].

### 2.1 Leakage Model

The operations of a dynamic SSE scheme naturally yield multiple leakage patterns. We denote an adaptive semantic secure SSE scheme as $SSE = (Setup : \lambda \to D,\ Query : D \times Q \to RF,\ Update : D \times U \to \delta)$, in which $D$ is the encrypted database, $RF$ is the set of files matching the queries $Q$, $U$ is the set of updates containing $op$ and $(w, id)$, and $\delta$ represents the state of the client after the update from $U$. We define two types of patterns:

• the *access pattern* is the family of functions $ap : D \times Q^t \to R^t$ such that for a sequence of queries $Q = (q_1,\ \ldots,\ q_t)$, it outputs the response file identifiers $R = (ids(q_1),\ \ldots,\ ids(q_t))$.

• the *access injection pattern* is the family of functions $aip : D \times Q^t \to IR^t$ such that for a sequence of queries $Q = (q_1,\ \ldots,\ q_t)$, it outputs the response injected file identifiers $IR = (iids(q_1),\ \ldots,\ iids(q_t))$.

We note that most of LAA schemes [7, 25, 34] rely on the access pattern; while ZKP [51] exploits the access injection pattern for injection attack. We also formally define the search and volume patterns used in our attacks.

• the *search pattern* is the family of functions $sp : D \times Q^t \to M^{t\times t}$ where, for a sequence of queries $Q = (q_1,\ \ldots,\ q_t)$, it outputs a binary $t \times t$ matrix $M$ such that $M[i,\ j] = 1$ if $q_i = q_j$ and otherwise, $M[i,\ j] = 0$.

• the *response length pattern* is the family of functions $rlp : D \times Q^t \to RL^t$ where, for a sequence of queries $Q$, it outputs the number of the response files $RL = (\#D(q_1),\ \ldots, \#D(q_t))$.

• the *response size pattern* is the family of functions $rsp : D \times Q^t \to RS^t$ where, for a sequence of queries $Q$, it outputs the word count of the response files $RS = (\sum_{f \in D(q_1)} |f|_w, \ldots, \sum_{f \in D(q_t)} |f|_w)$.

## 2.2 Attack Model

To capture a general injection attack model, we enable the adversary to generate and inject files. We regard the adversary as an *honest-but-curious server* who follows the protocols but can still inject files into the client database[3]. We divide the entire attack process into three stages (see Figure 1).

(1) In the baseline phase, the adversary observes the client's query leakage $LP^*$ as pre-knowledge (provided the client sends queries to the server). This step is of importance for injection attacks (except for ZKP [51] and single-round attack [38]). This is because the adversary should obtain the correlation between the keywords and response files from the baseline's observations. The idea is to compare the volume difference between the response results before (baseline) and after injection for query recovery.
(2) In the injection, the adversary should carefully generate files $F$ by using its known keywords $W$ and the information obtained from the baseline. Next, the client encrypts the files and uploads them to the server.
(3) During the recovery phase, the adversary obtains the target queries' leakages $LP$ and recovers them by combining all the known information, namely $W$, $LP^*$, $F$, and $LP$.

We let $Q$ denote the target query set observed by the adversary and $Q_r$ denote the query recovery results. We refer to the recovery rate as $Rer : \#CorrectPred(Q_r)/\#Q$. We denote the number of injected files as $ILen : \#F$ and the word count of the injected files as $ISize : |F|_w$, where $F$ is the set of injected files. *The goal of injection attacks is to achieve a high Rer and minimize ILen and ISize by exploiting the known keyword universe and pre-injection leakage.*

## 3 Practical Volumetric Injection Attacks

Given the keyword universe $W$, VIAs can recover the client's queries on the encrypted database $D$ by observing the *vp* from the response results. A well-design and practical injection attack should limit the number and size of injected files. Our first attack, BVA, uses a dynamic parameter $\gamma$ to flexibly set the size of files to adjust the recovery rate. With a slight loss of recovery rate, it can significantly reduce the size of injection. Unlike the decoding attack, it does not need to calculate $\gamma$ fully and accurately, reducing the computational cost. To further optimize the injection size and boost the recovery rate in the worst-case scenario (i.e., $\gamma = \#W/2$), we propose the second attack called BVMA by *twisting* both the volume pattern and the search pattern. By carefully controlling the size of each injected file, the BVMA can ensure that each known keyword has a distinct injection volume. For any query, it can reveal the underlying keyword according to the difference of the response results before and after injection. We also provide a generic transformation that enables VIAs (e.g., [3, 38] and our attacks) to counter the TC.

[3]Or one may regard the adversary as an *active observer* who can generate injected files for the client and observe the query response.

**Algorithm 1:** BVA.

```
1  procedure Baseline($\widetilde{Q}$)
2      observe the response size $\widetilde{RS} = (\widetilde{rs}_1, ..., \widetilde{rs}_m)$ for query in
         $\widetilde{Q} = (\widetilde{q}_1, ..., \widetilde{q}_m)$;
3      return $\widetilde{RS}$;
4  procedure Injection($W$)
5      $F \leftarrow \emptyset$;
6      select an injection parameter $\gamma = \{\gamma \in \mathbb{N} \cap \gamma \geq \#W/2\}$;
7      for $i = 1 \rightarrow \log \lceil \#W \rceil$ do
8          generate the files $f_i$ that contains the keywords $w \in W$
             whose $i$th bit is 1;
9          pad $f_i$ until its $size = \gamma \cdot 2^{i-1}$;
10         $F = F \cup f_i$;
11     return $F$;
12 procedure Recovery($Q$)
13     initialize an empty set $Q_r$;
14     observe the response size that $RS = (rs_1, ..., rs_n)$ for target
         query in $Q = (q_1, ..., q_n)$;
15     for $i = 1 \rightarrow \#RS$ do
16         find $u \in \#W$ satisfying that $rs_i - u \cdot \gamma = \widetilde{rs}_j$ for some
             $\widetilde{rs}_j \in \widetilde{RS}$;
17         add $w_u$ to $Q_r$;
18     return $Q_r$;
```

### 3.1 Binary Variable-Parameter Attack

The BVA (see Algorithm 1) works as follows. In the baseline phase, the adversary observes and records the response sizes of unknown queries. During the injection phase, it generates and injects files according to the injection parameter $\gamma$ in a *binary* way. In the last phase, the adversary observes an additional sequence of the client's queries $Q = (q_1, ..., q_m)$ as the attack targets with response size $RS = (rs_1, ..., rs_m)$. It aims to recover all the queries in $Q$.

*$BVA - Baseline$.* The adversary observes the response size $\widetilde{RS}$ for a sequence of queries $\widetilde{Q}$ (line 2).

*$BVA - Injection$.* The adversary identifies the keyword universe $W$ with the set $\{0, ..., \#W - 1\}$ and indicates the injected files with the set $F = \{f_1, ..., f_{\lceil \log \#W \rceil}\}$. First, the adversary selects the injection parameter $\gamma$ satisfying $\gamma \in \mathbb{N}$ and $\gamma \geq \#W/2$ to ensure each file can accommodate half of the keywords (line 6). Second, each file $f_i$ has a size of $\gamma \cdot 2^{i-1}$ and contains keywords whose $i$th bit is equal to 1 so that when $w_i$ is queried, the total response size of injected files is $\gamma \cdot i$ (lines 7-10).

Note an example (see Figure 11, Appendix D) shows the differences between the decoding attack and ours in terms of injection length and size.

*$BVA - Recovery$.* The adversary observes $RS = (rs_1, ..., rs_n)$ again for the target queries $Q = (q_1, ..., q_n)$. For a query $q_i \in Q$ whose response size is $rs_i \in RS$, it traverses $\widetilde{RS}$ (obtained from the *baseline* phase) to get a $u \in [\#W]$, satisfying the condition:

$rs_i - u \cdot \gamma = \widetilde{rs}_j$. The adversary then recovers the query $q_i$ with $w_u$ (lines 15-17).

Let $P$ be the probability distribution over the keyword universe and $P(w_u)$ be the probability that the client will query the keyword $w_u$. We use $\widetilde{rs}_{w_u}$ to represent the response size of $w_u$ observed in the *baseline* phase. We formalize the probability of incorrect recovery as follows.

**Claim 1.** *For any query $q_{w_i}$ and $\gamma \geq \#W/2$, the probability that $BVA-Recovery(q_{w_i})$ outputs an incorrect $w_u$ is*

$$Pr(w_u \neq w_i) = \sum_{\substack{u \in [\#W], u \neq i, \\ rs_{w_i} = \widetilde{rs}_{w_u} + u * \gamma}} P(w_u)$$

*Proof.* Consider the client made a query $q_{w_u}$ with probability $P(w_u)$, and its response size $\widetilde{rs}_{w_u}$ was observed in the baseline. In the recovery phase, the adversary observed that the target query $q_{w_i}$'s response size is $rs_{w_i}$ after injection. We then have three cases. (1) $rs_{w_i} < \widetilde{rs}_{w_u}$. The response size (after injection) must be $> \widetilde{rs}_{w_u}$ (before injection). In this case, the adversary will not guess any keywords. (2) $rs_{w_i} \geq \widetilde{rs}_{w_u}$ but $\gamma \nmid rs_{w_i} - \widetilde{rs}_{w_u}$. The adversary will not output any guess according to the recovery algorithm. (3) $rs_{w_i} \geq \widetilde{rs}_{w_u}$ and $\gamma \mid rs_{w_i} - \widetilde{rs}_{w_u}$. The adversary will guess an incorrect $w_u$ so that $u = (rs_{w_i} - \widetilde{rs}_{w_u})/\gamma$ if $u \neq i$; otherwise, it will reveal the correct keyword. ■

We see that only the third case can cause an incorrect recovery, which implies that the recovery rate depends on the selection of $\gamma$. A well-selected $\gamma$ can greatly reduce the occurrence of the third case and make the adversary achieve the same recovery rate as the decoding attack. Even in the worst case (i.e., $\gamma = \#W/2$), the recovery rate can still maintain $> 70\%$ in Enron and Lucene. We further state that the binary injection approach can restrict the injection length to $\log \#W$, which outperforms prior schemes [3, 38]. Note more details will be given in the experiments (see Section 4).

**Claim 2.** *For the keyword universe $W$ and the injection parameter $\gamma \geq \#W/2$, the total injection size incurred by $BVA-Injection(W)$ is $O(\gamma \#W)$.*

*Proof.* Through the binary injection, the BVA requires $\log \#W$ files in total, and the size of each file is $\gamma \cdot 2^{i-1}$ for $i \in [\log \#W]$. Therefore, the total injection size during this phase is $\gamma \cdot (2^0 + 2^1 + \ldots + 2^{\log \#W - 1}) = O(\gamma \#W)$. ■

The attack can recover multiple queries with a high recovery rate (e.g., averagely 80% in Enron) and a small injection volume by leveraging the *rsp*, even the file contents and co-occurrences of keywords are hidden. It does not require any knowledge of query and file distribution. The BVA is a more practical VIA compared to prior attacks.

## 3.2 Binary Volumetric Matching Attack

Unlike the decoding attack relying on *offset*, the BVA adjusts a proper $\gamma$ for different datasets. This could affect the recovery rate and injection size. For example, setting a small $\gamma$ can decrease the injection size (compared with decoding) but could fail to distinguish some queries. To refine the attack performance, we introduce the BVMA, which is completely independent of any *offset* or $\gamma$. The BVMA is the first injection attack that *combines* the response length and size patterns (as well as the *sp*). The combination can accurately filter known candidate keywords with a smaller injection size. See the BVMA in Algorithm 2.

**Algorithm 2:** BVMA.

```
1  procedure Baseline($\widetilde{Q}$)
2      observe and record the baseline response size $\widetilde{RS}$, the response
         length $\widetilde{RL}$, and the frequency $\widetilde{Freq}$ for query in
         $\widetilde{Q} = (\widetilde{q}_1, \ldots, \widetilde{q}_m)$;
3      return $(\widetilde{RS}, \widetilde{RL}, \widetilde{Freq})$;
4  procedure Injection($W$)
5      $F \leftarrow \emptyset$;
6      for $i = 1 \rightarrow \log \lceil \#W \rceil$ do
7          generate the file $f_i$ containing the keywords $w$ in $W$ whose
             $i$th bit is 1;
8          pad $F_i$ until its $size = 2^{i-1} + \#W/2$;
9          $F = F \cup f_i$;
10     return $F$;
11 procedure Recovery($Q$)
12     initialize an empty set $Q_r$;
13     gather the new observed response size $RS$, response length $RL$,
         and frequency $Freq$ for victim's target queries $Q$;
14     for $i = 1 \rightarrow \#Q$ do
15         $CA \leftarrow \emptyset$;
16         for $\widetilde{rs}_j \in \widetilde{RS}, u \in [\#W]$ do
17             add $(j, w_u)$ to $CA$;
18             if $rs_i - \#file_u \cdot \#W/2 - u \neq \widetilde{rs}_j$ then
19                 remove $(j, w_u)$ from $CA$;
20             if $rl_i \neq \#file_u + \widetilde{rl}_j$ then
21                 remove $(j, w_u)$ from $CA$;
22         find the minimum value of $|freq_i - \widetilde{freq}_j|$ for
             $(j, w_u) \in CA$ ;                 // non essential step
23         add $w_u$ to $Q_r$;
24     return $Q_r$;
```

*$BVMA - Baseline$.* This is similar to the BVA-Baseline; but we get queries' response length $\widetilde{RL}$, response size $\widetilde{RS}$, and frequency from the *sp* after a period of observation (line 2).

*$BVMA - Injection$.* We inject files in a binary manner. Different from the BVA (where the size of injected files relies on the $\gamma$), we set the size of file $f_i$ with $size = 2^{i-1} + \#W/2$ for $i \in [\log \#W]$ containing the keywords whose $i$th bit is 1 (lines 6-9). For the keyword $w_u \in W$, we then record its injection length as $\#file_u$ and injection size as equal to $u + \#file_u \cdot \#W/2$. Each keyword has a unique pair of injection size and length and thus we can distinguish different keywords with a relatively high probability.

*$BVMA - Recovery$.* We combine the leakage pattern $LP \in \{rlp, rsp, sp\}$ to filter candidate keywords. For a target query $q_i \in Q$, we generate a candidate set $CA$ by adding all $(j, w_u)$ to it for $\forall j \in [\#\widetilde{RS}], u \in [\#W]$ (line 13). In the beginning, we exploit the *rsp* to remove $(j, w_u)$ from $CA$ such that the pair

does not satisfy $rs_i - \#file_u \cdot \#W/2 - u = \widetilde{rs}_j$ (lines 18-19). This condition means that $\widetilde{rs}_j$ and $w_u$ are not the original response size and underlying keyword of $q_i$, respectively. In the second filtering stage, we exclude those candidate keywords that dissatisfy $rl_i = \#file_u + \widetilde{rl}_j$ for $w_u \in CA$ by exploiting the *rlp* (lines 20-21). If the condition does not hold, we confirm that the $\widetilde{rl}_j$ and $w_u$ are not the response length of $q_i$ before injection and the correct recovery keyword, respectively. We then remove the pair $(j, w_u)$ from the candidate set. Finally, we can use the *sp* to identify the closest frequency of the queries between the *baseline* and *recovery* phases from the remaining candidate keywords of *CA* (line 22). We note that the *sp* moderately improves the recovery rate. Even without using it, the BVMA can still achieve about 80% recovery, e.g., in Enron (see Appendix E).

**Claim 3.** *For any target query $q_{w_i}$, the probability that $BVMA-Recovery(q_{w_i})$ outputs an incorrect $w_u$ is*

$$Pr(w_u \neq w_i) \leq \sum_{\substack{u \in [\#W], u \neq i, \\ rs_{w_i} - \widetilde{rs}_{w_u} = u + \#file_u \cdot \#W/2, \\ rl_{w_i} - \widetilde{rl}_{w_u} = \#file_u}} P(w_u)$$

*Proof.* Assume that the adversary observed the response size $\widetilde{rs}_{w_u}$ and length $\widetilde{rl}_{w_u}$ of a query $q_{w_u}$ with probability $P(w_u)$ during the baseline. In the recovery phase, the adversary observed the target query $q_i$, whose response length is $rl_{w_i}$ and response size is $rs_{w_i}$ (after injection). If it determines $w_u$ as the underlying keyword, that means the difference of the *rsp* before ($\widetilde{rs}_{w_u}$) and after ($rs_{w_i}$) injection is equal to the total injection size ($u + \#file_u \cdot \#W/2$) and the difference on *rlp* before and after injection is the injection length ($\#file_u$). The adversary chooses $w_u$ as an incorrect recovery only when the above two conditions (for response length and size) and $u \neq i$ are satisfied. The probability is no more than $\sum P(w_u)$ as we can use other leakages (e.g., the frequency exploited by using the *sp*) to further eliminate those incorrect keywords. ■

We also provide a claim and its proof for the injection size.

**Claim 4.** *For the keyword universe $W$, the total size output by the $BVMA-Injection(W)$ is $O(\#W \log \#W)$.*

*Proof.* The attack requires $\log \#W$ files with size of $2^{i-1} + \#W/2$ for $i \in [\log \#W]$. Thus, the total injection size is $(2^0 + 2^1 + \ldots + 2^{\log \#W - 1}) + \#W \cdot \log \#W/2 = O(\#W \log \#W)$. ■

We see that the BVMA inherits the advantages of the BVA (except the usage of $\gamma$) and provides an improvement on injection size, for example, the BVMA outperforms the best case's BVA in Enron dataset (i.e., when $\gamma = \#W/2$, see Figure 5).

## 3.3 Attacks against Threshold Countermeasure

Threshold countermeasure (TC) [51] can prevent large-size files (e.g., #word > the threshold $T$) from being injected into the database. The $T$ could be set relatively small in practice

**Algorithm 3:** Injection Attacks against TC.

```
procedure Injection_shard($T$, $Basic\_F$)
    $Shard\_F \leftarrow \emptyset$;
    // Note $|f|_w$ is the word count of file $f$
    for $f \in Basic\_F$ do
        if $|f|_w > T$ then
            // Cutting
            extract keyword set $C\_W$ from $f$;
            cut $f$ into $\lceil \#C\_W/T \rceil$ smaller files $Cut\_F$ with each
              size of $T$, and each contains different keywords
              from $C\_W$;
            add $Cut\_F$ to $Shard\_F$;
            // Refilling.
            for $c\_f \in Cut\_F$ do
                generate $\lceil |f|_w/T \rceil - 2$ files $Ref\_F_1$ with each
                  size of $T$ and each containing all keywords
                  from $c\_f$;
                generate as fewer files $Ref\_F_2$ as possible
                  containing all keywords from $c\_f$, and each
                  size is $|f|_w - (\lceil |f|_w/T \rceil - 1) \cdot T$ ;
                add $Ref\_F_1$ and $Ref\_F_2$ to $Shard\_F$;
        else
            add $f$ to $Shard\_F$;
    return $Shard\_F$;
```

to counter injection attacks, without seriously affecting the functionality of dynamic SSE. For example, only 3% of the files in Enron ($\#W = 3{,}000$) contain more than 500 words. If employing TC with $T = 500$, we could skip these 3% of files from indexing. Under this setting, a dynamic SSE can effectively resist prior attacks (e.g., [3, 38] and ours). This is because each injected file must contain a considerably large number of words (e.g., $\geq \#W/2$ words), in particular, the single-round [38] and decoding [3] attacks force an injected file to contain $O(m\#W)$ and $O(offset \cdot \#W)$ words.

We design a generic transformation method to "protect" VIAs from TC (see Algorithm 3). The transformation takes the threshold $T$ and the set of large files *Basic_F* as input and runs the *cutting* and *refilling*, where *Basic_F* is the output of a concrete VIA algorithm, e.g., BVA. In Algorithm 3, we cut large files (with #word>$T$) into smaller ones to satisfy the threshold. To eliminate the impact of file cutting on recovery rate, we keep the injection length $\#file_u$ or size $|file_u|_w$ of each keyword $w_u \in W$ consistent before and after Algorithm 3. For example, in the single-round attack [38], for any keyword $w_u$, we keep its injection length $\#file_u$ unchanged (after cutting) to avoid bringing any impact to the recovery phase. In the decoding [3] and our attacks, we maintain a consistent injection size for the keywords (by refilling).

During the *cutting* phase (lines 7-9), for each $f \in Basic\_F$, we extract all keywords $C\_W$ from $f$. We then generate $\lceil \#C\_W/T \rceil$ files for *Cut_F* by following 1) the size (word count) of each file is $T$; and 2) each file contains up to $T$ distinct (non-overlapped) keywords from $C\_W$ so that each keyword $w_u$'s injection length $\#file_u$ is 1 (note the number remains the same before and after cutting). We finally have sets of small-size files to bypass TC while maintaining the consistency of the keywords' injection length.

In the *refilling* phase (lines 11-14), for each file $c\_f \in Cut\_F$, we generate two file sets $Ref\_F_1$ and $Ref\_F_2$. $Ref\_F_1$ is constructed by refilling $|f|_w/T-2$ files, in which each file is with size of $T$ and it contains all the keywords in $c\_f$. Next, we generate as few files as possible (for $Ref\_F_2$) to accommodate all the keywords in $c\_f$, with each file having a size of $|f|_w - (\lceil |f|_w/T \rceil - 1)\cdot T$. We ensure that the injection size of keywords in $c\_f$ is still $|f|_w$ after the refilling.

**Claim 5.** *For a large-size file set $Basic\_F$ output by $Injection(W)$ and a threshold $T$, the injection size of each keyword $w_u$ output by the algorithm $Injection\_shard(T, Basic\_F)$ is the same as that of $Injection(W)$, where $Injection(W)$ is the injection phase of a VIA (e.g., [3, 38], BVA, BVMA).*

*Proof.* During the *cutting* phase, for a file $f$ from the set $Basic\_F$, we produce multiple small files $Cut\_F$ containing different keywords with size $T$, so the injection size of each keyword is $T$. In the $refilling$ phase, we generate $\lceil |f|_w/T \rceil - 2$ files, in which each file is with size $T$ and contains all the keywords in a small file $c\_f \in Cut\_F$. For each keyword $w_u$, we generate another file with size of $|f|_w - (\lceil |f|_w/T \rceil - 1)\cdot T$ to include the keyword. After the above steps, the injection size of each keyword is $size = T + (\lceil |f|_w/T \rceil - 2)\cdot T + |f|_w - (\lceil |f|_w/T \rceil - 1)\cdot T = |f|_w$, which is the same as the amount before the transformation. ■

We say that the injection length strongly relies on the word count of each file $f$ and the number of files in $Basic\_F$. Our attacks, requiring fewer injected files than others, can naturally provide a practical performance against TC. This is proved by the experiments (see Figure 7). For example, in Enron with $\#W = 3,000$, setting $T = 500$, $\gamma = \#W/2$ for the BVA, and $m = \#W/2$ for the single-round attack, we see that the lower bounds of the injected files for our attacks are approximately $10^5$ and $10^3$ which are at least $10^2\times$ less than that of the single-round ($10^7$) and decoding ($10^9$) attacks.

## 4 Evaluation

We compared our attacks with the multiple-round and single-round attacks [38], the search and decoding attacks [3], and ZKP [51], under various metrics in the real-world datasets. We also evaluated our attacks against well-studied defenses (e.g., TC, padding) and client's active update. We used Python 3.5 to implement the experiments and run the codes in Ubuntu 16.04 of 64-bit mode with 16 cores of an Intel(R) Xeon(R) Gold 5120 CPU(2.20GHz) and 64 GB RAM. Our codes are publicly available in https://github.com/Kskfte/BVA-BVMA.

### 4.1 Experimental Setup

**Datasets.** We used three real-world datasets with different scales. The first one is the Enron email corpus [48] between 2000-2002, which contains 30,109 emails. The second one is the Lucene mailing list between 2001-2020, with about 113,201 emails from Apache Foundation [17]. The last dataset is from Wikipedia [18]. We extracted the contents of Wikipedia (in 2020) into a subset with 6,154,345 files by using an extraction algorithm in [41]. We also applied Python's NLTK corpus [44] to obtain a list of all English words without stopwords and then selected the most frequent words to build the keywords set. We set the total extracted keyword universe $W$ for Enron, Lucene, and Wikipedia as 3,000, 5,000, and 100,000, respectively.

Table 2: Descriptions on datasets

| | Enron | Lucene | Wikipedia |
|---|---|---|---|
| #Keyword | 3,000 | 5,000 | 100,000 |
| #File | 30,109 | 113,201 | 6,154,345 |
| QI | GTrend [22] | GTrend | Pageview [33] |
| Coverage | 260 weeks | 260 weeks | 75 months |

We used Google Trends [22] of 260 weeks trends between October 2016 and October 2021 to simulate the query trends for Enron and Lucene. We applied the Pageviews, Toolforge [33] containing 75 months of page views from July 2015 to September 2021 to generate query trends for Wikipedia. We assumed that the client performs 1,000 queries weekly for Enron (Lucene) and 5,000 queries monthly for Wikipedia. We regarded the client's queries (within ten weeks/months) as the target in the recovery phase. The dataset information is given in Table 2, where QI represents the source of query trends, and "coverage" is the time interval of QI.

In the experiments, we made 30 runs for each test and further output the average result. We measured the query recovery rate for the compared attacks, in which the rate represents how much percentage of the client's queries the attacks could recover correctly.

**Keyword leakages.** We used *keywords (Kws) leakage percentage* to measure the prior knowledge of the adversary. We say that the keyword universe could be easily and partially obtained for several reasons: (1) a tiny amount of files could contain a large number of keywords; (2) a public database known by the adversary, which shares similar distribution with the target database, could contain partial target keywords [12]; (3) some expired or spam emails could leak keywords. To test our statements, we present the number of known keywords with file leakage in Figure 2. We assume that Enron is the client database and Lucene (rather than Wikipedia) is the public database known by the adversary. We state that Enron and Lucene are both email datasets and thus, they have similar keyword distributions; and also, they share similar keyword universes. In Enron, 0.5% leakage files can lead to the exposure of half of the keyword universe; while obtaining 10% of the leakage files, the adversary can reveal the entire universe. Given Lucene as the similar database, the adversary easily reveals $> 80\%$ keywords from Enron.

**Observation periods.** The observation periods may vary depending on the datasets. For Enron and Lucene, which contain several thousand keywords, we only used a few weeks to observe queries to obtain the statistical information of most keywords. We spent more time (dozens of months) on Wikipedia than others to collect sufficient leakage information. To help the reader to understand the relationship between the occurrence of new queries and observation period, we show some examples in Figure 3. We observed 1,000 queries per week for Enron and Lucene, and 5,000 per month for Wikipedia. The number of new queries in Figure 3 decreases dramatically with the extension of observation period under different datasets and query distributions. In particular, there are $< 10\%$ new queries per week for Enron (resp. Lucene) after the observation lasts $> 8$ (resp. 16) weeks. We say that 8 (resp. 16) weeks are sufficient for the adversary to observe queries leakage from these datasets. Even for Wikipedia, a 32-month observation is practical to obtain statistics information. Note we also see that the occurrence probability of new queries performs similarly under different query distributions (i.e., real-world and uniform). We set the observation period to 8 weeks, 16 weeks and 32 months respectively for Enron, Lucene and Wikipedia. We further randomly selected the start period for the observation from the coverage given in Table 2.

## 4.2 Comparison with Other Attacks

We compared our attacks with prior injection attacks in three real-world datasets. Note we do not consider padding strategies in this section but will test the attacks against them later. We tested the performance of the BVA and decoding attack under different parameters since they exploit the *rsp* with similar recovery approaches. We also performed empirically evaluations on the BVA, the BVMA and other attacks. Note that we did not put ZKP into further comparisons (except in Figure 7 against TC). The crucial difference between our attacks and ZKP is on the leakage pattern. Our attacks exploit the *vp* while ZKP focuses on the *aip*. There are both pros and cons for the attacking strategies. For example, our attacks could bypass ORAM (around 60% against both static padding and ORAM on average, see Figure 8), while ZKP cannot; but ZKP requires less injection size than ours (see Figure 7).

**Injection Parameter for BVA and Decoding.** Recall that the BVA uses $\gamma$ to flexibly control the injection volume. We evaluated the recovery rate and injection size while $\gamma$ varies from $\#W/2$ to the *offset* (see Figure 4). Like the setting in the decoding attack [3], we assume that all queried keywords fall in the adversary's known keyword universe. As the $\gamma$ increases, the recovery rate of the BVA abruptly rises to the maximum (providing the same recovery as the decoding attack) and meanwhile, the injection size performs steady. One can observe that even *in the worst case*, where $\gamma = \#W/2$, the BVA can still achieve practical recovery rates ($> 70\%$ in

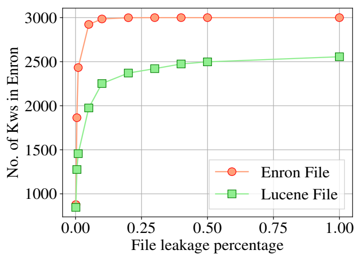


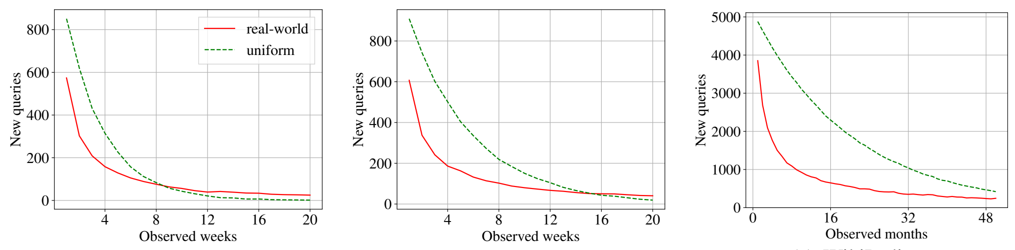
Figure 2: The no. of Enron keywords corresponding to the file leakage. We refer Enron as the client database and Lucene as the public database with similar distributions.

Enron and Lucene, $> 60\%$ in the large-scale Wikipedia). It shows 20% decline on average in Enron and Lucene (note 28% in Wikipedia) at the recovery in this case, as compared to the decoding attack. But its injection size is approximately five orders of magnitude less than that of the decoding attack. We also see that setting $\gamma = \mathit{offset}/4$ can sufficiently ensure the BVA to achieve the similar recovery rate ($< 5\%$ gap) as the decoding attack. A further increase on $\gamma$ could not produce significant improvement on the recovery rate. Based on the results, we confirm that a small but reasonable $\gamma$ (e.g., $\gamma = \#W/2$) can guarantee a practical recovery rate ($> 60\%$) with a relatively small injection size (around $10^8$ in Enron and Lucene, $10^{11}$ in Wikipedia) as compared to the decoding attack ($> 10^{12}$ in Enron and Lucene, $10^{17}$ in Wikipedia).

**Overall Comparison with Decoding and Single-round.** We compared our attacks with the single-round [38] and decoding [3] attacks. We tested the recovery rate *Rer*, the injection length *ILen*, and the injection size *ISize* under different keyword leakage ratios in the fixed observation period. We evaluated the BVA with error bars by varying $\gamma \in [\#W/2, \mathit{offset}/4]$. We simulated the single-round attack with two specific cases where $m = 1$ and $m = \#W$. The recovery rate is presented in Figure 5 (and Figure 13, 14 in Appendix G) and the running time is listed in Table 6, Appendix G.

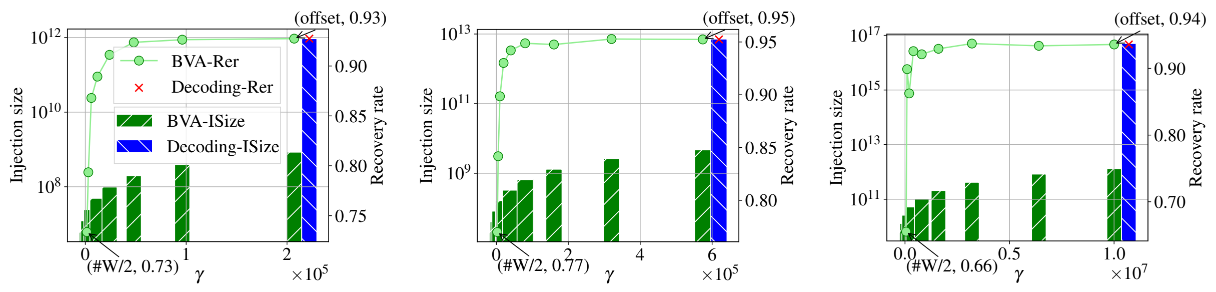
Table 6 shows the time cost of restoring $10 \times 1{,}000$ queries in Enron Dataset (provided that the adversary knows the keyword universe). Most of the attacks take $< 3s$, while the BVMA is slower than others (about $19s$). This is because it merges multiple leakages for keyword filtering, which naturally increases the time complexity.

Figure 5 illustrates that as the number of leaked keywords increases, *Rer*, *ILen* and *ISize* present an upward trend. More concretely, *Rer* delivers a quasi-linear growth, while the incline of *ILen* and *ISize* gradually flattens. From the recovery rate, we see that the single-round attack with $m = 1$ can only restore $< 1\%$ queries, which performs the worst. The BVA and BVMA share a small gap (e.g., $< 10\%$ on average in Enron) with the decoding and the single-round ($m = \#W$) attacks. We also notice that the BVMA surpasses the minimum recovery rate of the BVA (see the error bar in Figure 5(a)) by



(a) Enron (b) Lucene (c) WikiPedia

Figure 3: The occurrence probability of new queries as the observation period increases.



(a) Enron (b) Lucene (c) WikiPedia

Figure 4: Comparison between the decoding attack and BVA under γ.

Table 3: *offset* with different sizes of keyword universe. Assume the adversary can observe the response of all keywords.

| $\#W$ | 30 | 300 | 1,000 | 3,000 | 100,000 |
|---|---|---|---|---|---|
| *offset* | 157 | 6,615 | 48,447 | 207,015 | $> 10^7$ |

almost 10% when the keyword leakage reaches 100%. This is reasonable as the BVMA combines different leakage patterns (e.g., *rlp* and *rsp*) to filter candidate keywords, which can bring advantage on recovery rate.

The results of *ILen* (see Figure 5(b)) show that both BVA and BVMA (only injecting $< 20$ files) require much fewer injections (at least $50\times$) than the decoding attack. The single-round attack with $m = 1$ takes the same injection length as the decoding attack but with a poor recovery rate (see Figure 5(a)). We note one may set $m = \#W$ to produce a practical recovery in the single-round attack. But this requires a massive amount of injected files ($> 10^6$ files).

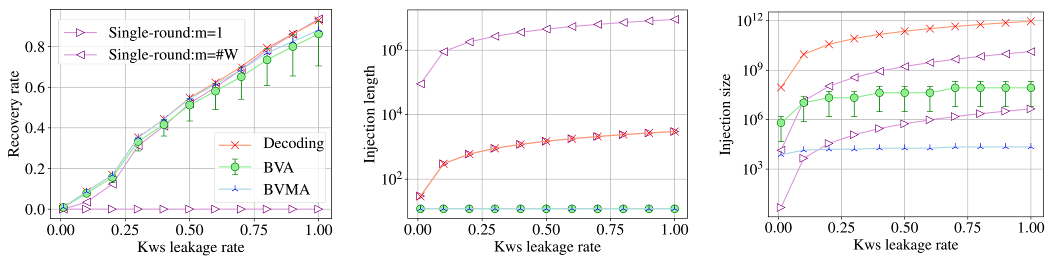
As for the metric *ISize*, the decoding attack gives the worst performance. This is because the attack strongly relies on *offset*. Table 3 illustrates that *offset* grows abruptly with the expansion of keyword universe. When $\#W = 3,000$, each injected file $f_i$ must contain $207,015 \times i$ words (where $i \in [\#W]$), which is extremely impractical in reality. Similarly, the performance of the single-round attack is restricted by $m$. To achieve a $> 80\%$ recovery, the attack (with $m = \#W$) requires a relatively large injection size, around $10^{10}$ in Enron.

Fortunately, the BVA and BVMA are independent of the *offset* (and $m$) and require fewer injections. For example, in Enron (resp. Lucene) with 3,000 (resp. 5,000) keywords (see Figure 5, 13), they achieve $> 80\%$ recovery rate on average by only taking nearly $10^8$ and $10^4$ injection size, respectively. The costs are multiple orders of magnitude less than those of the decoding attack ($10^{12}$) and single-round attack with $m = \#W$ ($10^{10}$). In Wikipedia (see Figure 14) including 100,000 keywords, the injections of the BVA and BVMA are only $10^{10}$ and $10^6$ with approx. 60% recovery rate. To maintain the same level of recovery, the single-round and decoding attacks must take respectively $10^4\times$ and $10^6\times$ costs.

We conclude that our attacks can provide: 1) practical recovery rate, and 2) much fewer injections (length and size) than other attacks. Our attacks (the BVA in particular) can be also applicable to the large-scale dataset (see Appendix G).

**Comparison for Single Query.** The multiple-round [38] and search attacks [3] only recover one query at a time. In Figure 6, we show how the average injection size (per query) varies with the increasing number of target queries. We did not consider the recovery rate in the experiments, because the performances of the BVA and BVMA with a single query are comparable to those shown in Figure 5(a) (and Figure 13(a), 14(a) in Appendix G). We note the multiple-round and search attacks can provide 100% recovery rate by taking sufficiently large (e.g., $\#W \log \#W$) injection volumes and attack rounds.

With the increase in the target queries, the multiple-round and search attacks give constant straight lines on injection size, while our attacks provide a sharp decline. The cause of



(a) Recovery accuracy (b) Injection length (c) Injection size

Figure 5: Comparison on the recovery rate, injection length, and injection size with different keywords (Kws) leakages in **Enron**.

the drop is that we can reuse injections on the previous queries to recover the following queries. At the beginning, when no. of queries is 10, the cost of the BVMA (e.g., slightly $> 10^4$) is close to that of the multiple-round and search attacks. After 100 queries, the BVMA outperforms others. In Enron and Lucene, the BVA yields the similar results as the multiple-round and search attacks, around $10^4$, when no. of queries is up to $2,000$. But its cost is roughly $10^2\times$ larger than that of the BVMA. Our attacks do show a noticeable advantage on injection size with the increase of query number.

**Comparison against TC.** To circumvent TC, we proposed a transformation Algorithm 3 (see Section 3.3). We note there is also a variant for the ZKP [51] that can counter TC. We evaluated the number of injected files caused by the Algorithm 3 and the ZKP variant with different thresholds $T$ (see Figure 7). We limited $T$ to be no more than the total number of keywords (i.e., $\#W$) in the experiments, because 1) when $T$ reaches to $\#W$, the number of injected files approaches to stable, in particular, for the ZKP and BVMA; and 2) the word count of a file (in a real-world dataset) is normally less than the total number of keywords.

In Figure 7, increasing $T$ leads to the decline of the injection length. The ZKP variant requires the least number of injection. This is because it uses a unique approach (instead of just the volume information) to recognize the injected files. In contrast, other attacks leverage injections to increase the differences in the volume of each keyword. There is a small gap on injection between the BVMA and the ZKP variant. The injection amount of the former drops fast and ultimately approaches to that of the latter. Given a small and reasonable $T$ (e.g.,$T = 500$), the injection length of the BVMA is around $10^3$ in Enron and Lucene, which is still practical. Under a large threshold (e.g., $T = 2,000$), the single-round and decoding attacks take $>10^6$ injected files in Enron and Lucene which are at least $10^3\times$ and $10^5\times$ larger than our costs, respectively. In Wikipedia (under $T > 25,000$), they require at least $10^4\times$ and $10^7\times$ more injections than ours.

Our attacks (especially the BVMA) outperform most of existing VIAs against TC in terms of injection length. The BVMA and the ZKP variant deliver similar results under a proper $T$ (e.g., $< 10^2$ injected files in Enron with $T = 2,000$). The former is more applicable than the latter if the adversary prefers to exploit the *vp*.

## 4.3 Attacks against Padding

Padding [5,7,10,15,37,46,49] can protect the volume pattern. One may apply static padding [5,10,15,37] upon establishing a database or use dynamic padding [2,46,49,52] for both setup and update stages. Efficient padding strategies usually leverage *keyword clustering* to balance security and padding overhead. This technique clusters keywords so that those in a cluster share the same or computationally indistinguishable volume after padding.

We tested our attacks against both static and dynamic keyword clustering padding strategies in Enron. We used SEAL [15] as the static padding solution because it requires less query bandwidth overhead than [10,37] and can mitigate well-known attacks (e.g., [7,25]). As for the dynamic padding, we chose ShieldDB [46] (which is a more practical encrypted database than [2,52]) supporting keyword search with a dynamic countermeasure against query recovery attacks. In our experiments, we assumed that each dummy file employed dummy keywords not present in the database, utilizing the padding modules of ShieldDB and SEAL for random file padding to ensure that the file's size is within $[min\_file\_size, max\_file\_size]$ for obfuscation, where $min\_file\_size$ (resp., $max\_file\_size$) is the smallest (resp., largest) file size in the original client database. Note when using the ORAM module in SEAL, we assumed that the system uses blocks with the same size for filling (instead of dummy files with random sizes). We measured the padding overhead of SEAL besides the recovery rate. We defined the padding overhead as $N_{padding}/N_{no-padding} - 1$, which reflects a ratio between the extra overhead (i.e., number of files) of padding and "no padding". We calculated the storage overhead from setup ($Setup\&Fill$) and injection&fill ($Inj\&Fill$), and the bandwidth overhead from queries-after-setup (*S-Query*) and queries-after-$Inj\&Fill$ (*I-Query*).

**Static Padding by SEAL.** SEAL [15] has two core modules. One is the quantized ORAM module [43] hiding the *ap* and *sp*, and the other is the parameterized padding module concealing

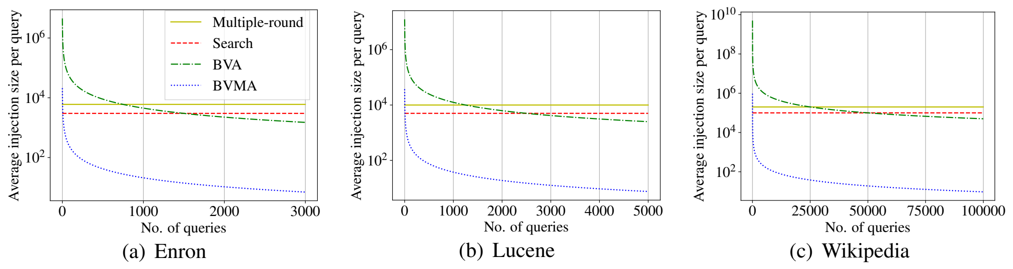


Figure 6: Average injection size with number of queries (we set the keyword partition $k = 2$ for multiple-round).

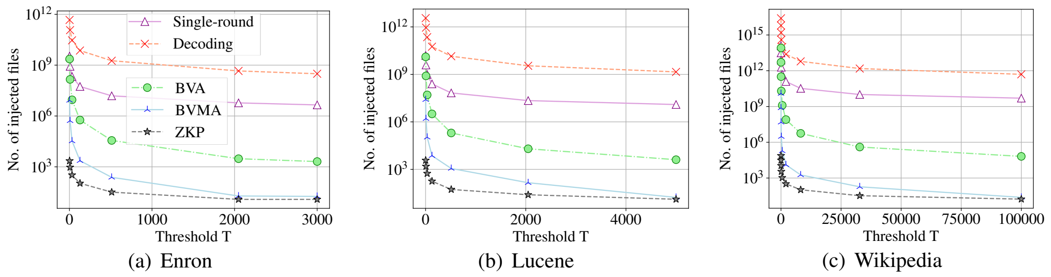


Figure 7: The number of injected files against TC (we set $m$ and $\gamma$ to $\#W/2$ for the single-round attack and BVA).

the *vp*. We assume the response is $D_q$ for a query $q$, where $D$ is the encrypted database stored in the form of ORAM blocks. Note each block in ORAM shares the same size (e.g., with $B$ keywords). The number of blocks *rlp* for a query $q$ is $\#D_q$, and the total response size *rsp* is $\#D_q \cdot B$. In this case, the *rsp* and *rlp* are somewhat "mixed", which indicates that the *rlp* can be derived from *rsp* and vice versa, so that the BVMA "degenerates" to the BVA only relying on the *rsp*. We thus only evaluated the BVA against SEAL (with both padding and ORAM, see Figure 8(b)).

In the ORAM module, for a file $f$, if its size is smaller than the block size $B$ (i.e., $|f|_w < B$), the system will pad the file to size $B$ and then store the resulting file in a random block; if $|f|_w > B$, the system will split and store the file in $\lceil |f|_w/B \rceil$ blocks. Due to ORAM, each query will obliviously access to the related blocks. In the padding module, the system can add dummy blocks with the underlying keyword of $q$ until the query's response length $\#\widetilde{D_q}$ is the next power of $x$: $\#\widetilde{D_q} = \min\{x^k : x^k > \#D_q, k \in [\log \#D]\}$.

In the experiments, we set $B$ as the average size of files in Enron. We selected a minimum $\gamma$ satisfying $\gamma \geq \#W/2 \wedge B|\gamma$ to ensure that each injected file $f$ can be exactly divided into $|f|_w/B$ blocks. We tested the BVA and BVMA against SEAL with only padding module (see Figure 8(a)) and also investigated the BVA applying both padding and ORAM modules (see Figure 8(b)). Note we tested SEAL's padding overhead instead of ORAM cost, as it was hard to evaluate the ORAM's complexity which relies on various factors (e.g., block variables, access method, bucket size). We could regard the overhead as the lower bound of SEAL's cost. This makes sense as ORAM definitely yields extra and noticeable cost.

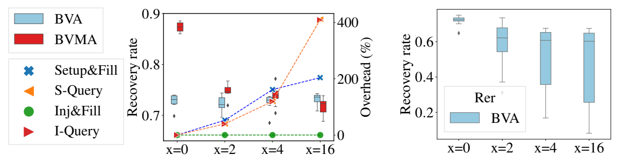
In Figure 8(a), the increase of $x$ (in the padding module) has little influence over the recovery rate. The recovery remains at about 70%. But the padding cost, especially in the setup and query, increases by 200% and 400% (when $x$ is up to 16), which seriously affects the practicability. Figure 8(b) shows that padding and ORAM modules disturb the stability of the BVA's performance. For example, when $x = 16$, the BVA achieves 70% recovery in the best case and only 10% in the worst case. The negative impact incurred by SEAL on the average recovery rate is still limited. Under various $x$, the BVA can maintain 60% recovery on average. We note that one may keep increasing $x$, but this will make SEAL become less and less practical. The results indicate that the static padding is not the best countermeasure to our attacks.

**Extend SEAL to Dynamic Padding.** We notice that SEAL [15] has potential for extension to support dynamic updates (e.g., by handling the update operations using $SD_a$ [14]); but it is currently unknown how we can properly extend SEAL to support dynamic padding. A straightforward idea[4] could be to fill the total *rlp* of the corresponding keyword into $x^t$ after every batch update. We designed an extra experiment (see Table 4) to evaluate our attacks against such a dynamic variant. We assume that after a batch injection (by our attacks), the padding module recalculates the *rlp* of each keyword and then fills it to the proper $x^t$. The padding can appropriately

[4]Following the SEAL's padding (as well as settings), the idea supports the strategy during (dynamic) update.

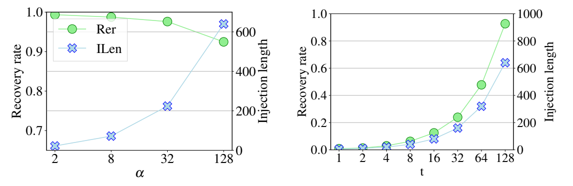
Table 4: Performance under the extended SEAL. The overhead (× / No.) represents (the ratio between the extra overhead of padding and "no padding"; the number of files). Setup&Fill and Inj&Fill are the storage overheads on the database setup and file injection, respectively. S-Query and I-Query are the average query bandwidth overheads before and after Inj&Fill, respectively.

| Extended SEAL | Overhead (× / No.) | | | | Recovery rate % | |
|---|---|---|---|---|---|---|
| | Setup&Fill | S-Query | Inj&Fill | I-Query | BVA | BVMA |
| no padding | 0 / 30$k$ | 0 / 730 | 0 / 12 (injection) | 0 / 735 | 70 | 87 |
| $x = 2$ | 0.5 / 45$k$ | 0.4 / 1,058 | 2,730 / 33$k$ | 1.8 / 2,106 | $< 1$ | $< 1$ |
| $x = 4$ | 1.6 / 79$k$ | 1.2 / 1,595 | 16,384 / 197$k$ | 7.6 / 6,316 | $< 1$ | $< 1$ |
| $x = 16$ | 2.0 / 91$k$ | 4.7 / 4,147 | 81,920 / 983$k$ | 88 / 66,027 | $< 1$ | $< 1$ |



(a) Padding. (b) Padding & ORAM.

Figure 8: BVA against static (pre-injection) padding&ORAM (SEAL). Note "x=0" is the baseline ("no defense").



(a) Recovery rate for different α. We set $t = \alpha$ in this case. (b) Recovery rate for different $t$. We set $\alpha = 128$ in this case.

Figure 9: Optimization against dynamic (post-injection) padding (ShieldDB).

resist VIAs ($< 1\%$ recovery rate) but produce huge overheads. More concretely, given $x = 2$ (i.e., the minimum padding), as compared to "no padding", the query-after-injection overhead raises by nearly 200%. Meanwhile, the number of extra files for padding expands by $> 10^3\times$. The padding requires approx. 33,000 extra files against injections and this number is even larger than the total files of the entire database (30,109 files).

**Dynamic Padding by ShieldDB.** ShieldDB [46] uses the parameter α to set the size (i.e., number of keywords) of each cluster. Concretely, assume that the added files containing the keyword $w$ are $U_w$, where $w$ is located in the keyword cluster $C$. After padding, the update length $\#\widetilde{U_w}$ of $w$ is: $\#\widetilde{U_w} = \max\{\#U_{w_m}, \forall w_m \in C\}$.

BVA and BVMA cannot distinguish the keywords belonging to the same cluster. Their recovery rates could significantly decline if ShieldDB sets a large α. For example, they achieve around 3% recovery rate under $\alpha = 128$ [50]. To tackle this issue, we propose an optimization using "multi-group" binary injections against ShieldDB. The optimized attack guarantees that the keywords in the same cluster have the same $rlp$ to bypass the padding. See Appendix H for more details.

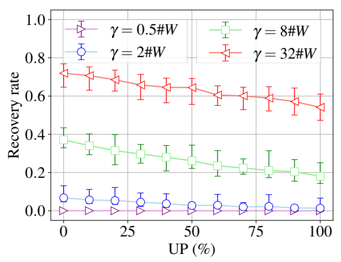
We conducted experiments to evaluate the performance of our optimized attack against ShieldDB (see Figure 9). For simplicity, we considered two batch paddings. One is to cluster and fill the keywords in the setup phase for the server; while the other is in the injection phase, where we inject specific files containing all the keywords and further use padding to obfuscate the response length. In Figure 9, we show the recovery rate and injection length against ShieldDB in Enron under different parameters $(\alpha, t)$, where $t (t \leq \alpha)$ is the number of keyword groups. The experiments demonstrated that the α has little impact on the recovery rate, and even when $\alpha = 128$, the recovery can remain $> 90\%$. To mitigate the impact of padding, we set the number of keyword groups ($t$) to be the same as that of clusters (α) in ShieldDB. We observe a steady increase in the injection length from approximately 20 to 200 and finally up to around 600. Figure 9(a) and 9(b) show that we can still counter ShieldDB by increasing the number of injected files (thereby enhancing the attack ability). The results reveal that a "larger" number of injections can lead to a "higher" recovery rate.

## 4.4 Injections with Client Active Update

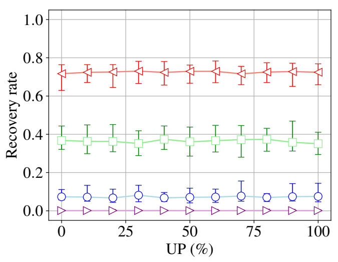
If a large number of file updates (e.g., $> 50\%$ of the database) covering almost all keywords occur between the baseline and recovery phases, our attacks without optimization may fail. To tackle this issue, we modify BVA below to limit the impact of updates while preserving an $O(log\#W)$ injection length.

In the baseline phase, the adversary observes queries' results and identifies the largest rsp (denoted as $rsp_{max}$). It predicts (or sets) the total size of all files that the client may update during the attack (denoted as $Tsize_{upd}$). In the injection phase, the adversary sets $\gamma > rsp_{max} + Tsize_{upd} \wedge \gamma \geq \#W/2$ and then injects files according to BVA's binary injection strategy. In the recovery phase, for a target query $q$, the adversary observes its response size (denoted as $rsp_q$) after injection and update and then recovers the query $q$ as the keyword $w_{\lfloor rsp_q/\gamma \rfloor}$ (see more details and experiments in [50]).

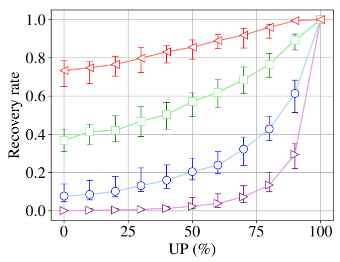
In the modified attack, we impose an upper bound of γ to eliminate the negative impact of client updates. Actually,



(a) All updates are add operations



(b) Randomly selects add or delete



(c) All updates are delete operations

Figure 10: The modified attack's recovery rate with update percentage (UP).

setting a small $\gamma$ (e.g., $\gamma = c \cdot \#W$, $c$ is a constant and $c \ll \#W$) already can help us to obtain a reasonable recovery rate. We implemented the *modified* attack and conducted experiments on Enron. We used half of Enron (about 15k files) as the real database and the rest as the auxiliary database used for "add" file operations (which means that each newly added file is from this auxiliary database). We assume the client can select an operation from $[add, delete]$ for each update. For an *add* operation, the client randomly selects a file from the auxiliary database and further creates the keyword-file indexes; as for a *delete* operation, it deletes a file and the keyword-file pairs randomly from the real database. We examined the recovery rate under the client's update scenarios: (1) all add operations (see Figure 10(a)); (2) uniform add and delete operations (see Figure 10(b)); (3) all delete operations (see Figure 10(c)).

In the experiments, we tested the impact of different $\gamma$ and update percentage $UP$ on the recovery rate, where $UP$ represents the proportion of updated files in the dataset. The results show that increasing $\gamma$ improves the recovery rate (e.g., obtaining $> 50\%$ recovery rate when $\gamma = 32\#W$), but this also leads to larger injection sizes (i.e., $O(\gamma \cdot \#W)$). Furthermore, the recovery rate shows different trends for different update operations. Figure 10(a) shows that increasing "add" operations can harm the recovery rate. For example, when $\gamma = 32\#W$, the recovery rate on $UP = 0\%$ is about 20% higher than that on $UP = 100\%$. Figure 10(b) demonstrates that random updates barely affect the attack performance. In the last sub-figure, we see that "delete" operations are beneficial for the recovery (e.g., when $\gamma = 32\#W$, the recovery rate on $UP = 50\%$ is about 10% higher than that on $UP = 0\%$). Especially when $UP = 100\%$, our modified attack achieves a 100% recovery rate on all reasonable $\gamma$ (i.e., $\gamma \geq \#W/2$). This is because both the original and the updated files are all considered as noise, and $UP = 100\%$ indicates that the noise is eliminated (i.e., all files in the original database are deleted), allowing us to obtain a perfect recovery rate.

## 5 Conclusion

We proposed new attacks against dynamic SSE using a binary volumetric injection strategy. Unlike existing attacks requiring a significant amount of prior knowledge (e.g., LAA [7,25]) and injections (e.g., [3, 38]), our attacks can offer a high recovery rate with fewer injections and pose a non-trivial threat to current defenses (e.g., padding). We provided empirical evidence to confirm the performance of our attacks. More experimental results are available in the full version [50].

## Acknowledgement

We would like to thank the shepherd and the anonymous reviewers for their valuable comments. This work was partly supported by the Guangdong Provincial Key Research and Development Plan Project under Grant No. 2019B010139001, the National Natural Science Foundation of China under Grant No. 62272186, the Wuhan Applied Foundational Frontier Project under Grant No. 2020010601012188, and the European Union's Horizon Research and Innovation Programme under Grant No. 952697 (ASSURED), No. 101021727 (IRIS), and No. 101070052 (TANGO).

## References

[1] Rakesh Agrawal, Jerry Kiernan, Ramakrishnan Srikant, and Yirong Xu. Order preserving encryption for numeric data. In *SIGMOD*, 2004.

[2] Ghous Amjad, Sarvar Patel, Giuseppe Persiano, Kevin Yeo, and Moti Yung. Dynamic volume-hiding encrypted multi-maps with applications to searchable encryption. *IACR ePrint*, 2021.

[3] Laura Blackstone, Seny Kamara, and Tarik Moataz. Revisiting leakage abuse attacks. In *NDSS*, 2020.

[4] Alexandra Boldyreva, Nathan Chenette, Younho Lee, and Adam O'neill. Order-preserving symmetric encryption. In *EUROCRYPT*, 2009.

[5] Raphael Bost and Pierre-Alain Fouque. Thwarting leakage abuse attacks against searchable encryption–a formal approach and applications to database padding. *IACR ePrint*, 2017.

[6] Raphaël Bost, Brice Minaud, and Olga Ohrimenko. Forward and backward private searchable encryption from constrained cryptographic primitives. In *CCS*, 2017.

[7] David Cash, Paul Grubbs, Jason Perry, and Thomas Ristenpart. Leakage-abuse attacks against searchable encryption. In *CCS*, 2015.

[8] David Cash, Stanislaw Jarecki, Charanjit Jutla, Hugo Krawczyk, Marcel-Cătălin Roşu, and Michael Steiner. Highly-scalable searchable symmetric encryption with support for boolean queries. In *CRYPTO*, 2013.

[9] David Cash and Stefano Tessaro. The locality of searchable symmetric encryption. In *EUROCRYPT*, 2014.

[10] Guoxing Chen, Ten-Hwang Lai, Michael K Reiter, and Yinqian Zhang. Differentially private access patterns for searchable symmetric encryption. In *INFOCOM*, 2018.

[11] Reza Curtmola, Juan Garay, Seny Kamara, and Rafail Ostrovsky. Searchable symmetric encryption: improved definitions and efficient constructions. In *CCS*, 2006.

[12] Marc Damie, Florian Hahn, and Andreas Peter. A highly accurate query-recovery attack against searchable encryption using non-indexed documents. In *USENIX Security*, 2021.

[13] Emma Dauterman, Eric Feng, Ellen Luo, Raluca Ada Popa, and Ion Stoica. DORY: an encrypted search system with distributed trust. In *OSDI*, 2020.

[14] Ioannis Demertzis, Javad Ghareh Chamani, Dimitrios Papadopoulos, and Charalampos Papamanthou. Dynamic searchable encryption with small client storage. In *NDSS*, 2020.

[15] Ioannis Demertzis, Dimitrios Papadopoulos, Charalampos Papamanthou, and Saurabh Shintre. SEAL: attack mitigation for encrypted databases via adjustable leakage. In *USENIX Security*, 2020.

[16] Roy Fielding and Reschke. Hypertext transfer protocol (http/1.1): Message syntax and routing. *RFC 7230*, 2014.

[17] Apache Foundation. Mail archieves of lucene, 1999. https://mail-archives.apache.org/mod_mbox/#lucene.

[18] Wikipedia Foundation. Wikipedia databases, 2020. https://www.wikipedia.org.

[19] Sanjam Garg, Payman Mohassel, and Charalampos Papamanthou. Tworam: efficient oblivious ram in two rounds with applications to searchable encryption. In *CRYPTO*, 2016.

[20] Marilyn George, Seny Kamara, and Tarik Moataz. Structured encryption and dynamic leakage suppression. In *EUROCRYPT*, 2021.

[21] Javad Ghareh Chamani, Dimitrios Papadopoulos, Charalampos Papamanthou, and Rasool Jalili. New constructions for forward and backward private symmetric searchable encryption. In *CCS*, 2018.

[22] Google. Google trends, 2004. https://trends.google.com/trends/.

[23] Peeyush Gupta, Sharad Mehrotra, Shantanu Sharma, Nalini Venkatasubramanian, and Guoxi Wang. Concealer: Sgx-based secure, volume hiding, and verifiable processing of spatial time-series datasets. In *EDBT*, 2021.

[24] Yanyu Huang, Siyi Lv, Zheli Liu, Xiangfu Song, Jin Li, Yali Yuan, and Changyu Dong. Cetus: an efficient symmetric searchable encryption against file-injection attack with SGX. *Sci. China Inf. Sci.*, 2021.

[25] Mohammad Saiful Islam, Mehmet Kuzu, and Murat Kantarcioglu. Access pattern disclosure on searchable encryption: Ramification, attack and mitigation. In *NDSS*, 2012.

[26] Seny Kamara, Abdelkarim Kati, Tarik Moataz, Thomas Schneider, Amos Treiber, and Michael Yonli. Sok: Cryptanalysis of encrypted search with leaker – a framework for leakage attack evaluation on real-world data. In *EuroS&P*, 2022.

[27] Seny Kamara and Tarik Moataz. Boolean searchable symmetric encryption with worst-case sub-linear complexity. In *EUROCRYPT*, 2017.

[28] Seny Kamara and Tarik Moataz. Computationally volume-hiding structured encryption. In *EUROCRYPT*, 2019.

[29] Seny Kamara, Tarik Moataz, and Olga Ohrimenko. Structured encryption and leakage suppression. In *CRYPTO*, 2018.

[30] Seny Kamara, Charalampos Papamanthou, and Tom Roeder. Dynamic searchable symmetric encryption. In *CCS*, 2012.

[31] Evgenios M. Kornaropoulos, Nathaniel Moyer, Charalampos Papamanthou, and Alexandros Psomas. Leakage inversion: Towards quantifying privacy in searchable encryption. In *CCS*, 2022.

[32] Shangqi Lai, Sikhar Patranabis, Amin Sakzad, Joseph K Liu, Debdeep Mukhopadhyay, Ron Steinfeld, Shi-Feng Sun, Dongxi Liu, and Cong Zuo. Result pattern hiding searchable encryption for conjunctive queries. In *CCS*, 2018.

[33] Marcel Ruiz Forns MusikAnimal, Kaldari. Pageviews toolforge, 2015. https://pageviews.toolforge.org/.

[34] Jianting Ning, Xinyi Huang, Geong Sen Poh, Jiaming Yuan, Yingjiu Li, Jian Weng, and Robert H. Deng. LEAP: leakage-abuse attack on efficiently deployable, efficiently searchable encryption with partially known dataset. In *CCS*, 2021.

[35] Simon Oya and Florian Kerschbaum. Hiding the access pattern is not enough: Exploiting search pattern leakage in searchable encryption. In *USENIX Security*, 2021.

[36] Sarvar Patel, Giuseppe Persiano, and Kevin Yeo. Symmetric searchable encryption with sharing and unsharing. In *ESORICS*, 2018.

[37] Sarvar Patel, Giuseppe Persiano, Kevin Yeo, and Moti Yung. Mitigating leakage in secure cloud-hosted data structures: Volume-hiding for multi-maps via hashing. In *CCS*, 2019.

[38] Rishabh Poddar, Stephanie Wang, Jianan Lu, and Raluca Ada Popa. Practical volume-based attacks on encrypted databases. In *EuroS&P*, 2020.

[39] David Pouliot and Charles V. Wright. The shadow nemesis: Inference attacks on efficiently deployable, efficiently searchable encryption. In *CCS*, 2016.

[40] Zhiwei Shang, Simon Oya, Andreas Peter, and Florian Kerschbaum. Obfuscated access and search patterns in searchable encryption. In *NDSS*, 2021.

[41] David Shapiro. Convert wikipedia database dumps into plaintext files, 2021. https://github.com/daveshap/PlainTextWikipedia.

[42] Dawn Xiaoding Song, D. Wagner, and A. Perrig. Practical techniques for searches on encrypted data. In *S&P*, 2000.

[43] Emil Stefanov, Marten van Dijk, Elaine Shi, Christopher W. Fletcher, Ling Ren, Xiangyao Yu, and Srinivas Devadas. Path ORAM: an extremely simple oblivious RAM protocol. In *CCS*, 2013.

[44] Liling Tan Steven Bird. Nltk corpus, 2021. https://www.nltk.org/howto/corpus.html.

[45] Shifeng Sun, Xingliang Yuan, Joseph K. Liu, Ron Steinfeld, Amin Sakzad, Viet Vo, and Surya Nepal. Practical backward-secure searchable encryption from symmetric puncturable encryption. In *CCS*, 2018.

[46] Viet Vo, Xingliang Yuan, Shifeng Sun, Joseph K. Liu, Surya Nepal, and Cong Wang. Shielddb: An encrypted document database with padding countermeasures. *TKDE*, 2021.

[47] Jianfeng Wang, Shi-Feng Sun, Tianci Li, Saiyu Qi, and Xiaofeng Chen. Practical volume-hiding encrypted multi-maps with optimal overhead and beyond. In *CCS*, 2022.

[48] CMU William W. Cohen, MLD. Enron email datasets, 2015. https://www.cs.cmu.edu/~./enron/.

[49] Lei Xu, Anxin Zhou, Huayi Duan, Cong Wang, Qian Wang, and Xiaohua Jia. Toward full accounting for leakage exploitation and mitigation in dynamic encrypted databases. IACR ePrint, 2022.

[50] Xianglong Zhang, Wei Wang, Peng Xu, Laurence T. Yang, and Kaitai Liang. High recovery with fewer injections: Practical binary volumetric injection attacks against dynamic searchable encryption. ArXiv, 2023. https://arxiv.org/abs/2302.05628.

[51] Yupeng Zhang, Jonathan Katz, and Charalampos Papamanthou. All your queries are belong to us: The power of file-injection attacks on searchable encryption. In *USENIX Security*, 2016.

[52] Yongjun Zhao, Huaxiong Wang, and Kwok-Yan Lam. Volume-hiding dynamic searchable symmetric encryption with forward and backward privacy. *IACR ePrint*, 2021.

## A Summary of Notations and Concepts

We denote $[n]$ as the set of integers $\{1,...,n\}$. For a set $S$, we use $\#S$ to refer to its cardinality. For a file $f$ (or file set $F$), we use $|f|_w$ ($|F|_w$) to represent its word count, also called file *size*. $\mathcal{A} \to x$ means that $x$ is the output of an algorithm $\mathcal{A}$. $\mathbb{N}$ denotes natural number. We use $\lambda$ to denote the security parameter. See the frequently used notations in Table 5.

## B Searchable Symmetric Encryption

A standard dynamic SSE [30] includes a polynomial-time algorithm, $Setup$, executed by the client and two protocols, $Query$ and $Update$, run between the client and the server.
• $Setup(\lambda)$: This probabilistic algorithm takes the security parameter $\lambda$ as the input and outputs $(K,\delta;D)$, where $K$ and $\delta$ are the secret key and state of the client, respectively; and $D$ is an empty encrypted database stored in the server.
• $Update(K,\delta,op,(w,id);D)$: The update protocol carries $K$, $\delta$, $op \in \{add, delete\}$, a keyword-file pair $(w,id)$, and $D$ as input. It outputs a new client state $\delta'$ and an encrypted database $D'$ after the "add/delete" operation.
• $Query(K,\delta,q;D)$: This protocol takes the client secret key $K$, state $\delta$, and query $q$ as input, in which the server takes the database $D$ as input. It returns $D(q)$ as the query's response.

In the query protocol, the client only retrieves the response identifiers. Later, it can perform an extra interaction with the server to obtain the encrypted files with the identifiers.

## C Definition for Attack Round

The multiple-round [38] and search attacks [3] recover one target query via many rounds of injections and observations. Recall that the adversary in the context of the multiple-round attack "forces" the client to replay the same query repeatedly, in which "completing a replay of the client query" is regarded as "one attack *round*". The search attack requires the adversary to carry out a sufficient amount of observations on the client's queries after a successful single file injection. In this context, an attack *round* is referred to as "the adversary completes a single file injection". In other attacks for multiple queries, the adversary aims to reveal the queries through an immediate observation right after multiple files injection. Here, completing a batch of file injections is also seen as an attack round. We define the attack *round* as follows.

**Definition 1** *Let $OB$ be a non-empty set (except the $OB_0$) of continuous observation operations and $INJ$ denote a non-empty set containing continuous "inject" operations. We denote $\Gamma = (OB_0, INJ_1, OB_1, INJ_2, OB_2, ...INJ_t, OB_t)$ as all the pairwise operations for an injection attack. Then, the total number of the attack round is $t$.*

For example, in the search attack with a $W$, the attack process is $\Gamma = (OB_0,\ INJ_1,\ OB_1,\ ...,\ INJ_{\lceil \log \#W \rceil}, OB_{\lceil \log \#W \rceil})$. Thus, the number of attack round is $\lceil \log \#W \rceil$. As for other attacks against multiple queries (e.g., single-round, decoding, and our attacks) their attack process is as $\Gamma = (OB_0, INJ_1, OB_1)$ and thus, they only require *one attack round*.

Table 5: Notation and Concept

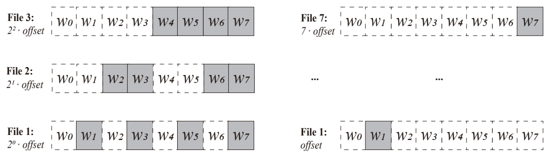
| Notation | Description |
|---|---|
| $D$ | An encrypted database. |
| $W$ | A keyword universe $W = \{w_1, w_2, ..., w_m\}$. |
| $Q$ | A sequence of queries $Q = \{q_1, q_2, ..., q_l\}$. |
| $R$ | The response to each query. |
| $F$ | A set of injected files. |
| $\|f\|_w$ | Total word count for a file $f$. |
| $file_u$ | Injected files containing a keyword $w_u$. |
| *offset* | Minimum file size for decoding. |
| $\gamma$ | Basic injection size for BVA. |
| $k$ | Keyword partitions for multiple-round. |
| $m$ | Injection constant for single-round. |
| $T$ | Size threshold of each file. |
| $LP$ | Leakage pattern. |
| $RL$ | Response length $RL = \{rl_1, rl_2, ..., rl_l\}$. |
| $RS$ | Response size $RS = \{rs_1, rs_2, ..., rs_l\}$. |
| $Freq$ | Query frequency $Freq = \{freq_1, ..., freq_l\}$. |
| $Q_r$ | A query recovery set, $Q_r \subseteq Q$. |
| Leakage Pattern | Description |
| access pattern (*ap*) | identifiers of the files matching a query. |
| access injection pattern (*aip*) | identifiers of injected files matching a query. |
| search pattern (*sp*) | whether a query is repeated. |
| response length pattern (*rlp*) | the number of (response) returned files. |
| response size pattern (*rsp*) | the total word count of (response) returned files. |
| volume pattern (*vp*) | include *rlp* and *rsp*. |
| Volume Information | Description |
| injection length (ILen) | the number of injected files. |
| injection size (ISize) | the total word count of injected files. |
| injection volume | include ILen and ISize. |
| update length | the number of update files. |
| Attack | Description |
| passive attack | attacks only based on the observations. |
| injection attack | attacks can actively inject files. |
| volumetric injection attacks (VIAs) | injection attack relying on *rlp* (or *rsp*). |
| Defense | Description |
| ORAM | oblivious retrieval to hide *ap* and *aip*. |
| padding | index dummy files to obfuscate *vp*. |
| TC | limit file size to avoid large-size file injection. |

## D An Example: BVA *vs*. Decoding

In the example (see Figure 11), we assume there exists a special case where $\gamma = offset$ so that the BVA and decoding attack can achieve the same recovery rate. We see that the BVA only needs three files (each with a size of $2^{k-1} \cdot offset$) as compared to the decoding attack injecting seven files (each with a size of $(t-1) \cdot offset$), where $k \in [3]$ and $t \in [8]$. Specifically, the BVA yields $7 \cdot offset$ injection size, while the decoding attack is $4\times$ of the cost of the BVA, i.e., $28 \cdot offset$.

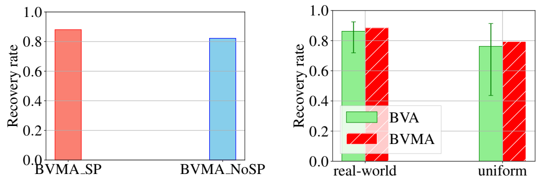
## E Effect of Search Pattern on BVMA

We tested the effect of the sp on the recovery rate of the BVMA (see Figure 12(a)). By exploiting the sp (i.e., "BVMA_SP"), the attack slightly improves the recovery, around 5% (under 100% keywords leakage); without the sp (i.e., "BVMA_NoSP"), it still provides around 80% recovery.



(a) BVA injection (b) Decoding injection [3]

Figure 11: An example: the adversary knows $\#W = 8$.



(a) Impact of sp on BVMA. (b) BVA and BVMA under different query distributions.

Figure 12: Supplementary experiments of BVA and BVMA.

## F Queries with Different Distributions

We used Google Trends and PageViews Toolforge to simulate the queries in the real-world distribution. Alternatively, one may use other query distributions, such as uniform query. In this section, we tested the recovery rate under two query distributions: real-world and uniform (see Figure 12(b)). We assumed that the adversary knows $W$ and has spent at least 8 weeks on observations in Enron. We set error bars for the BVA to evaluate the recovery when $\gamma \in [\#W/2, offset/4]$.

The recovery rate of the BVA varies moderately (about a 20% gap) under the real-world query, but fluctuates (e.g., $< 50\%$ recovery in the worst case and $> 90\%$ in the best case) under the uniform distribution. This indicates that the BVA with a small $\gamma$ (e.g., $\gamma = \#W/2$) cannot perform stably in uniform query. The average recovery rates on the two distributions only differ approximately 10%. The gap could nearly disappear if we set $\gamma$ to be sufficiently large (e.g., $\gamma = offset/4$) to yield the max. recovery under both distributions. Recall that besides the frequency information, the BVMA relies on two types of volume patterns for query recovery. The adversary can distinguish queries and exclude the incorrect keywords by exploiting the patterns. The uniform query (hiding frequency information) only harms the recovery of the BVMA by 10% as compared to the real-word query.

The distributions do not seriously affect our attacks' average recovery rate ($< 10\%$). The attacks perform slightly better on the real-world distribution than the uniform. This is because under the real-world query, the adversary can always recognize its target query of which frequency is higher than others. In contrast, the uniform distribution makes the adversary difficult to distinguish the target query, incurring the drop of the recovery rate.

## G Comparison in Lucene and Wikipedia

We show the comparison among the single-round, decoding and our attacks in Lucene and Wikipedia (see Table 6 and Figure 13, 14). The single-round attack is least time-consuming ($< 1s$) but yields the poorest recovery (when $m = 1$) or injection length (when $m = \#W$) in Lucene. The decoding attack and our attacks are at a comparable performance level in terms of recovery and running time. In Wikipedia, our attacks achieve around $60-80\%$ recovery in which the BVA and BVMA take around 3 and 15 mins respectively. For injection length, other attacks inject $10^2\times$ more files than ours. The BVMA only requires $10^6$ injection size in Wikipedia. To maintain a similar level of recovery (e.g., $Kws = 0.25$), the decoding and single-round attacks (with $m = \#W$) cost at least $10^6\times$ more injection size than the BVMA. We see that the BVMA in Wikipedia (up to 60% recovery) does not perform as well as in Enron and Lucene. If it performs on the same injection size as the BVA (but still much $<$ that of the decoding and single-round attacks), its recovery will be close to the BVA's.

## H Optimization against ShieldDB

We developed an optimization (see Algorithm 4) against ShieldDB [46], which is an extension of the BVA and BVMA. It starts with the BVA injection method and further optimizes the method by keyword grouping. Following the BVMA, the optimization uses the $rlp$ to determine the keyword clustering. Following the attack model, it has three main stages.

In the baseline phase, we determine the cluster $CW$ at which each keyword is located (line 3) and record $rsp$ of queries as $\widetilde{RS}$ (line 4). Here, each row $CW[i,:]$ represents a set containing all the keywords in cluster $i$, and each column $CW[:,j]$ represents the keyword set under the same position ($j$) of different clusters. Recall that in the update phase of ShieldDB, there are two *cases* for file upload according to if the keywords of clusters have been stored in the server.

*Case 1:* The server's current database has not yet contained any keywords of a cluster. In this case, only after all keywords of the cluster have been (updated and) stored in the local cache, they are filled and uploaded.

*Case 2:* All the keywords in the cluster have already been stored in the server's database. There are two conditions that trigger the uploads: 1) when the time interval between the last upload and the current time slots exceeds a certain threshold (i.e., *Tthreshold*); 2) when the number of keyword-file pairs (i.e., (w, id)) of the cluster locally stored exceeds a concrete size (i.e., *Cthreshold*).

Once the corresponding conditions are satisfied, the system

Table 6: Running time for query recovery ($10 \times 1,000$ queries for Enron and Lucene, $10 \times 5,000$ queries for Wikipedia). Note the cost of the BVA is within the time range by varying $\gamma = [\#W/2, offset/4]$.

| Running time | Decoding | Single-round $(m = 1)$ | Single-round $(m = \#W)$ | BVA | BVMA |
|---|---|---|---|---|---|
| Enron | $2.48s$ | $0.01s$ | $0.02s$ | $(1.81s, 2.46s)$ | $19.16s$ |
| Lucene | $3.54s$ | $0.01s$ | $0.02s$ | $(2.53s, 3.15s)$ | $33.10s$ |
| Wikipedia | $3min\ 42s$ | $0.05s$ | $0.06s$ | $(2min\ 52s,\ 3min\ 37s)$ | $15min\ 34s$ |

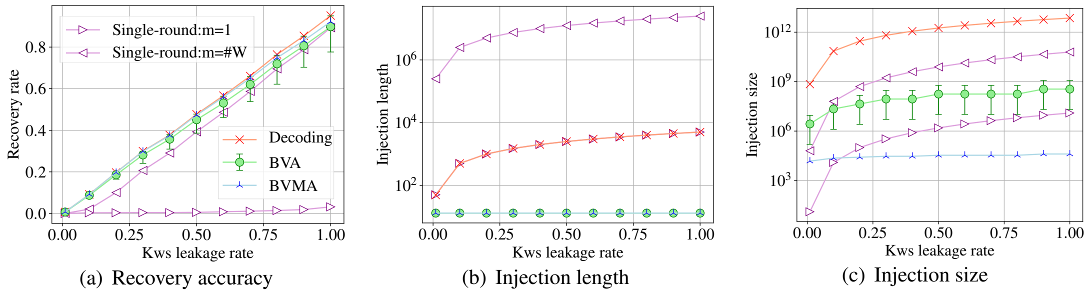


Figure 13: Comparison on the recovery rate, injection length, and injection size with different Kws leakages in **Lucene**.

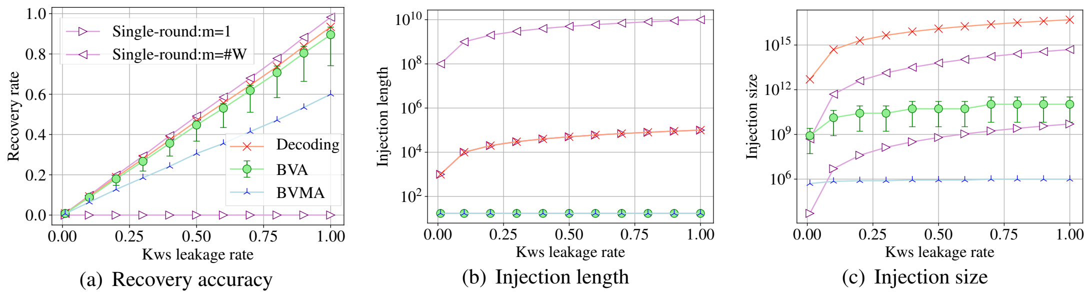


Figure 14: Comparison on the recovery rate, injection length, and injection size with different Kws leakages in **Wikipedia**.

fills the relevant keywords and uploads them to the server. By utilizing the upload strategy of ShieldDB, we can use the observe-inject-observe approach to guess the cluster for a $w$. First, we inject a file containing all keywords and wait for a time period $> Tthreshold$ to ensure that the injected file has been uploaded to the server. It ensures that all keywords have been included in the server, and the subsequent uploads (and injections) follow *Case 2*. To predict the cluster, we observe the *rlp* of all queries, send an injected file containing only $w$ to the client, wait for a duration $> Tthreshold$, and then observe the *rlp* once more. In this case, only the cluster containing $w$ is uploaded to the server (where none of the keywords in other clusters is updated). We then determine that $w$ belongs to the cluster with the "changed" *rlp* after the injection. We can see that the above process requires an $O(\#W)$ injection length and attack rounds. Note that we can also determine the cluster by using extra prior knowledge. Recall that ShieldDB uses a training dataset to cluster keywords in the real dataset. The training dataset could not be kept entirely confidential. We can leverage other public datasets with similar distributions to identify the cluster.

In the injection phase (lines 5-10), we choose $t$ $(t \leq \alpha)$ groups of keywords, with each group containing the keywords in a column of $CW$. We select different $\gamma$ (from the set $\Gamma$) for each group to launch the binary injection (similar to the BVA). This grouping approach ensures that the target keywords in the cluster share the same number of injected files. After injection and padding, we record the total response length of each keyword as $TRL$ (line 13).

In the recovery phase (lines 14-20), given a target query $q_i$, we first identify the candidate clusters by using $q_i$'s *rlp* and then collect the correct keywords $CW[v,u]$, from the clusters, satisfying: $rs_i - v \cdot \Gamma[u] = \widetilde{rs}_j, if\ \exists \widetilde{rs}_j \in \widetilde{RS}$, where $rs_i$ is the *rsp* of $q_i$, and $\Gamma$ is the injection parameter set.

## I Discussions

**Other Countermeasures** Besides the clustering-based padding strategies [5, 7, 15, 46, 49], one may consider using other defense systems. For example, some PIR (e.g., DORY [13]) and ORAM (e.g., [20, 29]) related techniques focus on protecting the *ap* and *sp* (besides the *vp*). These solutions could not be as simple and natural as a direct padding (on *vp*). Chen et al. [10] obfuscated the *ap* and *vp* by adding random false-positive and false-negative files, but this approach cannot protect the *sp*. Patel et al. [37] and Wang et al. [47] introduced the volume-hiding encrypted multi-maps with low server storage. Shang et al. [40] proposed to hide the *sp* by obfuscating the search token. As these technologies do not protect the leakage in the dynamic context, they cannot work properly against VIAs. Kamara and Moataz [28] investigated the dynamic volume-hiding system. But their approach does not support the client to make *atomic* update, e.g., adding and deleting a *single* keyword-file pair; and it also requires a high query bandwidth. [2, 52] extended KM [28] and PPYY [37] to propose fully dynamic volume-hiding encryption systems, respectively. They can resist most of the query recovery attacks with a price that the query complexity is proportional to the maximum *rlp* ($O(rlp_{max})$). We say that an effective countermeasure to VIAs should be *hybrid* and *probabilistic*, i.e., being able to hide both file size and response length by random (or differentially private) noisy padding.

It is also interesting to see that some systems, such as [23, 24], apply trusted hardware (e.g., SGX) to counter VIAs. We say that is orthogonal to this work.

**Queries with multiple keywords.** Our attacks can also apply to those dynamic SSE schemes that enable multiple keywords queries [8, 32]. We here present a heuristic solution to support our attacks for conjunctive queries with two keywords. Given $W = \{w_1, \dots, w_m\}$, we combine any two keywords into one search query. That means a new search set $\overline{W} = \{\overline{w_t} | \overline{w_t} = (w_i, w_j), i, j \in [m], i \neq j)\}$ is generated while a tuple $(w_i, w_j)$ is paired as a conjunctive query. For each element $\overline{w}_t \in \overline{W}$, we inject a file with size of $t \cdot \gamma$ ($\gamma$ is the parameter of BVA) containing the element. As a result, we inject $m(m-1)/2$ files in total. It is an open problem to reduce the injection volume for the conjunctive queries.

**SEAL's Dynamic Padding.** We used a straightforward approach to extend SEAL to support the dynamic padding. The extension can effectively resist VIAs, but its overhead (w.r.t. Inj&Fill) is extremely impractical, for example, the cost is even higher than simply streaming the entire database to the client. It is an open problem to design a more practical solution to balance the security and efficiency. One may consider combining clustering-based schemes (e.g., [7]) with the probabilistic padding (e.g., [37]). For example, for each batch update, one may divide keywords into several clusters and then fill each keyword in a cluster to $x^t + noise$, so that the keyword's *rlp* will contain the same $x^t$ and probabilistic (and different) noise, where $x$ is the parameter of SEAL.

**LEAKER.** Kamara et al. [26] proposed an open-source attack evaluation framework LEAKER to test the recovery of passive LAA with different known file rates. LEAKER unfortunately cannot cover injection attacks [3, 38, 51] as it does not capture the necessary information leakage obtained and used by active adversaries. Extending this evaluation framework to test injection attacks could be an interesting problem. We note that is orthogonal to the main focus of this work.

**Algorithm 4:** Optimization against ShieldDB

```
procedure OP_against_ShieldDB($\widetilde{Q}, W, Q, \alpha, t$)
    // Baseline
    Pick up the keywords $CW[i, j]$ which represents the $j$th
      keyword in the $i$th cluster ;   // adversary can obtain
      this information based on other knowledge or
      additional injection
    Observe the leakage information $\widetilde{RS}$ as in BVA.Baseline($\widetilde{Q}$);
    // Injection
    $Inj\_F \leftarrow \emptyset$;
    Select $t$ co-prime numbers $\Gamma[1], \dots, \Gamma[t]$ satisfying that
      $\{\Gamma[k] \geq \#W/(2\alpha) \mid k \in [t]\}$;
    for $j = 1 \rightarrow t$ do
        inject files $F$ by calling algorithm BVA.Injection($CW[:, j]$)
          but replace the injection parameter $\gamma$ with $\Gamma[j]$;
        $Inj\_F = Inj\_F \cup F$;
    // Padding
    Pad $Inj\_F$ to $IPad\_F$ according to the padding strategy and
      upload $IPad\_F$ to server ;  // Completed by the client
    Record the total response length $TRL$ of each keyword after
      injection and padding ;  // adversary can obtain this
      information with the help of $CW, Inj\_F$
    // Recovery
    initialize an empty set $Q_r$;
    gather the new observed response size $RS$, response length $RL$
      for victim's target queries $Q$;
    for $i = 1 \rightarrow \#RS$ do
        $CA \leftarrow \emptyset$;
        add keywords to $CA[:, :]$ from $CW[j, :]$ satisfying that
          $TRL[j] \equiv RL[i], j \in [\#CW]$
        find $CW[v, u]$ satisfying that $rs_i - v \cdot \Gamma[u] = \widetilde{rs}_j$ for
          $CW[v, u] \in CA, \exists \widetilde{rs}_j \in \widetilde{RS}$ add $CW[v, u]$ to $Q_r$;
    return $Q_r$;
```

**Experiments on Multiple Datasets.** In Section 4.3, 4.4 and Appendix E, F, we conducted the experiments in Enron. Our attacks performance will not differ significantly in Lucene and Enron as these email datasets share similar scales and keyword distributions. The Wikipedia could moderately harm the performance. For example, the recovery against static padding will be similar to that of Figure 14 (around 60%); whilst dynamic padding could further restrict the attacks performance. Under the clients' active updates, the attacks could achieve similar trends and results as in Figure 10. Interested readers may use our codes to conduct extra experiments.