Vaccine: Obfuscating Access Pattern Against File-Injection Attacks

Hao Liu† Boyang Wang† Nan Niu† Shomir Wilson‡ Xuetao Wei†
†University of Cincinnati ‡Pennsylvania State University
liu3ho@mail.uc.edu, {boyang.wang, nan.niu}@uc.edu, shomir@psu.edu, weix2@ucmail.uc.edu

***Abstract*—Searchable Encryption can search over encrypted data without accessing data or queries in plaintext. It preserves privacy while queries are performed over data on an untrusted server. To ensure the efficiency of search, most Searchable Encryption schemes reveal *access patterns*, i.e., a server learns which encrypted files are retrieved for each query. Unfortunately, by collecting access patterns, a *file-injection attack* can completely compromise the query privacy offered by Searchable Encryption.**

In this paper, we propose a novel *pre-encryption obfuscation* mechanism, referred to as *Vaccine*, which can effectively protect searchable encrypted data against a file-injection attack. Specifically, the main idea of Vaccine is to introduce a *self file-injection attack*, which obfuscates access patterns obtained by an attacker and prevents this attacker from inferring correct queries in plaintext. In addition, by harnessing natural language processing techniques, Vaccine can effectively remove self-injected files from search results, and therefore introduce minimal tradeoffs. Our experimental results on a real-world dataset show that Vaccine can reduce an adversary's guessing probability from 1 to 3.7×10^{-3}, which significantly promotes privacy protection. Furthermore, Vaccine introduces only 3.4% false negatives and no false positives in search results.

I. Introduction

Searchable Encryption [1]–[5] can search over encrypted data without accessing data or queries in plaintext. Specifically, with Searchable Encryption, a data owner can outsource a set of files to a server in ciphertext. In addition, this data owner can submit a *search token* (a keyword query in ciphertext), and the server can return encrypted files containing this keyword without decrypting files or queries. Searchable Encryption can preserve *data privacy* and *query privacy* against an untrusted server [2]. Many applications, such as CryptoDB [6], Google's Big Query [7] and Microsoft Always Encrypted Database [8], have implemented Searchable Encryption.

In order to ensure the efficiency of search over encrypted data, most schemes need to reveal *access patterns*, i.e., a server learns which encrypted files are retrieved for each query [2]. Unfortunately, by collecting access pattern, a *file-injection attack* [9] is able to completely compromise the query privacy rendered by Searchable Encryption. Specifically, an attacker in a file-injection attack injects a set of files (e.g., emails) to a data owner, where each file contains keywords selected by this attacker. If a data owner encrypts all those injected files and stores those encrypted files on an untrusted server, then any query in plaintext will map to a unique access pattern on all those encrypted injected files. As a result, given a search token and its corresponding access pattern on injected files, an attacker can easily indicate which keyword query a data owner is searching for. A file-injection attack is essentially a *chosen-document attack* [10]. Details of this file-injection attack are further presented in Sec. II.

In principle, leveraging Oblivious RAM [11] can hide access patterns and preserve privacy against file-injection attacks [12]. However, recent research [13] has shown that implementing Oblivious RAM in a Searchable Encryption scheme would introduce an unbearable cost, where the communication overhead of a query is even higher than directly retrieving an entire dataset itself. Searchable Encryption schemes [14]–[19] supporting *forward security* can mitigate leakage if a search token is submitted before the completion of a file-injection attack, but they still completely reveal query privacy for any search token submitted after the attack. In light of the limitations of these existing solutions, there is an emerging and imperative need to develop efficient and effective countermeasures to minimize privacy leakage under file-injection attacks.

In this paper, we propose a novel *pre-encryption obfuscation* mechanism, referred to as *Vaccine*, to effectively minimize the privacy leakage of Searchable Encryption under a file-injection attack. Specifically, our main idea in Vaccine is to develop a *self file-injection attack*, where self-injected files produced by the data owner obfuscate the access pattern on files injected by an attacker. Consequently, an attacker would fail to distinguish unique access patterns, thereby failing to uncover queries in plaintext using a file-injection attack. Our main contributions are summarized below:

- Our proposed mechanism can effectively preserve access patterns under a file-injection attack, and mitigate the privacy leakage of Searchable Encryption. Moreover, Vaccine can be applied to any Searchable Encryption scheme running keyword queries. As necessary tradeoffs, Vaccine increases encryption time, search time and storage requirements 2X in *the worst case*. With additional methods in practice, these tradeoffs can be optimized to 1.29X on a real-world dataset.
- By harnessing *natural language processing*, Vaccine introduces minimal impacts to the the correctness of search results. Specifically, self-injected files, which alleviate privacy leakage under a file-injection attack, have no semantic value and can be excluded from the search results for a keyword query. By building a semantic filter based on n-*grams* [20] and *cross entropy* [21], Vaccine can

assist a data owner to automatically remove self-injected files from search results.

- We leverage the *number of indistinguishable bits in an access pattern vector* as the main privacy metric. Based on this privacy metric, we formally define privacy in terms of an attacker's guessing probability, and rigorously analyze the privacy of Vaccine.
- Our experimental results on the Enron email dataset [22] show that an attacker, who was able to correctly infer a query with a probability of 1 on a system not protected by Vaccine, can infer a correct query with a probability of only 3.7×10^{-3} on average under the protection offered by Vaccine. A semantic filter in Vaccine is highly efficient. It only takes 3 MBs storage overhead and 2.03 seconds to build. Moreover, it only introduces 3.4% false negatives and no false positives in search results.
- With the privacy mitigation, Vaccine forces an attacker to change their attack strategy. For instance, an attacker would have to keep resending duplicate or similar emails, which diminishes the effectiveness of the attack. More importantly, this misbehavior is abnormal compared to legitimate emails and can be utilized as a critical feature to detect file injection attacks.

There are two notable limitations to our contributions in this paper. First, our scheme does not completely thwart file-injection attacks but instead mitigates privacy leakage to an attacker performing a file-injection attack. If an attacker could inject an unlimited number of injected files, query privacy would eventually be revealed since Searchable Encryption needs to render correct search functions. Second, our proposed scheme aims to obfuscate access patterns particularly against file-injection attacks, where the assumptions of an attacker in our system and threat model are the same as the assumptions of file-injection attacks described in [9]. Our scheme does not support privacy mitigation for other query types, such as range queries, which are not involved in file-injection attacks. Moreover, our approach is *not a generic solution* hiding access pattern against other attacks, such as inference attacks [23], [24] or query injection attacks [23], where the assumptions and prior knowledge of the adversaries are different.

II. Background

A. System Model

As shown in Fig. 1, the system model of a Searchable Encryption (SE) scheme includes two parties, a *data owner* and a *server*. The data owner outsources a set of files to the server to reduce local storage costs. Each file that the data owner uploads to the server contains several keywords. When the data owner searches the files by submitting a keyword, the server returns associated files that include this keyword.

Due to concerns about outside or inside attacks on the server, the data owner does not fully trust the server with the privacy of their files or queries. As a result, the data owner encrypts their files with SE before uploading them to the server. For each keyword query, the data owner generates a search token by leveraging SE. With a search token, the server is able to search the encrypted files without obtaining these files or keyword queries in plaintext.

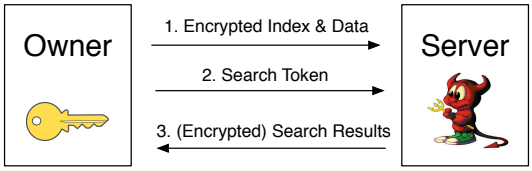

Fig. 1. The system model of a SE scheme.

B. Searchable Encryption

A SE scheme contains five algorithms, including GenKey, Enc, $\mathsf{GenToken}$, Query and Update,

- $sk \leftarrow \mathsf{GenKey}(1^\lambda)$*: is a probabilistic algorithm that is run by the data owner. It takes a public security parameter* λ *as input, and outputs a secret key* sk*.*
- $(\Gamma, \mathbf{c}) \leftarrow \mathsf{Enc}(\mathbf{f}, sk)$*: is a probabilistic (or deterministic) algorithm that is run by the data owner. It takes a set of files* $\mathbf{f} = \{f_1, ..., f_n\}$ *and a secret key* sk *as input, and outputs an encrypted index* Γ *and a set of encrypted files* $\mathbf{c} = \{c_1, ..., c_n\}$*.*
- $tk_w \leftarrow \mathsf{GenToken}(w, sk)$*: is a probabilistic (or deterministic) algorithm that is run by the data owner. It takes a keyword query* w *and a secret key* sk *as input, and outputs a search token* tk_w*.*
- $\mathbf{I}_w \leftarrow \mathsf{Query}(\Gamma, tk_w)$*: is a deterministic algorithm that is run by a server. It takes an encrypted index* Γ *and a search token* tk_w*, and outputs a set of identifiers* $\mathbf{I}_w$*, where if* $w \in f_i$*, then* $I_i \in \mathbf{I}_w$*, for* $i \in [1, n]$*.*
- $(\Gamma', c) \leftarrow \mathsf{Update}(\Gamma, f, sk)$*: is a probabilistic (or deterministic) algorithm that is run by the data owner. It takes an encrypted index* Γ*, a file* f *and a secret key* sk *as input, and outputs an encrypted index* Γ' *and an encrypted file* c*.*

A SE scheme builds an *encrypted index*, which enables keyword search over encrypted data through equality checking. Each file itself is protected with AES-CBC-256. The objective of algorithm Query is to find identifiers, i.e., pointers or memory addresses, of matched encrypted files for each keyword query. Once obtaining those identifiers, a server can easily return matched encrypted files. The correctness of a SE scheme can be formally described as

- *If* $w \in f_i$*:* $\mathsf{Query}(\Gamma, tk_w) = \mathbf{I}_w$*, where* $I_i \in \mathbf{I}_w$*;*
- *If* $w \notin f_i$*:* $\Pr[\mathsf{Query}(\Gamma, tk_w) = \mathbf{I}_w$*, where* $I_i \notin \mathbf{I}_w] \geq 1 - \mathtt{negl}(\lambda)$

where $\mathtt{negl}(\lambda)$ is a *negligible function* in λ.

Algorithm Update changes a current encrypted index Γ based on a file f, and output an updated encrypted index Γ'. Normally, this update operation covers adding a file, modifying a file, or removing a file from an encrypted index.

C. File-Injection Attacks

Access Pattern. To maintain the efficiency of a search function over encrypted data, a SE scheme reveals an *access*

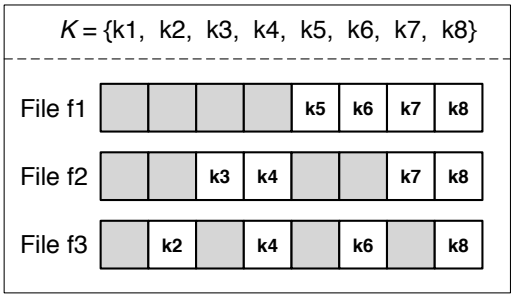

Fig. 2. A file-injection attack with 3 injected files, where f_1 includes $\{k_5, k_6, k_7, k_8\}$, f_2 includes $\{k_3, k_4, k_7, k_8\}$, f_3 includes $\{k_2, k_4, k_6, k_8\}$. An access pattern vector $(b_1, b_2, b_3) = (0, 1, 1)$ implies keyword k_4; an access pattern vector $(b_1, b_2, b_3) = (1, 1, 1)$ indicates keyword k_8.

pattern [2], [12], [25]. The access pattern leakage on a file can be represented as a single bit with 1 indicating retrieval. An access pattern of a search token on a number of β files can be described as a binary vector $\vec{b} = (b_1, ..., b_\beta)$. We refer to this vector $\vec{b}$ as an *access pattern vector*.

With recent attacks [9], [10], [25]–[27], an attacker defeats privacy through access pattern. For instance, by running a *file-injection attack* [9], an attacker reveals *query privacy*, which was meant to be protected by SE. Specifically, an attacker can learn a query in plaintext through a unique access pattern on a list of injected files.

Attack Assumptions. To perform a file-injection attack, a few necessary assumptions are on the system model and an attacker's *prior* knowledge. First, an attacker assumes that a data owner uses a SE scheme to protect private emails, where the data owner will automatically encrypt an email whenever it receives one and will immediately add the corresponding encrypted email to the encrypted index generated by SE. Besides observing access pattern on a server, an attacker is assumed to know the *keyword pool* (or dictionary) $\mathcal{K}$ and the number of keywords T of this pool in advance. Beyond these prior knowledge, *the attacker has no additional information*.

Attack Details. With the above assumptions, an attacker can initiate a (straightforward) file-injection attack as follows. It sends an email to the data owner each day, where each injected email contains a number of $T/2$ keywords. This attacker can choose those $T/2$ keywords in each injected email. After sending a number of $\lceil \log_2 T \rceil$ injected emails, the attacker reveals the query of any search token, since any search token maps to a unique access pattern vector on those $\lceil \log_2 T \rceil$ injected files. An example of a straightforward file-injection attack is illustrated in Fig. 2, where the keyword pool $\mathcal{K} = \{k_1, k_2, k_3, k_4, k_5, k_6, k_7, k_8\}$ and $T = 8$.

Since there are thousands of keywords in a keyword pool in general, it would be abnormal to have $T/2$ keywords in a single email/file in practice. A more realistic file-injection attack will set a threshold t as the maximum number of keywords in each injected-file [9], where $t < T/2$. For example, $T = 5,000$ and $t = 200$. To reveal the query of any search token, the total number of injected-files required is

$$\beta = \lceil T/t \rceil + \lceil T/2t \rceil \cdot \lceil \log_2(2t) \rceil \quad (1)$$

The basic idea of this attack considering this threshold t is to first use $\lceil T/t \rceil$ files to divide the entire keyword pool into $\lceil T/t \rceil$ subsets, where each subset has t keywords. Next, each two adjacent subsets can be grouped together to perform an instance of a straightforward file-injection attack, where the total number of keywords in an instance is $2t$ and each injected-file includes t keywords [9]. As a result, an access pattern vector on all the β injected files is able to uniquely define a keyword. Additional details can be found in [9].

D. Privacy Metric

In this paper, we propose to leverage a *privacy parameter* θ, which is defined as *the number of bits that are* ***indistinguishable*** *to a file-injection attacker in a β-bit access pattern vector*, as a privacy metric to validate the privacy of SE under a file-injection attack. The claim that a bit b in an access pattern vector is *indistinguishable* to an attacker means that

$$\Pr[b = 0] = \frac{1}{2}, \quad \Pr[b = 1] = \frac{1}{2}.$$

Based on this privacy parameter θ, we can further present privacy in terms of *an attacker's guessing probability*.

Definition 1. ***Attacker's Guessing Probability.*** *Given a number of θ indistinguishable bits in an access pattern vector $\vec{b} = (b_1, ..., b_\beta)$ obtained under a file-injection attack, the probability that an attacker correctly guesses a query in plaintext is $P = 1/2^\theta$.*

III. N-Gram and Cross Entropy

***N*-Gram.** An n-gram language model is one of the most common methods to predict the next word [20]. An n-gram language model takes a *corpus* as training data and builds a Markov model based on the probability distributions of sequences of words. This is specifically done for the next word in a sequence given the previous $n-1$ words,

$$\Pr(w_i | w_{i-(n-1)}, ..., w_{i-1}) = \frac{c(w_{i-(n-1)} ... w_{i-1} w_i)}{c(w_{i-(n-1)} ... w_{i-1} *)}$$

where $w_{i-(n-1)}, ..., w_{i-1}$ are the previous $n-1$ words preceding the word w_i, $c(w_{i-(n-1)} ... w_{i-1} w_i)$ denotes the frequency of sequence $w_{i-(n-1)}, ..., w_{i-1}, w_i$, and $*$ is a wildcard. Given a sequence of $n-1$ words, the sum of the probabilities of all the possible next words is 1,

$$\sum_{j=1}^{T} \Pr(w_{i,j} | w_{i-(n-1)}, ..., w_{i-1}) = 1$$

where T is the total number of words.

An example of a 2-gram model (a *bigram* model) is presented in Fig. 3. Specifically, given a word "`is`", there are two possible next word, "`a`" and "`not`", based on this corpus. The probability of the next word is "`a`" is $2/3$ and the probability of the next word is "`not`" is $1/3$. Once word sequence probabilities are learned from a corpus, an n-gram model can be used to predict the next word.

Cross Entropy. Based on this property, an n-gram can be utilized as a semantic filter to decide whether a given text is

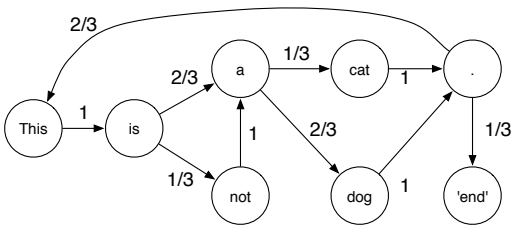

Fig. 3. An example of the Markov model based on a bigram, where the training corpus is {*This is a dog. This is not a dog. This is a cat.*}

Algorithm 1: SemFilter(f, h, MM, n)

Input: A text f, a threshold h, a Markov model MM, and parameter n
Output: A bit b

$(w_1, w_2, ..., w_N) \leftarrow$ Tokenize(f);
$H = 0$;
for $k = n; k \leq N; k++$ **do**
  $H = H + \log_2 \Pr(w_k | w_{k-(n-1)}, ..., w_{k-1})$;
end for
$H = -H/N$
if $H \geq h$ **then**
  return $b = 0$;
else
  return $b = 1$;
end if

similar in semantics [20], [21]. Specifically, assume a Markov model MM based on n-gram is trained by a set of semantic texts. Given a text f, we can compute *cross entropy* [21] between this text f and Markov model MM as,

$$H(MM, f) = -\frac{1}{N} \sum_{i=1}^{N} \log_2 \Pr(w_i | w_{i-(n-1)}, ..., w_{i-1}) \quad (2)$$

where N is the number of words in this text. Note that the above equation is an approximation of cross entropy, which is based on the Shannon-McMillan-Breiman Theorem and is often used in language modeling, since the true distribution of a language is unknown [28].

A higher cross entropy indicates text f is more "surprising" to the Markov model. It implies text f is not similar or relevant to the training corpus, which further indicates text f is likely not semantic. To avoid computing an undefined $\log_2 0$ in the calculation of cross entropy, we use a *smoothing* method [28], which redefines the probability of a next word as

$$\Pr(w_i | w_{i-(n-1)}, ..., w_{i-1}) = \frac{1 + c(w_{i-(n-1)} ... w_{i-1} w_i)}{T + c(w_{i-(n-1)} ... w_{i-1} *)}$$

where T is the total number of words in a corpus.

The pseudo code of calculating cross entropy for a given text f is described in Algo. 1, where we use Tokenize(f) to describe the process of segmenting a text f into a sequence of words. If the output of Algo. 1 is 1, it indicates input text f is similar to the training corpus and it is likely semantic.

IV. Vaccine: A Pre-Encryption Obfuscation

In this section, we present Vaccine, which preserves privacy of a SE scheme against a file-injection attack.

Semantics of Received Emails. Since an attacker in a file-injection attack does not write emails to a data owner for regular communication, but intentionally sends files with different subsets of keywords on purpose to reveal query privacy, one may think those injected files may not carry semantic texts, and could be easily identified by investigating the semantics of each received email. However, as pointed out in the original file-injection attacks [9], an attacker could generate semantic texts from a given subset of keywords using automate text generation [29]. In other words, it is *insufficient* to prevent file-injection attacks by directly examining the semantics of each received email.

A. Vaccine: Self File-Injection

To protect SE and mitigate privacy leakage under a file-injection attack, we propose a novel defense mechanism, referred to as *Vaccine*. The *main idea of Vaccine* is that, whenever a data owner receives an email, *either one sent by a legitimate user or injected by an attacker*, which the data owner could not distinguish, the data owner injects a file generated by itself. This self-injected file includes a set of random keywords that does not have intersection with the set of keywords in the received email. Both files will be encrypted by the data owner with SE and added to the server at the same time. As a result, this self file-injection will obfuscate access pattern on *some* of the files maliciously injected by an attacker, which effectively minimizes query privacy revealed to an attacker. Vaccine is a *pre-encryption obfuscation*. It is compatible with any keyword SE scheme.

Self file-injection can obfuscate access pattern, but will introduce redundancy (self-injected files) and affect the correctness in search results returned by SE. We propose to adopt a *semantic filter* at the data owner side to remove self-injected files from search results after this data owner decrypts the results locally. Self-injected files are designed with non-semantic texts, which can be distinguished by leveraging a semantic filter. Details of our scheme are further elaborated below. For ease of presentation, a *self-injected* file in the following indicates it is injected by a data owner, and an *injected* file indicates it is maliciously injected by an attacker.

Details of Vaccine. Vaccine includes two algorithms, including Inject and Remove. Inject introduces self-injected files for privacy protection, and Remove helps to remove self-injected files to ensure correct search results. The details of the two algorithms are illustrated in Fig. 4 and Fig. 5.

In Inject, when the data owner receives an email f with a number of t keywords, where this file is either sent by a legitimate user or injected by an attacker, this data owner randomly picks a number of t different keywords. Those t keywords are selected from the keyword pool $\mathcal{K}$ but are not shown in the received file f. By simply putting the t self-selected keywords together, the data owner outputs a *self-injected* file without

$\{\Gamma', c_1, c_2\} \leftarrow \mathsf{Inject}(f, \mathcal{K}, \Gamma, sk)$: Given a received file f, a keyword pool $\mathcal{K}$, an encrypted index Γ and a secret key $sk = \{sk_a, sk_s\}$, where received file f includes t keywords $\{k_1, ..., k_t\}$, the data owner

1) Selects t keywords $\{k'_1, ..., k'_t\}$, s.t.,

$$k'_j \xleftarrow{r} \mathcal{K}', \quad \mathcal{K}' = \mathcal{K} \setminus \{k_1, ..., k_t\}$$

where each k'_j, for $j \in [1, t]$, is uniformly chosen from set $\mathcal{K}'$;

2) Outputs a self-injected file f' s.t.,

$$\begin{cases} k^* \notin f', \quad \forall k^* \in \mathcal{K}^*, \quad \mathcal{K}^* = \mathcal{K} \setminus \{k'_1, ..., k'_t\} \\ \mathsf{AND} \quad k'_j \in f', \quad \forall j \in [1, t] \\ \mathsf{AND} \quad \lceil |f'|/256 \rceil = \lceil |f|/256 \rceil \end{cases}$$

3) Flips a fair coin, if Heads, runs

$$(\Gamma^*, c_1) \leftarrow \mathsf{SE.Update}(\Gamma, sk_s, f),$$
$$(\Gamma', c_2) \leftarrow \mathsf{SE.Update}(\Gamma^*, sk_s, f'),$$

otherwise runs

$$(\Gamma^*, c_1) \leftarrow \mathsf{SE.Update}(\Gamma, sk_s, f'),$$
$$(\Gamma', c_2) \leftarrow \mathsf{SE.Update}(\Gamma^*, sk_s, f),$$

4) Outputs $\{\Gamma', c_1, c_2\}$.

Fig. 4. Details of Inject in Vaccine.

putting any effort of making it semantically meaningful. In addition, the data owner matches the size of this self-injected file f' with the size of this received file f by repeating those t self-selected keywords, such that the two files have a same number of blocks after encrypting with AES-CBC-256, i.e., $\lceil |f'|/256 \rceil = \lceil |f|/256 \rceil$. An example of a semantic email and its corresponding self-injected email is described in Fig. 6.

Next, the data owner adds the self-injected file f' and the received file f to the encrypted index on the server. One fair coin is flipped to decide which file between f and f' will be added first. The output of Inject includes an updated encrypted index Γ' and two encrypted files c_1, c_2. Since an attacker is not able to distinguish which file is maliciously injected by itself between the two files, it will fail to indicate correct access pattern on the maliciously-injected file, *if the access pattern on the two encrypted files are different*. As a result, given a search token and corresponding access pattern, an attacker's probability of indicating a correct query in plaintext will be significantly *limited*, which ultimately improves privacy protection of a SE scheme under a file-injection attack.

When a data owner searches a keyword query w on its encrypted index, the server returns a set of encrypted files. After decrypting those files, the data owner can obtain files, e.g., $\mathbf{f}'_w = \{f'_{w,1}, ..., f'_{w,m}\}$, in plaintext. Some of those files were previously injected by this data owner itself and do not include semantic texts. The data owner can run Remove algorithm to decide which ones are self-injected files and remove them by using a semantic filter described in Sec. II.

$\mathbf{f}_w \leftarrow \mathsf{Remove}(\mathbf{f}'_w, h, MM, n)$: Given a set of files $\mathbf{f}'_w = (f'_{w,1}, ..., f'_{w,m})$, a threshold h, a Markov model MM, and a parameter n, the data owner

1) Sets $\mathbf{f}_w = \varnothing$
2) For each $f'_{w,i}$, for $i \in [1, m]$, runs

$$b'_i \leftarrow \mathsf{SemFilter}(f'_{w,i}, h, MM, n)$$
$$\textit{if } b'_i == 1 \quad \mathbf{f}_w \leftarrow \mathbf{f}_w \cup f'_{w,i}$$

3) Outputs $\mathbf{f}_w$.

Fig. 5. Details of Remove in Vaccine.

I don't think I will be able to make our meeting tomorrow. Can we pick it up again next week? Thanks.

larsen referenced beck busy pacific cora koch smoking lone fiction Amelia money message frame davis pinnacle bottom vol export Lynch

Fig. 6. An example of a legitimate email and its self-injected email.

B. Discussions

According to the design of Vaccine, if an attacker could replay a *same* injected file multiple times, where a data owner would self-inject multiple *different* files. The access pattern of a search token on those same malicious injected files would be always consistent, but its access pattern over those self-injected files would be various. This may allow this attacker to bypass the privacy protection provided by Vaccine. One such example is illustrated in Fig. 7.

However, this potential threat can be easily suppressed by ignoring any repeated email and only adding unique emails to an encrypted index. More importantly, if an attacker keeps sending same injected emails, which is extremely abnormal, the data owner can easily detect a file-injection attack. In practice, a Bloom filter [30] can be leveraged at the data owner side to test whether a received email is a duplicate one, where a Bloom filter only takes several kilobytes to maintain.

An attacker could also reply similar emails with a same set of injected keywords in order to bypass the protection of Vaccine. A data owner can eliminate these emails prior to encryption and identifying this misbehavior by checking the set of keywords using a Bloom filter. Although a data owner could receive some similar emails with a set of keywords occasionally, keeping receiving similar emails with a same set of keywords still suggests misbehavior of an attacker.

For instance, among the $30,109$ emails in Enron dataset, only 5% of the emails do not have a unique set of keywords. In addition, since an attacker also needs to fix the number of keywords in each injected file to make the attack efficient, there are at most 0.6% emails do not have a unique set of keywords given a fixed number of keywords. Obviously, keeping injecting similar emails with a same set of keywords would be suspicious in practice. Besides, replaying injected emails would significantly reduce the attack effectiveness.

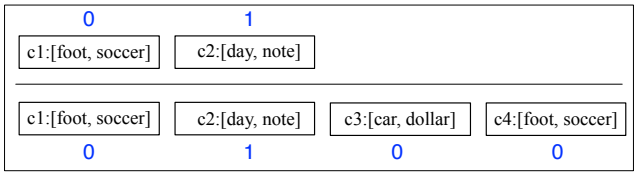

Fig. 7. If an attacker only injects a file f, Vaccine generates a self-injected file f_1' and outputs c_1, c_2. If access pattern on c_1 and c_2 are different, then Vaccine preserves privacy. If this attacker injects file f again, Vaccine generates a self-injected file f_2' and outputs c_3, c_4. If access pattern on c_3 and c_4 are the same, then Vaccine does not preserve privacy even though access pattern on c_1 and c_2 are different.

One may also think simply including a special keyword in each self-injected file as a unique indicator is sufficient to identify self-injected files from a search result, and there is no need to leverage a semantic filter to remove self-injected files. Unfortunately, this naive approach would compromise the privacy protection introduced by Vaccine. Specifically, if each self-injected file includes a same keyword as a unique indicator, then each self-injected file is indexed by the same keyword in the encrypted index while maliciously-injected files are not. As a result, an attacker could distinguish between a maliciously-injected file and a self-injected file by observing the encrypted index and access pattern on the server side.

V. Privacy and Tradeoff Analysis

Privacy Analysis. Vaccine can obfuscate some of the bits in an access pattern vector, which mitigates privacy leakage. We validate the formal privacy of Vaccine by using the number of distinguishable bits in an access pattern vector.

Theorem 1. *With Vaccine, the number of bits in an access pattern vector that are indistinguishable to an attacker is*

$$\theta = \alpha + (\beta - \alpha) \cdot \frac{t}{T - t} \tag{3}$$

where T is the total number of keywords, t is the number of keywords in each injected file, β is the number of bits in each access pattern vector, α is the number of 1s in an access pattern vector over the files injected by an attacker.

Proof. For a file f_i injected by an attacker, the data owner generates a self-injected file f_i' using Vaccine. Since the keywords selected in f_i' have *no intersection* with the keywords in f_i, if a token of a query keyword returns f_i, then it will certainly not return f_i'. For ease of analysis, we ignore the negligible probability of returning f_i' in a SE scheme.

If a token outputs an access pattern 1 on f_i, it will certainly generate a different access pattern 0 on f_i'. Because f_i and f_i' are added to the encrypted index successively with a random order in Vaccine, an attacker cannot distinguish the access pattern on f_i and f_i', which means it will not be able to distinguish bit b_i in an access pattern vector $\vec{b}$. Therefore, if there are originally α 1s in an access pattern vector over all the files injected by an attacker, those α bits are indistinguishable to an attacker after applying Vaccine.

On the other hand, if access pattern on f_i is 0 for a given token, access pattern on f_i' could be 0 or 1. If it is still 0, it does not offer additional privacy protection. However, if it is 1, it mitigates privacy leakage. Assume each keyword k_j' in f_j' is selected uniformly from $\mathcal{K}'$, where $\mathcal{K}' = \mathcal{K} \setminus \{k_1, ..., k_t\}$, for a random search token, the probability that access pattern on f_i' is 1 given access pattern on f_i is 0 is $\frac{t}{T-t}$. Therefore, if there are originally $(\beta - \alpha)$ 0s bits in an access pattern vector over all the files injected by an attacker, then the *average* number of bits among those $(\beta - \alpha)$ bits would be different after leveraging Vaccine is $(\beta - \alpha) \cdot \frac{t}{T-t}$. Overall, the number of bits in an access pattern vector that an attacker is not able to distinguish after using Vaccine is $\theta = \alpha + (\beta - \alpha) \cdot \frac{t}{T-t}$ □

Next, we further analyze $E[\theta]$, i.e., the *expected* number of distinguishable bits in an access pattern vector. First, we analyze the expected value of α.

Lemma 1. *The expected number of 1s in an access pattern vector $\vec{b} = (b_1, ..., b_\beta)$ over files injected by an attacker is*

$$E[\alpha] = 1 + \frac{1}{2} \cdot \lceil \log_2(2t) \rceil \tag{4}$$

where $\beta = \lceil T/t \rceil + \lceil T/2t \rceil \cdot \lceil \log_2(2t) \rceil$, T is the total number of keywords in keyword pool, and t is the number of keywords in each injected file.

Proof. According to a file-injection attack, $\beta = \lceil T/t \rceil + \lceil T/2t \rceil \cdot \lceil \log_2(2t) \rceil$, where first $\lceil T/t \rceil$ bits decide which subset a keyword belongs and the rest of $\lceil T/2t \rceil \cdot \lceil \log_2(2t) \rceil$ bits determine which exact keyword it is. Therefore, there is one bit among the first $\lceil T/t \rceil$ bits that is 1. The *expected* number of 1s among the rest of $\lceil T/2t \rceil \cdot \lceil \log_2(2t) \rceil$ bits is $\frac{1}{2} \cdot \lceil \log_2(2t) \rceil$. □

Based on Theorem 1 and Lemma 1, we have

Lemma 2. *With Vaccine, the expected number of indistinguishable bits in an access pattern vector $\vec{b} = (b_1, ..., b_\beta)$ is*

$$E[\theta] = 1 + \frac{t}{T - t} \cdot (\lceil T/t \rceil - 1) + \lceil \log_2 2t \rceil \tag{5}$$

where $\beta = \lceil T/t \rceil + \lceil T/2t \rceil \cdot \lceil \log_2(2t) \rceil$, T is the total number of keywords in keyword pool, and t is the number of keywords in each injected file.

Proof. The correctness of this lemma can be proved below

$$\begin{aligned}
E[\theta] &= E\left[\alpha + (\beta - \alpha) \cdot \frac{t}{T-t}\right] \\
&= E[\alpha] + E\left[\beta \cdot \frac{t}{T-t}\right] - E\left[\alpha \cdot \frac{t}{T-t}\right] \\
&= 1 + \frac{1}{2}\lceil \log_2(2t) \rceil + \lceil T/2t \rceil \cdot \lceil \log_2(2t) \rceil \cdot \frac{t}{T-t} \\
&\quad + \lceil T/t \rceil \cdot \frac{t}{T-t} - \left(1 + \frac{1}{2}\lceil \log_2(2t) \rceil\right) \cdot \frac{t}{T-t} \\
&= 1 + \frac{t}{T-t} \cdot (\lceil T/t \rceil - 1) + \lceil \log_2 2t \rceil
\end{aligned}$$

□

According to Def. 1, an adversary's guessing probability is $1/2^\theta$ if access pattern is completely protected on a number

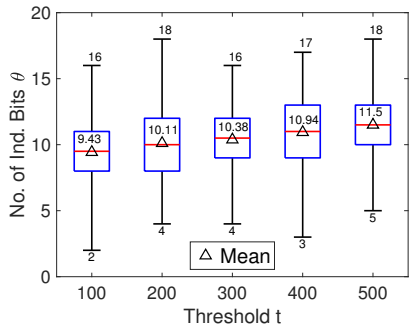

Fig. 8. The impact of threshold t on the number of indistinguishable bits, where $T = 5000$.

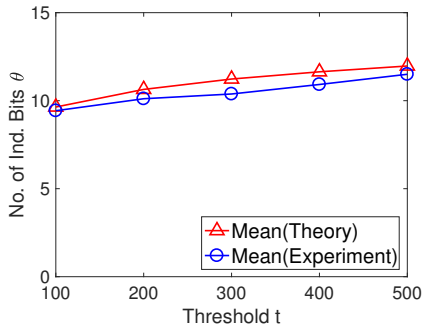

Fig. 9. The impact of threshold t on the number of indistinguishable bits, where $T = 5000$.

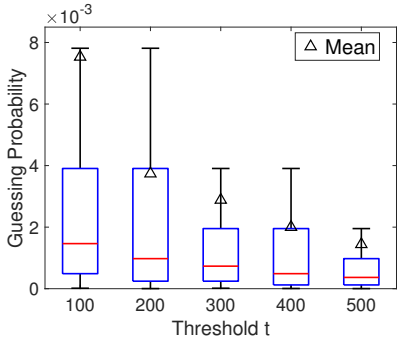

Fig. 10. The impact of threshold t on an adversary's guessing probability, where $T = 5000$.

of θ files. Assume the number of indistinguishable bits θ is a random variable, where $\theta \in [0, \beta]$, and we use $(\theta_0, \theta_1, ..., \theta_\beta) = (0, 1, ..., \beta)$ to denote all the unique values of θ. We also use ϕ_i to denote the frequency of the number of indistinguishable bits is θ_i, where $i \in [0, \beta]$, among N access pattern vectors for N keywords. Then with Vaccine, the adversary's expected guessing probability of each keyword given those N access pattern vectors under a file-injection attack can be computed as

$$E[P] = \sum_{i=0}^{\beta} \frac{1}{2^{\theta_i}} \cdot \frac{\phi_i}{N} = \frac{1}{N} \sum_{i=0}^{\beta} \frac{\phi_i}{2^{\theta_i}} \tag{6}$$

Tradeoffs in Vaccine. In Vaccine, whenever a data owner receives an email, it will output a self-injected email. The storage spent on encrypted index and encrypted files will increase by a factor of 2 *in theory*.

However, the above tradeoffs can be optimized using different methods. For example, emails from people in a white list (e.g., friends and colleagues) are trustworthy and are less likely to be injected emails. In addition, if there are subsequent emails exchanged between a data owner and a sender, those emails do not follow the algorithm of a file-injection attack, and therefore are less likely to be injected emails. A data owner can opt for not generating self-injected emails in those cases. For instance, among the $30,109$ emails from Enron dataset, 71.5% of those emails are internal emails exchanged between two `enron.com` addresses. If those emails are believed to be trustworthy, then only $8,580$ self-injected emails are needed, which only increases the storage by 1.29X.

VI. Performance

Enron Dataset We test the performance of Vaccine on a real-world dataset consisting of the Enron emails [22]. This dataset includes emails from 150 employees of the Enron Corporation, and it has been used in recent works [9], [10] to study access pattern leakage. In our experiments, we preprocess the emails from the Enron dataset using the Natural Language Toolkit (NLTK) [31]. Following others' efforts, we collect emails from each user's *"_sent_mail"* folder. There are 30,109 emails from the *"_sent_mail"* folders of all the users.

To obtain a keyword pool for the $30,109$ emails, we first sanitize the email text by segmenting it into words and removing all numbers and special characters. Additionally, we tag each word from those emails using Part-Of-Speech tagging and lemmatize all words by using the WorldNet lemmatizer [20]. Then, we remove words, e.g., "and", "an", etc., that normally would not be used as keywords.

As a result, we collect 89,788 unique words from the Enron corpus. As previous works, we take the top 5,000 frequent words as the keyword universe $\mathcal{K}$, where keyword "enron" is the most frequent keyword with a frequency of 144,316. Note that the number of unique words (i.e., 89,788) we obtained is not exactly the same as the one (i.e., 77,000) in previous works [9], [10]. It is mainly because we implement the same methodology but with different functions and libraries. For instance, we leverage the WorldNet lemmatizer while the Porter stemmer is utilized in [10].

Experiment Setting. We implement file-injection attacks and Vaccine with around $2,000$ lines of code in Python 2.7, and test our code on a MacBook Pro running High Sierra 10.13 with 2.5GHz Intel Core i5 and 12GB memory.

Privacy. We first test the number of indistinguishable bits introduced by Vaccine. Specifically, given the total number of keywords $T = 5000$ and a different value of t, we assume that an attacker can generate β injected files for a file-injection attack, where β is computed based on Eq. 1 and keywords in each file are selected by following the attack algorithm. For each injected file, Vaccine generates a self-injected file.

For each t, we randomly select $N = 100$ keywords from the keyword pool, record its access pattern on the β injected files and β self-injected files respectively, and calculate the number of indistinguishable bits of each keyword, i.e., the number of different bits between the one from injected files and the one from self-injected files. The results are presented in Fig. 8. When $t = 200$ and $T = 5000$, the average number of indistinguishable bits is 10.11 in an access pattern vector, where each access pattern vector has $\beta = 142$ bits. Fig. 8 also implies the average number of indistinguishable bits slightly increases with an increase on threshold t.

We also calculate the theoretical result of the number of indistinguishable bits based on Lemma 2, and compare it with our experimental results in Fig. 9. We can observe that the experimental results of the number of indistinguishable bits are consistent with our analysis in Lemma 2. Based on the results from Fig. 8, we also calculate an adversary's guessing probability by using Eq. 6 and present it in Fig. 10. For

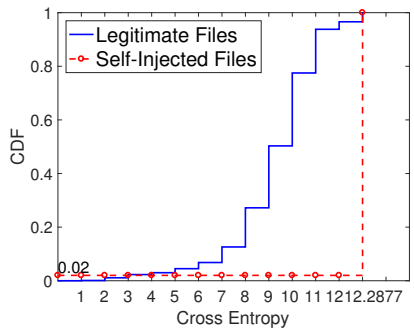

Fig. 11. The Cumulative Distribution Function of the cross entropy over the training data.

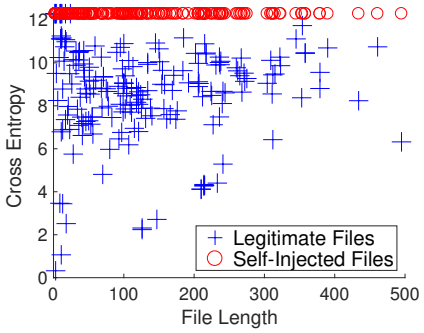

Fig. 12. The distribution of the training data by considering cross entropy and file length.

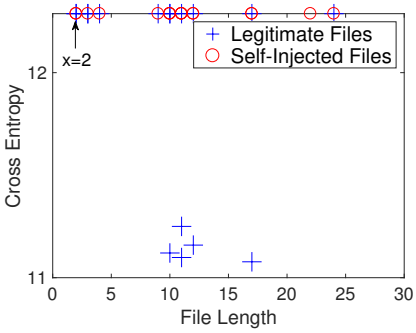

Fig. 13. The upper left corner of Fig. 12 (**3-gram** with **10%** Enron corpus.).

instance, given $t = 200$ and $T = 5000$, an adversary can reveal a correct keyword with a probability of 3.7×10^{-3} on average, which is significantly lower compared to a guessing probability of 1 without exploiting Vaccine.

Tradeoffs. Self-injected files are helpful to mitigate privacy leakage under a file-injection attack. On the other hand, they need to be removed in search results. To build a semantic filter to remove self-injected files, we assume the data owner already has 10% of Enron corpus, and can train a Markov model with this 10% of Enron corpus using n-gram, where $n = 3$ by default. This Markov model only takes 3 MBs of local storage and takes 2.03 seconds to train the model for the data owner.

Next, we randomly select 2,000 legitimate emails from the Enron dataset and generate 2,000 corresponding self-injected files for those legitimate emails. Those 2,000 selected legitimate emails do not have overlaps with the 10% Enron corpus that the data owner used for training the Markov model. Among those 2,000 legitimate emails and 2,000 self-injected files, we take 1,000 legitimate emails and their 1,000 self-injected files as a training set, and leave the other 1,000 legitimate emails and their self-injected files as a testing set.

For each file f in the training set, we run SemFilter with f and the Markov model to obtain cross entropy H. The CDF of this cross entropy over the training set is described in Fig. 11. Almost all the self-injected files have the highest cross entropy, which is $H = -\log_2 \frac{1}{T} \approx 12.2877$ given $T = 5000$. On the other hand, only 34 of legitimate emails in the training set has a cross entropy of 12.2877. Therefore, if we simply leverage the following threshold as a classifier

$$\hat{l} = \begin{cases} 0, & \textit{if} \quad H = -\log_2 \frac{1}{T} \\ 1, & \textit{if} \quad H < -\log_2 \frac{1}{T} \end{cases}$$

where $\hat{l} = 1$ indicates a semantic file, it introduces no false positives and 3.4% false negatives. We also explore Naive Bayes classifier with cross entropy, which introduces same false positives and false negatives as shown in Table I.

Besides assessing cross entropy, if we also investigate the length of each file in the training set (as shown in Fig. 12), we observe that false negatives in the threshold classifier are introduced when the length of an email is relatively small. For a false negative, it indicates that a legitimate email is semantically meaningful, but since its text is entirely *irrelevant* and *surprising* to the Markov model trained from the 10% of Enron corpus, and a cross entropy of 12.2877 is reported. Increasing the size of training corpus of the Markov model can further reduce false negatives. For example, if the Markov model is trained based on 20% of Enron corpus, then false negatives can be reduced to 3%.

TABLE I
FPs & FNs WITH 3-GRAM BASED ON **10%** ENRON CORPUS.

Classifiers	False Positives	False Negatives
Threshold	0%	3.4%
Naive Bayes	0%	3.4%
k-Nearest Neighbor ($k = 5$)	1.8%	1.8%

TABLE II
FPs & FNs WITH 3-GRAM BASED ON **20%** ENRON CORPUS.

Classifiers	False Positives	False Negatives
Threshold	0%	3%
Naive Bayes	0%	3.4%
k-Nearest Neighbor ($k = 5$)	1%	2.4%

TABLE III
FPs & FNs WITH 4-GRAM TRAINED BASED ON **10%** ENRON CORPUS.

Classifiers	False Positives	False Negatives
Threshold	0%	5.9%
Naive Bayes	0%	5.9%
k-Nearest Neighbor ($k = 5$)	0.1%	6.2%

In Fig. 13, some false negatives occur as the file length is smaller than 3. It is because when file length is smaller than the value of $n = 3$ in n-gram a probability of the next word is undefined. We assign the probability as $\frac{1}{T}$ using adding one smoothing [21], which leads to a cross entropy of $-\log_2 \frac{1}{T}$.

In addition to threshold and Naive Bayes, we also apply k-nearest neighbor classification by considering both cross entropy and file length, and reach 1.8% false positives and 1.8% false negatives on the testing set, where $k = 5$ as shown in Table I. In Table III, we also show that if we use 4-gram instead of 3-gram, then the false negatives increase. The main reason is that more legitimate emails are irrelevant and surprising to the Markov model trained by 4-gram.

VII. Related Work

Attacks on SE. Besides file-injection attacks [9], *query recovery attacks* proposed in [10], [25] are also able to recover

the keyword query of a search token. In the original query recovery attack proposed by Islam et al. [25], an attacker is assumed to have prior knowledge of keyword co-occurrence matrix M, where $m_{i,j}$ in this matrix represents the probability that keyword w_i and keyword w_j appear in a file f. By additionally observing access pattern and collecting a corresponding keyword co-occurrence matrix M', an adversary maps keywords with search tokens using an optimization algorithm. Cash et al. [10] improved this attack by assuming the number of files associated with each keyword is also known by the adversary. Compared to a query recovery attack, which is a *passive attack*, a file-injection attack is an *active* attack and does not require prior knowledge of the keyword co-occurrence matrix. Another passive attack, named *reconstruction attack* [26], [27], particularly targets SE schemes running range queries and reveals query privacy through access pattern, if range queries are uniformly distributed.

Countermeasures. Exploiting Oblivious RAM can hide access pattern. However, efficiently implementing Oblivious RAM in SE is challenging [12], [13]. SE schemes [14]–[19] supporting *forward security*, can mitigate leakage if a search token is submitted before the completion of a file injection attack, but still completely reveal query privacy for any search token submitted after the attack.

Chen et al. [32] proposed a framework to specifically obfuscate access pattern under query recovery attacks by using *d-privacy*. Each file is divided into m shares (or called *shards*) using (k, m)-erasure coding, and both false negatives and false positives are introduced on each of the m shares before encrypting with SE. However, this framework [32] does not address privacy leakage against a file-injection attack. Since each malicious file is injected separately, an attacker can distinguish access pattern on each injected file even this framework is applied, and still reveal query privacy.

Quan et al. [33] designed a pre-encryption obfuscation method to mitigate privacy leakage against *range injection attacks* by leveraging randomized response. Unfortunately, this method does not render privacy enhancement against file-injection attacks.

VIII. Conclusion

We design a novel pre-encryption obfuscation mechanism to obfuscate access pattern on searchable encrypted data against file-injection attacks. By evaluating the similarity of texts in emails, our mechanism introduces no false positives and minimal false negatives to search results.

References

[1] D. Song, D. Wagner, and A. Perrig, "Practical Techniques for Searches on Encrypted Data," in *Proc. of IEEE S&P'00*, 2000.

[2] R. Curtmola, J. A. Garay, S. Kamara, and R. Ostrovsky, "Searchable Symmetric Encryption: Improved Definitions and Efficient Constructions," in *Proc. of ACM CCS'06*, 2006.

[3] S. Kamara, C. Papamanthou, and T. Roeder, "Dynamic Searchable Symmetric Encryption," in *Proc. of ACM CCS'12*, 2012, pp. 965–976.

[4] D. Cash, S. Jarecki, C. Jutla, H. Krawczyk, M.-C. Rosu, and M. Steiner, "Highly-Scalable Searchable Symmetric Encryption with Support for Boolean Queries ," in *Proc. of CRYPTO'13*, 2013.

[5] V. Pappas, F. Krell, B. Vo, V. Kolesnikov, T. Malkin, S. G. Choi, W. George, A. Keromytis, and S. Bellovin, "Blind Seer: A Searchable Private DBMS," in *Proc. of IEEE S&P'14*, 2014.

[6] R. A. Popa, C. M. Redfield, N. Zeldovich, and H. Balakrishnan, "CryptDB: Protecting Confidentiality with Encrypted Query Processing," in *Proc. of ACM SOSP'11*, 2011.

[7] "Google encrypted bigquery client." [Online]. Available: https://github.com/google/encrypted-bigquery-client

[8] "Microsoft always encrypted (database engine)." [Online]. Available: https://docs.microsoft.com/en-us/sql/relational-databases/security/encryption/always-encrypted-database-engine?view=sql-server-2017

[9] Y. Zhang, J. Katz, and C. Papamanthou, "All Your Queries Are Belong to Us: The Power of File-Injection Attacks on Searchable Encryption," in *USENIX Security*, 2016, pp. 707–720.

[10] D. Cash, P. Grubbs, J. Perry, and T. Ristenpart, "Leakage-Abuse Attacks Against Searchable Encryption," in *Proc. of CCS'15*, 2015.

[11] O. Goldreich, "Towards a Theory of Software Protection and Simulation by Oblivious RAMs," in *Proc. of ACM STOC'87*, 1987.

[12] B. Fuller, M. Varia, A. Yerukhimovich, E. Shen, A. Hamlin, V. Gadepally, R. Shay, J. D. Mitchell, and R. K. Cunningham, "SoK: Cryptographically Protected Database Search," in *Proc. of IEEE S&P'17*, 2017.

[13] M. Naveed, "The Fallacy of Composition of Oblivious RAM and Searchable Encryption," https://eprint.iacr.org/2015/668.pdf.

[14] E. Stefanov, C. Papamanthou, and E. Shi, "Practical Dynamic Searchable Encryption with Small Leakage," in *Proc. of NDSS'14*, 2014.

[15] R. Bost, "Sophos: Forward Secure Searchable Encryption," in *Proc. of ACM CCS'16*, 2016.

[16] A. A. Yavuz and J. Guajardo, "Dynamic Searchable Symmetric Encryption with Minimal Leakage and Efficient Updates on Commodity Hardware," in *Proc. of SAC'15*, 2015.

[17] K. S. Kim, M. Kim, D. Lee, J. H. Park, and W.-H. Kim, "Forward Secure Dynamic Searchable Symmetric Encryption with Efficient Updates," in *Proc. of ACM CCS'17*, 2017.

[18] R. Bost, B. Minaud, and O. Ohrimenko, "Forward and Backward Private Searchable Encryption from Constrained Cryptographic Primitives," in *Proc. of ACM CCS'17*, 2017.

[19] M. O. Ozmen, T. Hoang, and A. A. Yavuz, "Forward-Private Dynamic Searchable Symmetric Encryption with Efficient Search," in *Proc. of IEEE ICC'18*, 2018.

[20] S. Bird, E. Klein, and E. Loper, *Natual Language Processing with Python*. O'REILLY, 2009.

[21] A. Hindle, E. T. Barr, Z. Su, M. Gabel, and P. Devanbu, "On the Naturelness of Software," in *Proc. of IEEE Internatinoal Conference on Software Engineering (ICSE)*, 2012.

[22] "Enron dataset." [Online]. Available: https://www.cs.cmu.edu/~enron/

[23] M. Naveed, S. Kamara, and C. V. Wright, "Inference Attacks on Property-Preserving Encrypted Databases," in *Proc. of ACM CCS'15*, 2015.

[24] P. Grubbs, T. Ristenpart, and V. Shmatikov, "Why Your Encrypted Database is Not Secure," in *Proc. of HotOS'17*, 2017.

[25] M. S. Islam, M. Kuzu, and M. Kantarcioglu, "Access Pattern Disclosure on Searchable Encryption: Ramification, Attack and Mitigation," in *Proc. of NDSS'12*, 2012.

[26] Georgios Kellaris and George Kollios and Kobbi Nissim and Adam O'Neil, "Generic Attacks on Secure Outsourced Databases," in *Proc. of ACM CCS'16*, 2016.

[27] M.-S. Lacharite, B. Minaud, and K. G. Paterson, "Improved Reconstruction Attacks on Encrypted Data Using Range Query Leakage," in *Proc. of IEEE S&P'18*, 2018.

[28] C. D. Manning and H. Schutze, *Foundations of Statistical Natural Language Processing*. MIT Press, 1999.

[29] Y. Yao, B. Viswanath, J. Cryan, H. Zheng, and B. Y. Zhao, "Automated Crowdturfing Attacks and Defenses in Online Review Systems," in *Proc. of ACM CCS'17*, 2017.

[30] B. H. Bloom, "Space/Time Trade-offs in Hash Coding with Allowable Errors," *Communications of the ACM*, vol. 13, no. 7, pp. 422–426, 1970.

[31] "Natural language toolkit." [Online]. Available: https://www.nltk.org/

[32] G. Chen, T.-H. Lai, M. K. Reiter, and Y. Zhang, "Differentially Private Access Patterns for Searchable Symmetric Encryption," in *Proc. of IEEE INFOCOM'18*, 2018.

[33] H. Quan, H. Liu, B. Wang, M. Li, and Y. Zhang, "Randex: Mitigaing Range Injection Attacks on Searchable Encryption," in *Proc. of IEEE CNS'19*, 2019.