PEKSrand: Providing Predicate Privacy in Public-key Encryption with Keyword Search

Benwen Zhu*, Bo Zhu*, Kui Ren†
*Concordia Institute for Information Systems Engineerings(CIISE)
Concordia University
(be_zh, zhubo)@encs.concordia.ca

†Department of Electrical and Computer Engineering
Illinois Institute of Technology
kren@ece.iit.edu

***Abstract*—Recently, Shen, Shi, and Waters introduced the notion of *predicate privacy*, i.e., the property that $t(x)$ reveals no information about the encoded predicate p, and proposed a scheme that achieves predicate privacy in the symmetric-key settings. In this paper, we propose two schemes. In the first scheme, we extend PEKS to support predicate privacy based on the idea of randomization. To the best of our knowledge, this is the first work that ensures predicate privacy in the public-key settings without requiring interactions between the receiver and potential senders, the size of which may be very large. Moreover, we identify a new type of attacks against PEKS, i.e., statistical guessing attacks. Accordingly, we introduce a new notion called *statistics privacy*, i.e., the property that predicate privacy is preserved even when the statistical distribution of keywords is known, and propose a scheme that makes a tradeoff between statistics privacy and storage efficiency (of the delegate). According to our analysis and experimental results, compared to PEKS, both of our schemes introduce reasonable additional communication and computation overheads and can be smoothly deployed in existing systems.**

I. Introduction

Public-key Encryption with Keyword Search (PEKS) introduced by Boneh et al. [5] is the first practical asymmetric searchable encryption scheme, as well as the first predicate encryption scheme. It is originally designed for the purpose of intelligent email routing. For example, as shown in Figure 1, a user R may receive emails through different devices, e.g., a PDA or a desktop at the office. Hence, she/he may want to selectively forward emails with certain keywords to a specific device, e.g., emails that contain the word "agenda" are forwarded to the PDA. To protect data confidentiality, emails are encrypted at the sender side. Hence, the mail server G has no access to the content of emails. To delegate G the capability of performing selective forwarding, however, G is assigned a set of trapdoors that are corresponding to keywords that might be used for searching at a later time. For a keyword x, the corresponding trapdoor $t(x)$ is generated from the master secret held only by R and is used to define a predicate p. Upon receiving a ciphertext, G can verify whether the corresponding plaintext is equal to x based on the predicate.

Most previous works on predicate encryption concentrate on *plaintext privacy*, i.e., the property that ciphertexts reveal no information about the encrypted data to any party without the private key other than what is inherently revealed by the trapdoors. However, researchers also identified a few other security/privacy issues relevant to PEKS [3], [7], [1], [15]. One major concern is to limit the delegate's capability of keyword searching within a certain time frame [3], [1]. Another important concern is that PEKS is subject to offline keyword guessing attacks firstly identified by Byun et al [7]. Later, Shen, Shi, and Waters formalized the second concern and introduced the notion of *predicate privacy* [15], i.e., the property that $t(x)$ reveals no information about the encoded predicate p. They also proposed a predicate encryption scheme that can achieve both plaintext privacy and predicate privacy in the symmetric-key settings. Moreover, Shen, Shi, and Waters claimed that it is inherently impossible to achieve predicate privacy in the public-key setting, such as PEKS. Interesting though, several researchers had actually proposed a few solutions to this problem [3], [1]. However, Shen, Shi, and Waters's claim may be based on an implicit assumption that the proposed solution should not conflict with one of the aims of PEKS, i.e., making keyword search possible without interaction between the sender and receiver, which was indicated by Baek et al [3]. Such an assumption is definitely reasonable, since in practice the size of potential senders could be a huge number. Moreover, the proposed solutions [3], [1] require to share some secret, in the form of either a set of public keys of the receiver [1] or the method of refreshing keywords [3], [1], between senders and the receiver. Considering the huge number of potential senders, the overhead of synchronizing the secret and protecting it from disclosure is overwhelming.

Formally, the predicate p of the equality test in PEKS can be defined as $p(e(x), t(x)) = 1$, in which x is the keyword, $e(x)$ is the encryption of x, and $t(x)$ is the trapdoor derived from x and the private key held only by R. In reality, the mail server G is usually considered as a semi-trusted entity, which is honest-but-curious [8]. Since the public-key encryption function does not require a secret key, G can encrypt any plaintext of her choice and then evaluate the resulting ciphertext with the

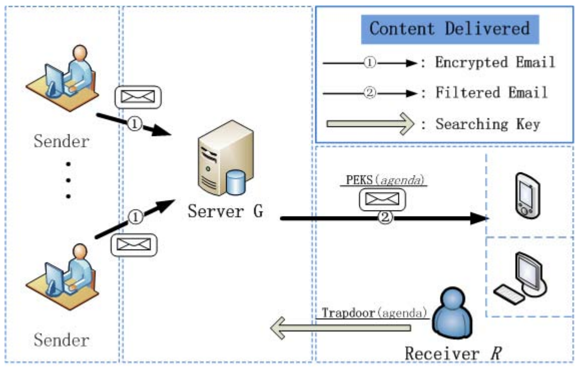

Fig. 1. Public Key Encryption with Keyword Search Framework

trapdoors assigned by R. By verifying whether the resulting ciphertext satisfies the predicate associating with a trapdoor, G learns whether the chosen plaintext is equal to the keyword that is corresponding to the trapdoor. In other words, the mail server G can launch an attack similar to brute-force password attacks. In particular, PEKS is especially fragile to this type of attacks in those applications, in which there exists a small set of keywords that are frequently used, such as "Urgent" and "Classified". In this paper, we refer this type of attacks as *brute-force guessing attacks*.

Our solution to brute-force guessing attacks are based on two basic ideas. One is to introduce randomness into the procedure of generating trapdoors so as to avoid the deterministic one-to-one mapping between a searching keyword and the corresponding trapdoor. It is also the underlying idea of previous solutions [3], [1]. The other is to limit the knowledge of the secret introducing randomness to only one or a small set of entities. Through creating random instances of keywords, the actual plaintexts used in the generation of trapdoors are known only to the entities that know the secret used in the randomization. Hence, in terms of the intelligent email routing, the semi-trusted mail server G cannot launch brute-force guessing attacks any more. On the other hand, by limiting the holders of the secret, the overhead of synchronizing and protecting the secret is much smaller.

Besides brute-force guessing attacks, an alternative attack that may be launched by the mail server is to make use of external knowledge about the statistical distribution of keywords to identify the relation between a trapdoor/predicate and a keyword. We call it as a *statistical guessing attack*. For example, in an application the probabilities that a keyword x and any other keyword are matched are 20% and no more than 10%. In such a case, the mail server can easily deduce the trapdoor corresponding to x by simply counting the number of times that each trapdoor is matched. Note that, the method of refreshing keywords by appending the time period to the keyword before the encryption is still subject to this type of attacks, because it does not change the statistical distribution of keywords within the same time period. Informally, the idea of our solution to statistical guessing attacks is to spread out the statistical distribution of keywords. In the previous example, it is much more difficult for the mail server to guess when the probabilities that a trapdoor mapped to keyword x and a trapdoor mapped to any other keyword are matched are 2% and no more than 1%, given that the same number of keyword matching events are observed. Apparently, when the mail server observes a sufficient number of events, it can still figure out the mappings between trapdoors and keywords. Hence, we have to refresh such mappings before it happens. In this paper, we provide detailed analysis on identifying an appropriate balance between privacy and efficiency.

In this paper, we proposed the PEKSrand scheme to provide strong privacy protection in PEKS. PEKSrand has two variants: *PEKSrand-BG* and *PEKSrand-SG*. Both variants are robustly against brute-force guessing attacks, and thus can ensure predicate privacy in the scenarios where the statistical distribution of keywords to be searched is unknown. Compared to PEKSrand-BG, PEKSrand-SG can further mitigate statistical guessing attacks at the cost of the storage overhead on the delegate, e.g., the mail server in the intelligent email routing application. According to our analysis and experimental results, both schemes introduce reasonabel additional communication and computation overheads and can be smoothly deployed in existing systems.

The remainder parts of this paper are organized as follows. In Section II, we briefly review PEKS. In Section III, we present the basic ideas of our solutions. The PEKSrand-BG and PEKSrand-SG schemes are presented in Section IV and Section V, respectively. Afterwards, we present the analysis on security and privacy of PEKSrand and the measures to the on-line guessing attacks in Section VI and Section VII respectively. The efficiency analysis and simulation results are presented in Section VIII, followed by the related work in Section IX. We draw the conclusion in Section X.

II. A Brief Review of PEKS

As shown in Figure 1, there are three types of entities in the PEKS scheme [5]: *receiver* R, *sender*, and *server* G. PEKS consists of the following four procedures:

1) $KeyGen(s)$: Takes as input a security parameter s, the receiver R generates a PEKS public/private-key pair, i.e., $\{A_{pub},\ A_{priv}\}$, as well as other public parameters;
2) $Trapdoor(A_{priv},\ x)$: given the private key A_{priv} and a keyword x, the receiver R produces a corresponding trapdoor T_x;
3) $PEKS(A_{pub},\ x)$: given the receiver's public-key (i.e., A_{pub}) and a keyword x, a *sender* generates the PEKS ciphertext of a message (e.g., an email) to be sent to R, which is denoted as S;
4) $Test(S, T_x)$: given the received PEKS ciphertext $S = PEKS(A_{pub},\ x')$ and a trapdoor $T_x = Trapdoor(A_{priv},\ x)$, the *server* G outputs 'yes' if $x = x'$ and 'no' otherwise.

An instantiated construction of PEKS is based on a bilinear map of elliptic curves. It uses two cyclic groups G_1, G_2 of the same prime order p and a symmetric bilinear map

$e: G_1 \times G_1 \rightarrow G_2$ between them. If we use a multiplicative notation to describe the operation in G_1 and G_2, e has following properties:

- Bilinear: $e(g^x, g^y) = e(g,g)^{xy}$ for all integers $x, y \in [1,p]$, $g \in G_1$;
- Non-degenerate: $e(g,g) \neq 1$ and if g is a generator of G_1 then $e(g,g)$ is a generator of G_2;
- Computable: There is a polynomial time algorithm to compute $e(g,h) \in G_2$ for all $g, h \in G_1$.

The security of PEKS is based on the assumption of Elliptic Curve DLP, which is believed to be intractable for certain carefully chosen groups including the group that is formed by the points on an elliptic curve defined over a finite field. More specifically, given two points on an elliptic curve, g and g^x, where x is a scalar, it is computationally infeasible to obtain x, if x is sufficiently large.

III. Basic Ideas of Our Solutions

We observe that, PEKS's incompetence in ensuring predicate privacy is due to two facts. On the one hand, the keywords used to create trapdoors in the targeted applications (e.g., intelligent email routing) are meaningful dictionary words. On the other hand, there exists a deterministic one-to-one mapping between a keyword and a trapdoor.

Based on this observation, our first idea is to randomize the original keywords so that the transformed keywords used to generate the trapdoors are not meaningful dictionary words any more. A naïve solution is that, user R (i.e., the receiver) and all other users (i.e., senders) share a secret N, which is concatenated with original keywords (e.g., the key refreshing solution proposed by Baek, Safiavi-Naini, and Susilo [3]) or is used as the key for hashing original keywords. However, such privacy protection is frail, since the protection of the shared secret is difficult. If any sender is compromised, which is very likely given that the size of the set of senders is usually large, this protection relies entirely on the security of the semi-trusted delegate. Moreover, it is also not suitable for scenarios where the membership of the set of senders might be dynamic, which results in additional costs of key/secret management. To address this issue, we limit the entities that hold the secret used for randomization to only one or a small set of proxy servers, which are well protected and thus are more secure than normal senders. This method also greatly reduces the cost of key/secret management. The PEKSrand-BG Scheme is built upon the first idea.

Another idea is to map a keyword to multiple trapdoors instead of one. It can weaken the effectiveness of statistical guessing attacks at the cost of the increasing overhead of storing trapdoors at the semi-trusted delegate. The PEKSrand-SG Scheme is developed through a combination of both ideas.

In our design, besides the three types of entities in the original PEKS system, we add a new type of entities called *proxy server*. In the remained parts of this paper, to avoid confusions, we denote searching server and proxy server as *gateway* and *proxy*, respectively.

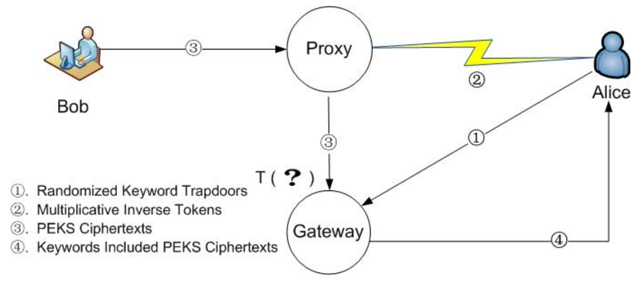
Bob
Proxy
Alice
Gateway
③
②
①
④
T (?)
①. Randomized Keyword Trapdoors
②. Multiplicative Inverse Tokens
③. PEKS Ciphertexts
④. Keywords Included PEKS Ciphertexts

Fig. 2. The Framework of the PEKSrand-BG Scheme

We assume that, both the proxy and the gateway are semi-trusted. In other words, they do not launch active attacks (e.g., probe-response attacks [13], [4], [16]) or collude with any malicious user, unless being compromised. We also assume that there exist certain security mechanisms that can detect the compromise that occurs on any proxy or the gateway (e.g., through monitoring behavior inconsistent with the protocol) and recover it within a short period. Hence, we assume that, although the adversary is capable of compromising the gateway or a proxy, she cannot control both all proxies and the gateway at the same time. We argue that this assumption is reasonable in practice, in particular in the PEKSrand-SG scheme, where a set of proxies instead of one are used.

IV. The PEKSrand-BG Scheme

The framework of the PEKSrand-BG scheme is illustrated in Figure 2. We denote the receiver and the sender as Alice and Bob, respectively. Now, Bob wants to send a message (e.g., an email) to Alice, who relies on the gateway to route the incoming messages based on the keywords contained in the messages.

In PEKSrand-BG, to be resistant to brute-force guessing attacks, Alice transforms the original meaningful keywords using a secret during the trapdoor generation. To guarantee that the searching function is still workable with randomized keywords without any interaction between Bob and Alice, we employ a proxy which sits between senders and the gateway. The proxy's major responsibility is to pre-process the PEKS ciphertexts received from the senders before forwarding them. We specify two hash functions $H_1: \{0,\ 1\}^* \rightarrow G_1$ and $H_2: G_2 \rightarrow \{0,\ 1\}^{\log p}$. The detailed procedures are as follows.

• $KeyGen(s)$: Alice picks a random number $\alpha \in Z_p^*$ and a generator g of G_1, and then outputs a public/private key pair $A_{pub} = [g,\ h = g^{\alpha}]$ and $A_{priv} = \alpha$. Afterward, Alice chooses a secret number $k \in \mathbb{Z}_p$ and calculates it's multiplicative inverse as $k^{-1} \in \mathbb{Z}_p$ which satisfies $k * k^{-1}(mod\ p) = 1$. At the end of this step, Alice sends k^{-1} to the proxy through a secure channel between Alice and the proxy;

• $Trapdoor(A_{priv}, x, k)$: Given the private key $A_{priv} = \alpha$, the secret k and a keyword x, Alice produces the trapdoor $T_x = H_1(x)^{\alpha * k}$ and delivers it to the gateway through another secure channel between Alice and the gateway;

• $PEKS(A_{pub}, x)$: For a keyword x, Bob first picks a random number $r \in Z_p^*$, and computes $t = e(H_1(x), h^r) \in G_2$, then outputs the PEKS ciphertext $S = [g^r, H_2(t)]$. Then, the PEKS ciphertext S is sent to the proxy;

• $PEKSrand(S, k^{-1})$: For each PEKS ciphertext S received, the proxy updates it with the multiplicative inverse number k^{-1}. More specifically, the transformed PEKS ciphertext (i.e., the PEKSrand ciphertext) is calculated as $S' = [g^{r*k^{-1}}, H_2(t)]$. Afterwards, the proxy forwards S' to the gateway.

• $Test(S', T_x)$: Let each PEKSrand ciphertext $S' = [A, B]$. The gateway tests if $H_2(e(T_x, A)) = B$. If so, then it is a match otherwise it is not match.

V. The PEKSrand-SG Scheme

Although PEKSrand-BG is efficient and can defend brute-force guessing attacks, we still have a few concerns about the security of this scheme. In PEKSrand-BG, we raise the threshold of breaking the system through compromising online server(s) from a single gateway in PEKS to two servers (i.e., a gateway and a proxy). However, in security-critical scenarios, we may want to further raise the bar. The other concern is that, the PEKSrand-BG scheme is still vulnerable to statistical guessing attacks. It is due to the fact that, the PEKSrand-BG scheme breaks only the deterministic and direct mapping between a meaningful keyword and the corresponding trapdoor through randomizing the original keyword but not the indirect one-to-one mapping between the original keyword and the new trapdoor. Hence, the frequency of the appearance of a specific keyword is the same as that of the corresponding trapdoor or predicate. Consequently, in the scenarios where the adversary has extra knowledge on the statistical distribution of keywords, the PEKSrand-BG scheme fails to protect predicate privacy.

For the first concern, a naïve solution of maintaining a few proxies holding the same secret k does not work. Even worse, it actually increases the risk of server compromises. Therefore, we think about increasing both the number of proxies and the number of secrets stored among the set of proxies. As to the second concern, our solution is to transform the one-to-one mapping, either direct or indirect, between an original keyword and a corresponding trapdoor in PEKS into a one-to-many mapping. To address these two concerns, in PEKSrand-SG we employ a combination of two methods: *Proxy Farm* and *Random Walk*.

A proxy farm consists of N proxies, each of which stores a distinct multiplicative inverse. In a simple application of this proxy farm method, upon receiving a PEKS ciphertext, the proxy performs the same type of ciphertext transformation as in PEKSrand-BG, with its own multiplicative inverse. Afterwards, the proxy forwards the resulting PEKSrand ciphertext to the gateway. In such a scheme, the PEKSrand ciphertexts corresponding to the same keyword are verified by distinct trapdoors at the gateway, if they are generated by different proxies. In other words, the original one-to-one mapping has been converted into a one-to-F mapping, where F is an important parameter related to privacy protection. As a result, the gateway has to store all the F trapdoors corresponding to the same keyword. Hence, the storage overhead at the gateway

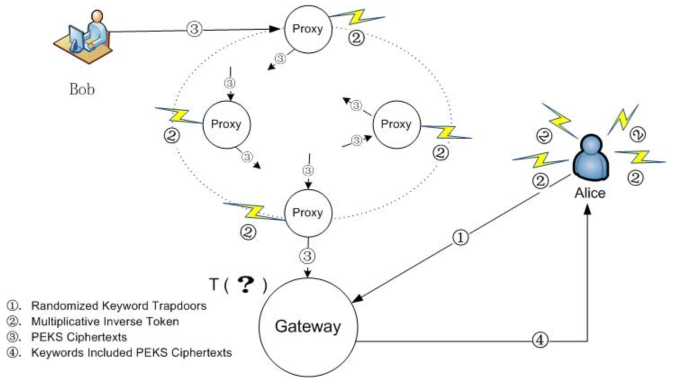
Bob
Proxy
Proxy
Proxy
Proxy
Alice
Gateway
T (?)
①. Randomized Keyword Trapdoors
②. Multiplicative Inverse Token
③. PEKS Ciphertexts
④. Keywords Included PEKS Ciphertexts

Fig. 3. The Framework of the PEKSrand-SG Scheme

is increased by a factor of F. In addition, we need F proxies in the proxy farm. Although the storage overhead at the gateway is reasonable in practice[1], it is costly to maintain a proxy farm with a large size, considering the level of security protection and trust level required. To mitigate this overhead, we integrate the idea of random walk into the proxy farm method. Now, a ciphertext will be transformed multiple times with distinct inverses instead of only once before it is finally forwarded to the gateway. Let U and u denote the number of proxies in the proxy farm and the number of times that a ciphertext is transformed with distinct inverses. In such a new method, with a proxy form with size U, we can achieve the same level of privacy protection as that is provided by a proxy form with size C_U^u in the simple application of this proxy farm method. The framework of the PEKSrand-SG scheme, which incorporates the ideas of proxy forma and random walk, is shown in Figure 3.

A. Procedures of the PEKSrand-SG Scheme

The PEKSrand-SG scheme consists of the four phases: setup, encrypt, random-walk, and keyword-searching.

1) Setup: To initialize the whole system, the following system-wide parameters are defined: U is the number of proxies that form the proxy farm, while u is the number of distinct proxies involved in a random walk; the security parameter s determines the size, p, of the groups G_1 and G_2, and e is a symmetric bilinear pairing between two groups and defined as $e : G_1 \times G_1 \rightarrow G_2$. Similar to PEKSrand-BG, to generate a system wide public key pair, Alice picks a random $\alpha \in Z_p^*$ and a generator g of G_1, and outputs $A_{pub} = [g, h = g^{\alpha}]$ and $A_{priv} = \alpha$.

To initialize the PEKSrand function in a proxy farm consisting of U proxies, Alice chooses U secret numbers ($k_i,\ for\ i = 1,\ 2, \ldots,\ U$) and calculates the corresponding multiplicative inverses k_i^{-1} that satisfies $k_i * k_i^{-1} (mod\ p) = 1$. Then, Alice sends each proxy a distinct $< i,\ k_i^{-1} >$ pair through a secure channel.

For each keyword x specified by Alice, C_U^u trapdoors corresponding to x, denoted as $T_x^j (j \in \{1, 2, \ldots, C_U^u\})$, are

[1]Please refer to Section VIII-A for more details.

generated. The trapdoors T_x^j's are calculated as follows:

$$T_x^j = H_1(x)^{\alpha * \prod_{i \in V_j} k_i}, \quad for\ j \in \{1, 2, \ldots, C_U^u\} \tag{1}$$

where V_j is a subset of $\{1,\ 2, \ldots,\ U\}$ with u elements. Let I_j denote a string that concatenates all elements in V_j with a predefined delimiter, such as ":". For example, given that $V_j = \{2, 4, 7\}$, I_j is denoted as "2:4:7". Finally, Alice distributes all C_U^u pairs of $< I_j,\ T_x^j >$ to the gateway through a secure channel.

2) Encrypt: In the encrypt phase of PEKSrand-SG, Bob encrypts the keyword x in the same way as in PEKSrand-BG and outputs the PEKS ciphertext $S = [g^r, H_2(t)]$. Afterwards, S is forwarded to a randomly chosen proxy in the proxy farm.

3) Random-Walk: Without loss of generality, we assume that proxy P_1 is the first proxy receiving the PEKS ciphertext S and P_1 holds the inverse k_1^{-1}. P_1 transforms the ciphertext with k_1^{-1} and outputs $S_1 = [g^{r*k_1^{-1}}, H_2(t)]$. Then, proxy P_1 generates a $<\ E_1,\ S_1\ >$ pair, where E_1 is the index of the multiplicative inverse held by proxy P_1 in the format of a string (i.e., "1" in this case), and forwards the pair to a randomly chosen proxy in the farm other than itself.

Without loss of generality, we assume that the path of the random walk within the proxy farm is $P_1 \rightarrow P_2 \rightarrow \ldots \rightarrow P_u$, and proxy P_i holds a multiplicative inverse k_i^{-1} for $i = 1,\ 2, \ldots,\ u$. For the following random walk process, we denote the PEKSrand ciphertext pair that a proxy P_i receives from another proxy as $<\ E_x,\ S_x = [g^{r*k_1^{-1}*k_2^{-1}\cdots*k_x^{-1}}, H_2(t)] >$ where x represents the number of proxies that have performed a transformation on the ciphertext so far during the random walk. Proxy P_i first checks whether the index of its multiplicative inverse is indicated in E_x. If it is true, it means that proxy P_i has previously performed a transformation on this ciphertext. In such a case, P_i simply forwards the received pair to a randomly chosen proxy again without any modification. Otherwise, proxy P_i will update the pair as $< E_{x+1},\ S_{x+1} = [g^{r*k_1^{-1}*k_2^{-1}\cdots*k_x^{-1}*k_{(x+1)}^{-1}}, H_2(t)] >$, where E_{x+1} is the concatenation of E_x and the index of the multiplicative inverse of P_i, separated by the predefined delimiter. Afterwards, proxy P_i checks the number of the indexes of inverses that appear in E_{x+1}. If it is less than u, P_i forwards the pair to a randomly chosen proxy in the farm other than itself. If it is equal to u, the random walk process is complete, and proxy P_i will forward the $<\ E_u,\ S_u\ >$ pair to the gateway.

4) Keyword-Searching: The whole trapdoor set that Alice assigns to the gateway can be divided into C_U^u subsets, each of which contains d trapdoors for d keywords that Alice chooses and is corresponding to a unique combination of u proxies. Each subset can be labeled with the corresponding I_j for $j \in \{1,\ 2, \ldots,\ u\}$. Upon receiving a $<\ E_u,\ S_u\ >$ pair from the last hop of the proxy farm, instead of searching the whole trapdoor set, therefore, we may first identify the subset of trapdoors corresponding to the combination of proxies that have performed the transformation operation on the ciphertext. It can be done by simply comparing X_u with the I_j's of subsets that the gateway receives from Alice.

Once the subset of trapdoors is determined, the gateway performs the keyword searching step in the same way as in the PEKSrand-BG scheme. Let $S_u = [A, B]$ denote the received PEKSrand ciphertext. More detailedly, the gateway executes $Test(S_u, T_x^j)$ to verify whether $H_2(e(T_x^j, A)) = B$ is satisfied. If so, it means that the original plaintext contains the keyword corresponding to the trapdoor used in the verification, i.e., x. The correctness of the verification is shown as follows.

$$\begin{aligned}
H_2(e(T_x^j, A)) &= H_2(e(H_1(x)^{\alpha*k_1*k_2*\cdots*k_n}, g^{r*k_1^{-1}*k_2^{-1}*\cdots*k_n^{-1}})) \\
&= H_2(e(H_1(x), g)^{\alpha*k_1*k_2*\cdots*k_n*r*k_1^{-1}*k_2^{-1}*\cdots*k_n^{-1}}) \\
&= H_2(e(H_1(x), g)^{\alpha*r*k_1*k_1^{-1}*k_2*k_2^{-1}*\cdots*k_n*k_n^{-1}}) \\
&= H_2(e(H_1(x), g)^{\alpha*r}) \\
&= B
\end{aligned}$$

VI. Security and Privacy Analysis of the PEKSrand Schemes

In this section, we analyze the level of security and privacy achieved in the PEKSrand-BG and PEKSrand-SG schemes.

A. Security Analysis

The security of both PEKSrand schemes relies on the difficulty of the Elliptic Curve DLP: suppose g^x and $g^{x*k^{-1}}$ (resp. g^{x*k}) are two points on an elliptic curve where both k^{-1} (respectively, k) and x are scalars. Given g^x and $g^{x*k^{-1}}$ (resp. g^{x*k}), it is computationally infeasible to obtain k^{-1} (respectively, k) , if k^{-1} (resp. k) is sufficiently large.

In the PEKSrand-BG scheme, due to the usage of randomized keywords, in order to break the system, e.g., compromising data confidentiality, the adversary has to compromise both the gateway and the proxy. In the PEKSrand-SG scheme, the protection is further enhanced in the sense that the adversary has to compromise both the gateway and at least u proxies.

B. Privacy Analysis

As an extension of PEKS, both variants of the PEKSrand scheme inherit PEKS's capability of ensuring plaintext privacy. Hence, in this paper we limit privacy analysis to the protection of predicate privacy, more specifically, privacy protection against brute-force guessing attacks and statistical guessing attacks. In addition, since these two types of attacks require the knowledge of trapdoors, which is only held by the semi-trusted gateway and the receiver, in the following analysis we focus on privacy protection against the gateway.

1) Protection against Brute-force Guessing Attacks: The root cause of brute-force guessing attacks against the original PEKS scheme is that, a predicate represents a deterministic and direct mapping between the original keyword and a trapdoor. In both variants of the PEKSrand scheme, such a mapping is changed. More specifically, the mapping represented by a predicate is neither deterministic (i.e., the original keyword is randomized before the generation of the trapdoor), nor direct (i.e., the mapping between the original keyword and the trapdoor is indirect, although there exists a direct mapping between the randomized keyword and the trapdoor). As a result, they are robust against brute-force guessing attacks.

2) Protection against Statistical Guessing Attacks: Unlike brute-force guessing attacks, in statistical guessing attacks the adversary has extra knowledge of the statistical distribution of keywords. We observe that, in spite of the randomization of keywords before trapdoor generation, in PEKSrand-BG there exists an indirect mapping between the original keyword and the trapdoor that is generated from a randomized instance of the original keyword. And such a mapping can be revealed through first recording the history of trapdoor mapping during the keyword searching procedure and then comparing the frequency of a specific keyword, which is obtained from the extra knowledge of the statistical distribution of keywords, with the frequency that each trapdoor has been successfully matched. Hence, the PEKSrand-BG scheme is vulnerable to statistical guessing attacks.

Since it is not feasible to limit the keyword usage at the sender side, we consider to mitigate the observed matched frequency of trapdoors during keyword searching phase at the gateway side. In the PEKSrand-SG variant, each keyword is mapped to multiple trapdoors instead of one in PEKSrand-BG. Theoretically, the PEKSrand-SG scheme is also vulnerable to statistical guessing attacks, since the expansion from one-to-one mapping to one-to-many mapping is applied to all keywords. Therefore, for two keywords x_1 and x_2, if the frequency of x_1 is higher than that of x_2, in PEKSrand-SG the frequency of any trapdoor that is mapped to x_1 is still higher than the frequency of any trapdoor that is mapped to x_2. For example, suppose that statistically the frequencies of x_1 and x_2 are 20% and 10%, respectively. Assume that, a proxy farm consisting of five proxies is deployed and each ciphertext has been transformed two times before being forwarded to the gateway. Hence, each keyword is mapped to $C_5^2 = 10$ trapdoors, and statistically the frequency of any trapdoor derived from x_1 (i.e., 2%) is higher than that of any trapdoor derived from x_2 (i.e., 1%). However, intuitively, given the same number of total successful trapdoor matching[2], to distinguish two events with the statistical probability of 2% and 1%, respectively, is more difficult than to distinguish two events with the statistical probability of 20% and 10%, respectively. In the following, we seek the theoretical basis that supports such an intuition and estimate the effectiveness of this method.

We begin with the entropy analysis [9]. We first define the *mapping factor* F as the number of trapdoors that are mapped to a single keyword. Hence, F is equal to 1 and C_U^u in PEKSrand-BG and PEKSrand-SG, respectively, where U and u denote the number of proxies in the proxy farm and the number of times that a ciphertext is transformed with distinct inverses. Let $X = \{x_1,\ x_2, \ldots,\ x_k\}$ denote the set of all keywords that the receiver chooses. Let $T_{BG} = \{T_1,\ T_2, \ldots,\ T_k\}$ denote the set of trapdoors corresponding to X in the PEKSrand-BG scheme. Let p_i denote the probability that keyword x_i or the corresponding trapdoor T_i is used. Hence, the entropy of keywords X or trapdoors T_{BG} in the PEKSrand-BG scheme, denoted as E_{BG}, can be calculated according to Equation (2).

$$E_{BG} = -\sum_{i=1}^{k} p_i \log p_i \tag{2}$$

Let $T_{SG} = \{T_1^1, \ldots,\ T_1^F,\ T_2^1, \ldots,\ T_2^F, \ldots,\ T_k^1, \ldots,\ T_k^F\}$ denote the set of trapdoors corresponding to X in the PEKSrand-SG scheme. p_i^j denote the probability that trapdoor T_i^j is used. In the PEKSrand-SG scheme, each keyword is mapped to F trapdoors evenly. Thus, we have $p_i^j = \frac{p_i}{F}$. Consequently, the entropy of keywords X or trapdoors T_{SG} in the PEKSrand-SG scheme, denoted as E_{SG}, can be calculated as follows.

$$\begin{aligned} E_{SG} &= -\sum_{i=1}^{k}\sum_{j=1}^{F} p_i^j \log p_i^j = -\sum_{i=1}^{k}\sum_{j=1}^{F} \frac{p_i}{F} \log \frac{p_i}{F} \\ &= -\sum_{i=1}^{k} p_i \log \frac{p_i}{F} = -\sum_{i=1}^{k} p_i (\log p_i - \log F) \\ &= -\sum_{i=1}^{k} p_i \log p_i + \sum_{i=1}^{k} p_i \log F \\ &= E_{BG} + \sum_{i=1}^{k} p_i \log F = E_{BG} + \log F \end{aligned} \tag{3}$$

According to Equation (3), compared to PEKSrand-BG, the entropy of keywords X is improved by a value of $\log F$ in PEKSrand-SG. In addition, Equation (3) also shows that, by increasing the mapping factor F, we can achieve better privacy protection on keywords. In practice, compared to entropy, probability is a more intuitive representation of the privacy criteria. Hence, in the following we present the probability analysis so as to illustrate the trade-off between privacy and efficiency in a more clear and intuitive manner.

To perform further analysis, we introduce a new concept called *n-F undistinguishable*. Let n denote the total number of successful trapdoor matching. Given a pair of trapdoors T_A and T_B corresponding to keywords A and B, respectively, without loss of generality, we assume that the frequency of A is higher than that of B according to the statistical distribution of keywords. If the actual number of times that trapdoor T_B is matched is no less than that of trapdoor T_A, we say that "the trapdoor T_A is n-F undistinguishable from the trapdoor T_B". Hence, our design goal is to maximize the probability of n-F undistinguishable, denoted as p_{n-F}. Further, let p_{n-F}^{BG} and p_{n-F}^{SG} denote the probability of n-F undistinguishable in PEKSrand-BG and PEKSrand-SG, respectively.

If we view each trapdoor matching as an experiment with only two possible results (i.e., "T_A is matched" and "Otherwise"), the probability of trapdoor T_A is matched k_A times in a sequence of n independent matching experiments can be calculated using Equation (4) according to the binomial distribution.

[2] In the theoretical analysis, we ignore unsuccessful trapdoor matching due to two reasons. The adversary's knowledge of the statistical distribution of keywords is defined in terms of all matched keywords. Moreover, we argue that taking unsuccessful trapdoor matching into consideration actually introduces noise to the statistical distribution and thus favor our goal of privacy protection.

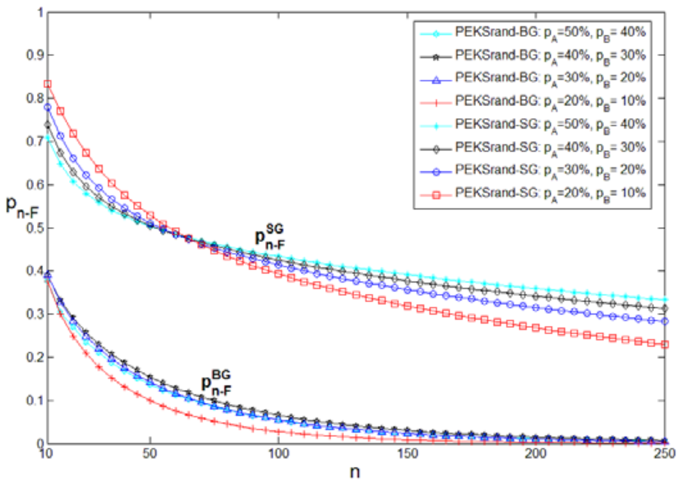

Fig. 4. Probability Comparison of PEKSrand-BG and PEKSrand-SG (10–250)

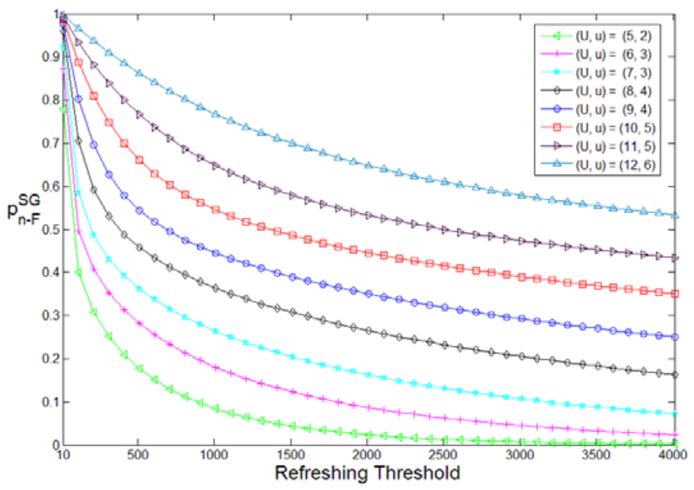

Fig. 5. Tradeoff between Statistics Privacy and Storage Efficiency

$$p_{A-k} = f(n, k_A, p_A) = C_n^{k_A} p_A^{k_A}(1-p_A)^{n-k_A} \tag{4}$$

where p_A is the statistical probability that trapdoor T_A is matched in the n independent experiments.

Similarly, if we view each trapdoor matching as an experiment with only two possible results (i.e., "T_B is matched" and "Otherwise"), among the remained $n-k_A$ independent matching experiments (namely, excluding k_A independent experiments matching T_A), the probability of trapdoor T_B is matched k_B times, given that trapdoor T_A is matched k_A times, can be calculated using Equation (5) according to the binomial distribution.

$$\begin{aligned} p_{B-k} &= f(n-k_A, k_B, \frac{n}{n-k_A}\cdot p_B) \\ &= C_{n-k_A}^{k_B}(\frac{n}{n-k_A}\cdot p_B)^{k_B}(1-\frac{n}{n-k_A}\cdot p_B)^{n-k_A-k_B} \end{aligned} \tag{5}$$

where p_B is the statistical probability that trapdoor T_B is matched in terms of all n experiments including those experiments matching T_A. Hence, p^{BG}_{n-F} can be calculated as shown in Equation (6).

$$\begin{aligned} p^{BG}_{n-F} &= \sum_{k_A=0}^{n}[f(n,k_A,p_A) * \sum_{k_B=k_A}^{n-k_A} f(n-k_A,k_B,\frac{n}{n-k_A}\cdot p_B)] \\ &= \sum_{k_A=0}^{n}[C_n^{k_A}p_A^{k_A}(1-p_A)^{n-k_A} * \\ &\quad \sum_{k_B=k_A}^{n-k_A} C_{n-k_A}^{k_B}(\frac{n\cdot p_B}{n-k_A})^{k_B}(1-\frac{n\cdot p_B}{n-k_A})^{n-k_A-k_B}] \end{aligned} \tag{6}$$

In the PEKSrand-SG, the one-to-one mapping between an original keyword and a corresponding trapdoor is transformed into a one-to-F mapping. Accordingly, the statistical probabilities of any trapdoor T'_A mapped to keyword A and any trapdoor T'_B mapped to keyword B are changed into $p'_A = \frac{p_A}{F}$ and $p'_B = \frac{p_B}{F}$, respectively. Hence, p^{SG}_{n-F} can be calculated according to Equation (7).

$$\begin{aligned} &p^{SG}_{n-F} \\ &= \sum_{k_A=0}^{n}[f(n,k_A,p'_A) * \sum_{k_B=k_A}^{n-k_A} f(n-k_A,k_B,\frac{n}{n-k_A}\cdot p'_B)] \\ &= \sum_{k_A=0}^{n}[C_n^{k_A}(\frac{p_A}{F})^{k_A}(1-\frac{p_A}{F})^{n-k_A} * \\ &\quad \sum_{k_B=k_A}^{n-k_A} C_{n-k_A}^{k_B}[\frac{np_B}{(n-k_A)F}]^{k_B}[1-\frac{np_B}{(n-k_A)F}]^{n-k_A-k_B}] \end{aligned} \tag{7}$$

Let $p^{SG\prime}_{n-F}$ denote the probability of n-F undistinguishable in PEKSrand-SG, given any possible pair of trapdoors T_A and T_B corresponding to keywords A and B, respectively. According to Equation (8), we know that $p^{SG}_{n-F} = p^{SG\prime}_{n-F}$.

$$p^{SG\prime}_{n-F} = \sum_{i=1}^{F}\sum_{j=1}^{F} p_{ij}\cdot p^{(i,j)}_{n-F} = \sum_{i=1}^{F}\sum_{j=1}^{F}\frac{1}{F^2}\cdot p^{SG}_{n-F} = p^{SG}_{n-F} \tag{8}$$

where p_{ij} denote that the probability of a specific pair that consists of the ith trapdoor corresponding to keyword A and the jth trapdoor corresponding to keyword B is chosen and $p^{(i,j)}_{n-F}$ denote the probability of n-F undistinguishable when such a pair is chosen.

In Figure 4, we compare the probabilities of n-F undistinguishable in PEKSrand-BG and PEKSrand-SG under four settings of p_A and p_B. As shown in Figure 4, p^{SG}_{n-F} is much higher than p^{BG}_{n-F} in all settings. Nonetheless, careful readers may notice that in Figure 4 the probability of n-F undistinguishable may drop to a low level if n is big enough, i.e., by observing a large number of trapdoor matching records, even in the PEKSrand-SG scheme.

To address this issue, we need to perform periodical secret refreshments, i.e., executing the setup phase in PEKSrand-SG after executing a predetermined number of successful trapdoor matchings, which we call the refreshing threshold. Apparently, there exists a trade-off between privacy and efficiency. More

TABLE I
MAPPING FACTORS AND REFRESHING THRESHOLDS WHEN $p_{n-F}^{SG} \geq 0.5$

(U, u)	F	TD	TD/F
(5, 2)	10	53	5.30
(6, 3)	20	108	5.40
(7, 3)	35	189	5.40
(8, 4)	70	379	5.41
(9, 4)	126	683	5.42
(10, 5)	242	1367	5.65
(11, 5)	462	2507	5.43

specifically, Figure 5 shows the trade-off between the probability of n-F undistinguishable and the refreshing threshold when $p_A = 30\%$ and $p_B = 20\%$. Note that, given that the number of proxies is fixed, to maximize the mapping factor F, we choose $u = \lfloor \frac{U}{2} \rfloor$. According to Figure 5, given a specific requirement on p_{n-F}^{SG}, by slightly increasing the number of proxies in the proxy farm, the refreshing threshold can be improved significantly. For example, assume that we set the privacy requirement as $p_{n-F}^{SG} \geq 0.5$, Table I shows the maximum refreshing threshold satisfying the privacy requirement under different settings of proxy farm and random walking. The mapping factor and the refreshing threshold is denoted as F and TD, respectively, in Table I. When the setting of random walking is $(U,\ u) = (5,\ 2)$, the maximum refreshing threshold is only 53. By simply increasing the number of proxies to 10 and 11, the maximum refreshing threshold is increased to 1367 and 2507, respectively. We argue that, such refreshing thresholds are sufficient for many real applications, e.g., intelligent email routing. A more detailed analysis about the overhead of the proposed scheme is given in Section VIII.

VII. RESISTANT TO ON-LINE GUESSING ATTACKS

As an improvement, PEKSrand is robust against off-line guessing attacks. Unfortunately, it does not guard again on-line guessing attacks, in that theoretically the semi-trusted (or intruded) gateway can act as a sender, by sending the PEKS chipertext of a keyword of her choice. Then she can receive the corresponding PEKSrand ciphertext from proxy server(s), and use it to test against possessed trapdoors.

In reality, these attacks can be prevented easily, since the semi-trusted (or intruded) gateway is required to operate on-line. Therefore, those on-line activities can be constrained by deploying misbehavior monitors such as Process Monitor [14]. In addition, since the gateway is reluctant to append identity information to avoid trace back, it baffles her to distinguish probes from other ciphertexts, considering the huge volume of traffic passing the gateway 24/7. Moreover, an important server, such as the gateway in PEKSrand, is usually protected properly by various security mechanisms such as IDS. Hence, an on-line intruder is usually captured at the early stage of attacks, and the server can be recovered shortly if needed. We also suggest to perform secret and trapdoor refreshment after recovery, i.e., executing $KeyGen(s)$ and $Trapdoor(A_{priv}, x, k)$ in PEKSrand-BG, and setup phase in PEKSrand-SG once, respectively, in case trapdoors are leaked for further attacks.

VIII. EFFICIENCY ANALYSIS AND EMPIRICAL RESULTS

Our PEKSrand implementation leverages the Identity Based Encryption [6] algorithms implemented in the MIRACL library [12]. We adopt the well-known Tate Pairing, which is the heart of the ciphertext generation, transformation, and testing processes. In our extension of PEKS, a 512-bit prime p is used for effective 1024-bit security, and G_1 and G_2 are groups on the supersingular elliptic curve $y^2 = x^3 + x \bmod p$ with 160-bit group order $q = 2^{159} + 2^{17} + 1$, a prime which divides p.

We simulate the PEKSrand-BG and PEKSrand-SG schemes on a desktop with an Intel(R) Core (TM)2 2.13GHz CPU (64-bit processor) and 2GB RAM. The programs run on Windows XP Professional operation system with ADO database connection to a Microsoft SQL 2000 database server.

A. Computation Overhead

Compared to the original PEKS, in terms of computation, PEKSrand-BG introduces only one additional exponential calculation per ciphertext at the proxy. In PEKSrand-SG, similarly, each proxy involved in the random walking process performs only one additional exponential calculation per ciphertext.

Note that, the number of trapdoor matching that the gateway needs to perform in PEKSrand-SG is the same as that of the original PEKS. It is due to the fact that, the trapdoor matching process is limited to a subset that contains d elements, where d is the number of keywords that Alice chooses, in spite that the total number of trapdoors is increased by a factor of F. More specifically, the gateway needs to perform $\frac{d}{2}$ trapdoor matching operation on average. Hence, the only additional operation at the gateway is to identify the subset that should be subject to the following trapdoor matching operation.

On our testbed, an exponential multiplication and a trapdoor matching operation take only 5 milliseconds and 13 milliseconds, respectively. And the operation of identifying a subset is also very efficient. It takes only 15 milliseconds in the worst case (i.e., $U = 11$ and $F = 462$). Therefore, compared to the original PEKS scheme, the additional computation overhead introduced by the PEKSrand schemes is negligible. Moreover, given a reasonable size of keywords to be searched, the actual computation overhead is small in most real world applications.

Table II shows the simulation results about the performance of the original PEKS and two PEKSrand schemes. The sample set we used are 256 keywords extracted from the Enron Email Dataset [11]. For each round, we randomly chose 50 out of 256 keywords and encrypt them, and then record both the number of trapdoor matching operations performed, as well as the exact time used at the gateway. Let $\#_{Test}$ and T_{Time} denote the number of trapdoor matching operations performed to identify all 50 keywords chosen and the time used that complete the keyword matching process. The results shown in the Table II are the averages of 50 rounds.

TABLE II
SIMULATION RESULTS

Scheme	$\#_{Test}$	T_{Time} (Second)
PEKS	6510	88.61
PEKSrand-BG	6510	88.61
PEKSrand-SG	6510	88.63

B. Storage Overhead

PEKSrand-BG has the same storage overhead at the gateway, since the number of trapdoors assigned to the gateway in PEKSrand-BG is the same as that of PEKS. In contrast, in PEKSrand-SG, the storage overhead is increased by a factor of F, while the original one-to-one mapping in PEKS is converted into a one-to-F mapping.

In our implementation, to achieve effective 1024-bit security, the size of a trapdoor is 128 bytes. Thus, given that there are 256 keywords in our simulation, the total storage overheads of PEKSrand-BG and PEKSrand-SG at the gateway's side are 32768 bytes and 32768*F bytes, respectively. We argue that, the storage overhead of PEKSrand-SG is still acceptable in many real world applications, considering that nowadays it is common that the hard drives of a server have the capacity of 1TB or more [10]. For example, in the intelligent email routing application, gateway with 1TB storage can support more than 2^{13} users, given that $F = 462$ and the number of keywords that each user specifies is 2^8 on average.

C. Communication Overhead

If we view the proxy in PEKSrand-BG or the proxy farm in PEKSrand-SG as a transparent component between the sender and the gateway, there is actually no additional traffic generated in the PEKSrand schemes, since the same number and size of ciphertexts are transmitted, although the content of packets are changed. However, the PEKSrand schemes indeed introduce some delay due to the ciphertext transformation and random walking within the proxy farm. Fortunately, as shown in Section VIII-A, the ciphertext transformation operation is lightweight. In addition, analysis in Section VI-B2 (in particular Table I) shows that only a small u, e.g., 4 or 5, is sufficient to satisfy the privacy requirement of most real world applications.

In PEKSrand-SG, the communication loads between a receiver and the gateway is increased by $C_U^u - 1$ times, in comparison with PEKS, to deliver trapdoors in both the setup and the periodical secret refreshments. However, as we presented in Section VI-B2, the frequency of the refreshment is reasonable small for most real world applications.

IX. RELATED WORK

The development of PEKS boosts many useful applications such as secure searchable automated remote email storage [2]. However, recent research on attacks against PEKS [3], [7], [1], [15] may discourage the usages of PEKS in real world applications.

Baek, Safiavi-Naini, and Susilo first brought the attention to the unlimited capability of keyword matching once the delegate is assigned the trapdoors [3]. They proposed to refresh the keywords by attaching time period information to them before performing the PEKS encryption. Later, Abdalla et al. proposed the public-key encryption with temporary keyword search (PETKS) scheme that aims at the same issue [1]. Two constructions are given. One is to generate a different key pair of the receiver for each time period. The other is the same as the one proposed by Baek, Safiavi-Naini, and Susilo [3]. Both works [3], [1] require interactions between the receiver and a large number of potential senders, and thus is impractical for real-world applications, e.g., intelligent email routing.

Byun et al. found that PEKS is susceptible to the offline keyword guessing attack, which is equivalent to the brute-force guessing attack defined in this paper. They claimed that anyone (insider/outside) can launch such attacks. However, such a claim is based on the assumption that the attacker can capture the valid trapdoor. They did not propose any countermeasure against the attack. Afterwards, Shen et al. formalized this type of attacks and defined predicate privacy in the context of predicate encryption system. They also gave a predicate privacy-preserving construction in the symmetric key setting for the inner-product predicate [15].

X. CONCLUSION

In this paper, we identified a new type of attacks against the original PEKS scheme (i.e., statistical guessing attacks) and proposed the PEKSrand scheme that aims at protecting predicate privacy and statistics privacy, which is a new concept introduced by us. Both variants of the PEKSrand scheme can prevent brute-force guessing attacks. However, only the PEKSrand-SG scheme can be used to mitigate statistical guessing attacks at the cost of a higher storage overhead at the gateway or delegate. According to our analysis and experimental results, both schemes introduce reasonable additional communication and computation overheads and can be smoothly deployed in existing systems.

REFERENCES

[1] Michel Abdalla et al. Searchable encryption revisited: Consistency properties, relation to anonymous IBE, and extensions. *Journal of Cryptology*, 21(3):350–391, 2008.

[2] Adam J. Aviv, Michael E. Locasto, Shaya Potter, and Angelos D. Keromytis. SSARES: Secure searchable automated remote email storage. In *Computer Security Applications Conference (ACSAC)*, pages 129–139, 2007.

[3] Joonsang Baek, Reihaneh Safavi-Naini, and Willy Susilo. Public key encryption with keyword search revisited. Cryptology ePrint Archive, Report 2005/191.

[4] John Bethencourt, Jason Franklin, and Mary Vernon. Mapping internet sensors with probe response attacks. In *USENIX Security Symposium*, 2005.

[5] Dan Boneh, Giovanni D. Crescenzo, Rafail Ostrovsky, and Giuseppe Persiano. Public key encryption with keyword search. In *EUROCRYPT 2004*, volume 3027 of *Lecture Notes in Computer Science*, pages 506–522, 2004.

[6] Dan Boneh and Matthew Franklin. Identity-based encryption from the Weil pairing. *SIAM J. of Computing*, 32(3):586–615, 2003.

[7] Jin Wook Byun, Hyun Suk Rhee, Hyun-A Park, and Dong Hoon Lee. Off-line keyword guessing attacks on recent keyword search schemes over encrypted data. In *Secure Data Management, Third VLDB Workshop 2006*, volume 4165 of *Lecture Notes in Computer Science*, pages 75–83.

[8] Oded Goldreich. *Secure Multi-Party Computation*, October 2002. Working draft, Version 1.4.

[9] R. M. Gray. *Entropy and information theory*. New York, Springer Verlag, 1990.

[10] IBM. Ibm system storage product guide.

[11] Bryan Klimt and Yiming Yang. The Enron corpus: A new dataset for email classification research. In *European Conference on Machine Learning 2004*, volume 3201 of *Lecture Notes in Computer Science*, pages 217–226, 2004.

[12] Shamus Software Limited. Multiprecision integer and rational arithmetic C/C++ library (MIRACL).

[13] Patrick Lincoln, Phillip Porras, and Vitaly Shmatikov. Privacy-preserving sharing and correlation of security alerts. In *USENIX Security Symposium*, pages 239–254, 2004.

[14] Mark Russinovich and Bryce Cogswell. Process monitor.

[15] Emily Shen, Elaine Shi, and Brent Waters. Predicate privacy in encryption systems. In *6th Theory of Cryptography Conference 2009*, volume 5444 of *Lecture Notes in Computer Science*, pages 457–473, 2009.

[16] Yoichi Shinoda, Ko Ikai, and Motomu Itoh. Vulnerabilities of passive internet threat monitors. In *USENIX Security Symposium*, pages 209–224, 2005.