Private Web Search with Malicious Adversaries*

Yehuda Lindell and Erez Waisbard

Department of Computer Science
Bar-Ilan University, Israel
{lindell,waisbard}@cs.biu.ac.il

Abstract. Web search has become an integral part of our lives and we use it daily for business and pleasure. Unfortunately, however, we unwittingly reveal a huge amount of private information about ourselves when we search the web. A look at a user's search terms over a period of a few months paints a frighteningly clear and detailed picture about the user's life. In this paper, we build on previous work by Castellà-Roca et al. (Computer Communications 2009) and show how to achieve privacy in web searches efficiently and practically without resorting to full-blown anonymous routing. In contrast to previous work, our protocol is secure in the presence of malicious adversaries.

1 Introduction

It is well known that users' search terms to web search engines contain significant amounts of sensitive information and, as such, the aggregation and use of these terms constitutes a severe privacy breach. The only way that a user can protect him or herself from this breach today is to use an anonymous routing system like Tor [7]. However, this can sometimes be an "overkill" measure. This is especially the case since in order to achieve a high level of security, such systems cause a considerable slowdown.

Recently, an interesting model for solving this problem was suggested by [2]. Essentially, their proposal is for a group of users to first *shuffle* their search words amongst themselves. After the shuffle, each user has someone's search word (but doesn't know whose), and the parties then query the search engine with the word obtained. Finally, the parties all broadcast the result to all others. This model is especially attractive because it doesn't involve the overhead of installing a full-blown anonymous routing system, and can be provided as a simple web service.

In [2], the authors present a protocol for private web search in the above model that is secure in the presence of semi-honest adversaries. That is, users' privacy is maintained only if all parties follow the protocol specification exactly. We argue that this level of security is not sufficient,

* This research was generously supported by the European Research Council as part of the ERC project "LAST".

especially due to the fact that the protocol of [2] has the property that a *single* adversarial participant can easily learn the queries of all users, without any malicious behavior being detected. This means that an adversarial entity who is a participant in many searches can learn all of the users' queries without any threat of retribution.

Our results. In this paper we construct a protocol for private web search in the model of [2] that is secure in the presence of *malicious adversaries* that may arbitrarily deviate from the protocol specification in order to attack the system. Our main technical tool is a highly efficient cryptographic protocol for parties to mix their inputs [3] that guarantees privacy in the presence of malicious adversaries. Unlike the usual setting of mix-nets, here the parties themselves carry out the mix. The novelty of our approach is based on the observation that, unlike the setting of voting where mix-nets are usually applied, the guarantee of *correctness* is not necessary for private web search. That is, we allow a malicious participant to carry out a "denial of service" type attack, causing the search to fail. In return, we are able to omit the expensive zero-knowledge proofs of correctness in every stage of the mix.

We stress that simply removing the correctness proofs from a standard mix protocol yields a completely insecure protocol that provides no privacy. For example, we still have to deal with "replacement attacks" where the first party carrying out the mix replaces all of the encrypted search words with terms of its own, except for the *one* ciphertext belonging to the user under attack. In this case, the result of the mix completely reveals the search word of the targeted user (because all other search words belong to the attacker). Our solution to this problem (and others that arise; see Section 3) is based on the following novel idea: instead of inputting search words into the mix, each party inputs an encrypted version of its search word. Then, after all stages of the mix are concluded, each party checks that its encrypted value appears. If yes, it sends true to all parties, and if not it sends false. If all parties send true, they can then proceed to decrypt the search words because this ensures that no honest party's search word was replaced. However, this raises a new challenge regarding how to decrypt the encrypted search word. Namely, a naive solution to the problem fails. For example, if each party encrypted their search word using a one-time symmetric key, then sending this key for decryption reveals the identity of the party whose search word it is. We therefore use a "one-time" threshold encryption scheme based on ElGamal [8] and have the parties encrypt the search words with the combined key. The parties then send their key-part in the case that all parties sent

true (a similar idea to this appears in [2] but for a different purpose). We call this a *private shuffle* in order to distinguish it from a standard mix-net. We provide a formal definition of security for a private shuffle and have a rigorous proof of security under this definition.

As we have mentioned, the private shuffle is the main technical tool used for obtaining private web search. However, as is often the case, the cryptographic protocol at its core does not suffice for obtaining a secure *overall solution*. In Section 5 we therefore discuss how a private shuffle primitive can be used to obtain private web search, and in particular how to bypass non-cryptographic attacks that can be fatal. One major issue that arises is how to choose the group of participants, and in particular, how to prevent the case that the adversary controls all but one participant (in which case the adversary will clearly learn the input of the sole honest party). This issue was not addressed in previous solutions.

Related work. A number of different anonymity-preserving techniques can be used in principal for private web search. For example, private information retrieval [4, 11] provides the appropriate guarantees. However, it is far too inefficient. A more natural candidate is a to use a mix-net [3]. However, as we have mentioned, considerable expense goes into proving correctness in these protocols. In addition, doing this efficiently and securely turns out to be quite a challenge; see for example [10, 6]. For further comparisons of existing techniques to the model that we adopt here, we refer the reader to [2] and the reference within. We remark that our protocol is about twice as expensive as the protocol of [2], and thus the efficiency comparisons between their solution and other existing techniques can be extrapolated to our solution. (For some reason, however, they used ElGamal over $\mathbb{Z}_p^*$ with a large p instead of an Elliptic curve group that would be considerably more efficient.) Our solution has some similarities to that of [2]. However, their protocol suffers from a number of attacks in the case of malicious adversaries, as described below in Section 3.

2 Definitions

In this section we present our definition of security for a private shuffle primitive. The shuffle functionality is simply the n-ary probabilistic function $f(x_1, \ldots, x_n) = (y_1, \ldots, y_n)$, such that for every i, $y_i = x_{\pi(i)}$ where π is a random permutation over $[n]$. Intuitively, a shuffle is *private* if an adversary cannot link between the inputs of the protocol and the outputs of the protocol. Namely, the adversary should not be able to link y_j to an honest party P_i where $j = \pi(i)$. Denoting the number of corrupted

parties by t, we have that a random guess regarding a "link" is correct with probability $\frac{1}{n-t}$. Thus, we formalize security by requiring that an adversary controlling t parties can output (i, j) where P_i is honest and $j = \pi(i)$ with probability that is at most negligibly greater than $\frac{1}{n-t}$.

The security experiment. We assume that the parties communicate over an open network with unauthenticated channels. We model this network by having all communication go through an adversary that can listen to all the communication, delete messages and inject messages of its choice. This is formally modeled by providing the adversary with stateful oracles that model the honest parties, as in [1]. The experiment modeling the success of the adversary appears in Figure 1.

FIGURE 1 (The Security Experiment $\mathsf{ExptShuffle}^{t,n}_{\mathcal{A},\pi}(k)$)

1. Invoke the adversary $\mathcal{A}$ with input 1^k and with parameters t and n (k is the security parameter determining the key sizes).
2. Receive from $\mathcal{A}$ a set of t indices $I \subset [n]$ designating the corrupted parties (note that $|I| = t$), and a vector of $n - t$ *distinct* inputs $w_1, \ldots, w_{n-t}$ for the honest parties.
3. Choose a random permutation π over $\{1, \ldots, n - t\}$ and initialize the ith honest-party oracle with input $w_{\pi(i)}$.
4. Execute the shuffle protocol, where $\mathcal{A}$ interacts with the $n - t$ oracles (who each runs the shuffle protocol honestly based on messages received from $\mathcal{A}$).
5. When it concludes, the adversary outputs a pair (i, j) for any i, j of his choice.

We say that the adversary succeeds in the experiment, in which case the output of the experiment $\mathsf{ExptShuffle}^{t,n}_{\mathcal{A},\pi}(k)$ equals 1, if and only if $\pi(i) = j$.

Defining security. We are now ready to define security. First, we require non-triviality, meaning that if all parties are honest, then the protocol output is a permuted vector of the inputs. Next, we require that an adversary controlling t out of the n parties can succeed in the experiment ExptShuffle with probability that is only negligibly greater than $\frac{1}{n-t}$ (where negl is a negligible function if for every polynomial p and all large enough k's it holds that $\mathsf{negl}(k) < 1/p(k)$):

Definition 2 *A protocol π is a* private shuffle *if it is non-trivial, and if for every probabilistic polynomial-time algorithm $\mathcal{A}$, every integer $n \in \mathbb{N}$ and every $0 < t < n$, there exists a negligible function $\mathsf{negl}(\cdot)$ such that:*

$$\Pr\left[\mathsf{ExptShuffle}^{t,n}_{\mathcal{A},\pi}(k) = 1\right] \leq \frac{1}{n-t} + \mathsf{negl}(k)$$

3 Constructing a Private Shuffle

In order to motivate our construction, we begin by describing the protocol of [2] that is secure in the presence of semi-honest adversaries. We then

describe the difficulties that arise when moving to the malicious model. A basic tool that is used is called *ElGamal remasking*. Intuitively, a remasking operation is a procedure that takes a ciphertext and rerandomizes it so that the result cannot be linked to the original ciphertext. Recall that an ElGamal encryption of a message M with public-key (g, y) is computed by choosing a random $r \in \mathbb{Z}_q^*$ (where the group has order q) and computing $u = g^r$ and $v = y^r \cdot M$; the ciphertext is the pair $c = (u, v)$. The remasking operation is computed as follows:

$$\mathsf{remask}(u, v) = (u \cdot g^{r'}, y^{r'} \cdot v)$$

where $r' \in_R \mathbb{Z}_q^*$. Observe that when $(u, v) = (g^r, y^r \cdot M)$ it follows that $\mathsf{remask}(u, v) = (g^{r+r'}, y^{r+r'} \cdot M)$ and so it is a valid encryption of the same message under the same public key. The fact that $\mathsf{remask}(u, v)$ cannot be linked to (u, v) is due to the fact that r' is random and follows from the decisional Diffie-Hellman (DDH) assumption. An informal description of the protocol of [2] appears in Protocol 3.

Protocol 3 (The protocol of [2] for semi-honest adversaries (overview))

- Parties $P_1, \ldots, P_n$ generate a joint ElGamal public key $y = \prod_{i=1}^{n} g^{x_i}$, where x_i denotes the private key of each party.
- Every party P_j encrypts its search word w_j using the joint public key, obtaining $c_j^0 = (u_j^0, v_j^0)$, and sends it to everyone.
- For every $i = 1, \ldots, n$, party P_i does the following:
  - *Remasks* the ciphertexts $(c_1^{i-1}, \ldots, c_n^{i-1})$ it received from P_{i-1}.
  - Randomly *permutes* the remasked ciphertexts.
  - Sends the shuffled and remasked ciphertexts to P_{i+1}, except for party P_n who broadcasts the result to all the parties.
- Given the shuffled and remasked ciphertexts $(c_1^n, \ldots, c_n^n)$, each party P_i decrypts a single ciphertext $c_i^n = (u_i^n, v_i^n)$. This is carried out as follows:
  - Each party P_j sends each P_i the share $(u_i^n)^{x_j}$ for every $i, j \in \{1, \ldots, n\}$, where x_j is P_j's private key.
  - Given the shares from all parties, each P_i computes $w_i = \frac{v_i^n}{\prod_{j=1}^{n} (u_i^n)^{x_j}}$

Although Protocol 3 was defined for the semi-honest model, it is instructive to see what attacks can be carried out by a malicious party:

Stage-skipping attack: A malicious party P_n may remask and permute the initial vector of ciphertexts sent by the parties instead of the vector that it received from P_{n-1}. In this case, when the vector is decrypted P_n will know exactly which party sent which message. Observe that this behavior would not be detected because the remask operation looks identical when applied once or n times.

Input-replacement attack: A malicious party P_1 can learn the input w_j of an honest party P_j by replacing all the ciphertexts in the in-

put vector with individually remasked copies of the *initial* ciphertext (u_j^0, v_j^0). In this case, all of the parties receive w_j; in particular P_1 receives w_j and so knows the search term of P_j.

Targeted public-key attack: A malicious P_n may compute its share of the public key after given all of the g^{x_i} values of the other parties. Specifically, P_n sets its share of the public-key to be $h = g^{x_n}/(\prod_{i=1}^{n-1} g^{x_i})$ for a random x_n. Observe that any encryption under $y = \prod_{i=1}^{n} g^{x_i}$ is actually an encryption under g^{x_n} only because $h \cdot y = g^{x_n}$. Thus, P_n can decrypt the values of all parties and learn who sent what. Once again, this attack would go completely unnoticed.

Private shuffle for malicious adversaries. We now motivate our protocol for private shuffle that achieves security in the presence of malicious adversaries. First, in order to guarantee privacy, we need to ensure that at least one honest user remasks and permutes all of the ciphertext values. This involves ensuring that all parties take part in the shuffle and that the parties shuffle the actual input values (that is, we need to ensure that neither a stage-skipping nor input-replacement attack is carried out). The classic way of achieving this in the mix-net literature [3, 10, 6] is to have each party P_i prove (at each stage) that the values that it passed onto P_{i+1} are indeed a remasked and permuted version of what P_i received from P_{i-1}. However, this is a costly step that we want to avoid. We therefore provide an alternative solution that is based on a two-stage protocol with double encryption of each input. In the first stage the parties shuffle the inputs without verifying correctness, while gradually removing the outer encryption. Then, at the end of this stage there is a verification step in which all parties check that their input value is still in the shuffled array (under the inner encryption). If all parties acknowledge that their value is present then we are guaranteed that all parties participated in the shuffle and that no inputs were replaced. We can therefore safely proceed to the second stage of the protocol where the inner encryption is privately removed, revealing the shuffled inputs. In addition to the above, we prevent the aforementioned targeted public-key attack by having each party prove that it knows its associated secret key.

We note that in order to prevent a powerful man-in-the-middle adversary from playing the role of all parties except for one, we assume the existence of a PKI for digital signatures; see Section 5 for a discussion of how to achieve this in practice. In addition, we assume that all parties hold a unique *session identifier sid* (e.g., this could be a timestamp), and a generator g and order q of a group for ElGamal.

Protocol 4 (Private Shuffle with Malicious Adversaries)

Input: Each P_j has a search word w_j, and auxiliary input (g, q) as described.

Initialization Stage:

1. Each party P_j chooses random $\alpha_j, \beta_j \in Z_q^*$, sends $g^{\alpha_j}, g^{\beta_j}$ to all the other parties and proves knowledge of α_j, β_j using a zero-knowledge proof of knowledge. P_j signs the message it sends together with the identifier *sid* using its certified private signing key (from the PKI).
2. Each party verifies the signatures on the messages that it received and aborts unless all are correct.
3. Each party P_j encrypts its input w_j using the public g^{α_i} shares of all the other parties. That is, it chooses a random $\rho_j \in_R Z_q^*$ and computes an encryption $c_j = (g^{\rho_j}, g^{\rho_j \cdot \sum_{i=1}^n \alpha_i} \cdot w_j)$. (The value $g^{\sum_{i=1}^n \alpha_i}$ is computed by multiplying all of the g^{α_i} values received in the previous stage.)
4. Each party P_j re-encrypts its ciphertext c_j using the public g^{β_i} shares:
   (a) The party computes $\Delta_0 = \prod_{i=1}^n g^{\beta_i} = g^{\sum_{i=1}^n \beta_i}$
   (b) It chooses a random value $\rho_j' \in_R Z_q^*$
   (c) It encrypts c_j by computing $(u_j^0, v_j^0) = (g^{\rho_j'}, (\Delta_0)^{\rho_j'} \cdot c_j)$ and sends the result to all the other parties.

The output of this phase is the list of the encrypted c_j's of all the parties, denoted $\mu_0 = \langle (u_1^0, v_1^0), \ldots, (u_n^0, v_n^0) \rangle$.

Shuffle stage: For $j = 1, \ldots, n$, party P_j receives vector μ_{j-1} and computes a shuffled version μ_j as follows:

1. For every (u_i^{j-1}, v_i^{j-1}) in μ_{j-1}, party P_j carries out the following steps:
   (a) **Remask:** it chooses a random $r_j^i \in Z_q^*$ and computes
   $$(u_i', v_i') = (u_i^{j-1} \cdot g^{r_j^i}, v_i^{j-1} \cdot (\Delta_{j-1})^{r_j^i}) \quad \text{where} \quad \Delta_{j-1} = g^{\sum_{i=j}^n \beta_i}$$
   where the computation of Δ_{j-1} can be carried out using the g^{β_i} values sent in the initialization phase.
   (b) **Remove β_j:** it computes $(u_i^j, v_i^j) = (u_i', u_i'^{(-\beta_j)} \cdot v_i')$
2. P_j chooses a random permutation π_j over $\{1, \ldots, n\}$ and applies it to the list of values (u_i^j, v_i^j) computed above; denote the result by μ_j.
3. P_j sends μ_j to P_{j+1}.

The last party P_n sends μ_n to all parties.

Verification stage:

1. Every party P_j checks that its encryption c_j of w_j under public key $\prod_{i=1}^n g^{\alpha_i}$ is in the vector μ_n. If yes it sends $(sid, P_j, \mathsf{true})$, signed with its private signing key, to all the other users. Otherwise it sends (P_j, false).
2. If P_j sent false in the previous step, or did not receive a validly signed message $(sid, P_i, \mathsf{true})$ from all other parties P_i, then it aborts. Otherwise, it proceeds to the next step.

Reveal stage:

1. For every $(u_i, v_i) \stackrel{\text{def}}{=} (u_i^n, v_i^n)$ in μ_n, party P_j removes its α_j from the encryption by sending $s_i^j = u_i^{\alpha_j}$ to P_i (including sending $s_j^j = u_j^{\alpha_j}$ to itself).
2. After receiving all the shares s_j^i, every party P_j computes $w_j = \frac{v_j}{\prod_{i=1}^n s_j^i}$, thereby removing the second layer of encryption and recovering the cleartext word w_j (here j denotes the current index in μ_n and not the index of the party who had input w_j at the beginning of the protocol).

Remarks on the protocol:

1. For the sake of efficiency, the zero-knowledge proof in the initialization stage can be implemented by applying the Fiat-Shamir heuristic [9] to Schnorr's protocol for discrete log [12]. In order to achieve independence, we also include the *sid* and the party ID of the prover inside the hash for generating the "verifier query". It is also possible to use the methodology of [5] at the expense of $\log n$ rounds of communication.
2. Observe that each input w_j is encrypted twice under ElGamal. However, the result c_j of the first encryption is actually *two* group elements. Thus, if the same group is used for both layers of encryption, then we need to separately encrypt the two elements in c_j. For the sake of clarity, we present the protocol as if the second encryption under Δ_0 is a larger group in which encryption of both elements is achieved in a single operation.
3. In the first stage of the protocol every party participates in the shuffle. However, as we will see in the proof it suffices to ensure that *one* honest party participated. Thus, if we assume that at most t parties are malicious (for $t < n$), then we can run the shuffle stage for $j = 1$ to $t + 1$ only, reducing the number of rounds from n to $t + 1$.

Non-triviality. The non-triviality requirement of a private shuffle is that if all parties are honest then the output is a permutation of the input values $(w_1, \ldots, w_n)$. We prove that this property holds for our protocol by following a single message w_ℓ that goes through the protocol, and showing that all the layers of encryption that are added are properly removed. For clarity, we present this for the case that no permutations are applied (and thus the indices remain the same); this clearly makes no difference.

1. In the initialization phase the message w_ℓ is encrypted first with $g^{\sum_{i=1}^n \alpha_i}$ (using random ρ_ℓ) resulting in c_ℓ, and then c_ℓ is encrypted with $g^{\sum_{i=1}^n \beta_i}$ (using random value ρ'_ℓ) yielding the pair (u^0_ℓ, v^0_ℓ) where $u^0_\ell = g^{\rho'_\ell}$ and $v^0_\ell = (\Delta_0)^{\rho'_\ell} \cdot c_\ell = g^{\rho'_\ell \cdot \sum_{i=1}^n \beta_i} \cdot c_\ell$.
2. Assume that before the jth iteration begins, the pair $(u^{j-1}_\ell, v^{j-1}_\ell)$ is an encryption of c_ℓ under the ElGamal public key $\Delta_{j-1} = g^{\sum_{i=j}^n \beta_i}$. This clearly holds for $j = 1$ by the way (u^0_ℓ, v^0_ℓ) are generated. We show that this holds after the jth iteration concludes. By the above assumption, before the jth iteration begins, there exists a value $r \in \mathbb{Z}^*_q$ such that $u^{j-1}_\ell = g^r$ and $v^{j-1}_\ell = (\Delta_{j-1})^r \cdot c_\ell$. In the jth iteration of the shuffle stage, party P_j computes $u'_\ell = u^{j-1}_\ell \cdot g^{r^\ell_j} = g^{r+r^\ell_j}$ and $v'_\ell = v^{j-1}_\ell \cdot (\Delta_{j-1})^{r^\ell_j} = (\Delta_{j-1})^r \cdot c_\ell \cdot (\Delta_{j-1})^{r^\ell_j} = (\Delta_{j-1})^{r+r^\ell_j} \cdot c_j$. Thus (u'_ℓ, v'_ℓ) constitute an encryption of c_ℓ under public-key Δ_{j-1}.

Next P_j computes $u_\ell^j = u'_\ell = g^{r+r_j^\ell}$ and $v_\ell^j = {u'}_\ell^{-\beta_j} \cdot v'_\ell$. It follows that $v_\ell^j = g^{-\beta_j(r+r_j^\ell)} \cdot \Delta_{j-1}^{r+r_j^\ell} \cdot c_\ell = g^{-\beta_j(r+r_j^\ell)} \cdot g^{(r+r_j^\ell)\sum_{i=j}^n \beta_i} \cdot c_\ell = g^{(r+r_j^\ell)\sum_{i=j+1}^n \beta_i} \cdot c_\ell = \Delta_j^{r+r_j^\ell} \cdot c_\ell$. We therefore conclude that after the jth iteration, the result is an encryption of c_ℓ under Δ_j, as required.

3. From the above, we have that after all n iterations are concluded the value c_ℓ is obtained in the clear (observe that $\Delta_n = g^0 = 1$).
4. Next, if all the parties are honest, then they all send true in the verification stage, and all send P_ℓ the values $s_\ell^i = u_\ell^{\alpha_i}$ (for every i). Recall that $c_\ell = (u_\ell, v_\ell)$ where $u_\ell = g^{\rho_\ell}$ and $v_\ell = g^{\rho_\ell \cdot \sum_{i=1}^n \alpha_i} \cdot w_\ell$. Now, $\prod_{i=1}^n s_\ell^i = \prod_{i=1}^n u_\ell^{\alpha_i} = \prod_{i=1}^n g^{\rho_\ell \cdot \alpha_i}$. Thus, $v_\ell = \prod_{i=1}^n s_\ell^i \cdot w_\ell$, implying that $\frac{v_\ell}{\prod_{i=1}^n s_\ell^i} = w_\ell$, as required.

We have proven that the output of the protocol consists of all the original inputs. The shuffle function definition also requires that these be in a randomly permuted order. However, since each party applies a random permutation to the vector of ciphertexts, this immediately follows. We conclude that when all parties are honest, the protocol computes the shuffle functionality as defined.

4 Privacy of Shuffle Protocol

The security of the protocol is based on the decisional Diffie-Hellman (DDH) assumption. Informally, this states that an adversary can distinguish tuples of the type (g, g^a, g^b, g^{ab}) from tuples of the type (g, g^a, g^b, g^c), where a, b, c are random in $\mathbb{Z}_q^*$, with probability that is negligible in k (where k is the bit-length of q). We have the following theorem:

Theorem 5 *Assume that the decisional Diffie-Hellman (DDH) assumption holds in the group of order q generated by g. Then, for every probabilistic polynomial-time algorithm $\mathcal{A}$, every integer $n \in \mathbb{N}$ and every $0 < t < n$, there exists a negligible function $\mathsf{negl}(\cdot)$ such that:*

$$\Pr\left[\mathsf{ExptShuffle}_{\mathcal{A},\pi}^{t,n}(k) = 1\right] \leq \frac{1}{n-t} + \mathsf{negl}(k)$$

We now provide intuition as to why the above theorem holds. The most important point is that as long as at least *one* honest party carries out the shuffle (remask and permute) operation on the vector of inputs, the adversary can succeed in $\mathsf{ExptShuffle}$ with probability at most negligibly greater than $1/(n-t)$. This is due to the fact that by the DDH assumption, no polynomial-time adversary can link between an ElGamal encryption (u_i, v_i) and its remasked version (u'_i, v'_i), without knowing all

β_i values. Thus, after an honest party remasks and permutes the values, the trail from the party who initially sent the relevant encryption is lost. Of course, it is necessary to show that the reveal stage at the end does not de-anonymize the values; however, this is straightforward. In order to show that at least one honest party carried out the shuffle, we show that unless the vector μ_n is the result of *all* parties carrying out the shuffle in turn, the honest parties all abort (except with negligible probability). In order to see this, we first argue that if an adversary carries out an *input-replacement* or *stage-skipping* attack (as described above), then the honest parties all abort except with negligible probability.

1. *Input-replacement attack:* The honest parties all encrypt their inputs w_j under $g^{\prod_{i=1}^{n}\alpha_j}$ and then re-encrypt the result under $g^{\prod_{i=1}^{n}\beta_j}$. Thus the adversary does not know the value of the ciphertext c_j which is the encryption of w_j under $g^{\prod_{i=1}^{n}\alpha_j}$. Since this ciphertext value is of high entropy (even if w_j is not), it follows that if an honest party's input P_j is replaced at any stage of the computation, the correct c_j will not appear in the verification stage. In this case, P_j will send (P_j, false) and all honest parties will abort. Note that since the true confirmation messages are signed, an adversary that controls the communication channels cannot send a true message when the actual P_j sends false.
2. *Stage-skipping attack:* This attack refers to a malicious party P_j remasking and permuting a vector μ_i instead of μ_{j-1}, where $i < j-1$ and an honest party P_ℓ is between P_i and P_j (note that such an attack is *not* a replacement attack). In order to see why such an attack is detected, recall that the g^{β_j} component of the outer encryption is removed iteratively in each stage. Thus, if P_j takes μ_i it follows that the encryption under g^{β_ℓ} is not removed. In such a case, none of the correct ciphertexts (encrypted under $g^{\prod_{i=1}^{n}\alpha_j}$) will be obtained and all honest parties will send false and abort. (This explains why the β_j components are removed iteratively, and not all together at the end.)

The intuition is completed by observing that if an adversary does not carry out an input-replacement or stage-skipping attack, then it holds that all honest parties participated in the shuffle, as required. The full proof is omitted here due to lack of space in this extended abstract; the full proof will appear in the full version.

5 Private Web Search

In this section we show how to use a *private shuffle* in order to achieve private web search. As we will show below, a system for private web search

needs to take into account additional considerations that are not covered by the notion of a private shuffle (or even a fully secure mix-net). In this section we address these considerations, describe the assumptions that we make, and present a general scheme that models real-world threats and is thus implementable in practice.

5.1 Background

As in [2], the basic idea of the scheme is to allow many users who wish to submit a web query to team up in a group, shuffle their queries in a private manner and then have each of them perform one of the queries without knowing who it belongs to. Upon receiving back the query results, each party just sends them to all others in the group so that the original party who sent the query can learn the result. This methodology prevents the search engine from linking between a user and its search query. Furthermore, the users in the group do not know on whose behalf they send a query; all they know is that it belongs to someone within the group. An important question in such a system is how to group users together. One possibility is to do this in a *peer-to-peer* way, so that whenever a user has a query it can notify the peer network in order to find out who else has a query at this time. The parties with queries can then join in an ad-hoc way in order to privately shuffle them before sending them to the search engine. (Note that parties who are currently idle can help by sending dummy queries, if they like.) This is a feasible model, but has significant implementation difficulties. The alternative suggested by [2], and one that we follow for the remainder of this section, is to use a *central server* whom anyone interested in searching can approach. The server then notifies the parties wishing to currently search of each others' identities so that they can form a group in order to carry out a private shuffle. This model is easily implemented by simply having the server be a website offering a "private search" utility.

As we mentioned in the introduction, the problem with the scheme suggested by [2] was that it assumed that all parties are semi-honest. In our view this is highly unrealistic, especially since a single corrupt party can completely break the privacy of the scheme and learn every party's search query. We now show how to achieve private web search in the presence of malicious adversaries. In order to do this, we use the *private shuffle* protocol presented in Section 3 that maintains privacy in the presence of malicious adversaries. We stress that private shuffle within itself does not suffice for obtaining private web search in practice for the following reasons:

1. A malicious central server can choose the group so that it controls all but one user. As we explain below, this completely bypasses the security guarantees of the shuffle.
2. The result of the web search queries must be sent to all parties because we don't know which user sent which query. This means that users learn the search results for all the members in their group, which is much more information than necessary (although the search engine must learn all queries, this is not the case for users).

Below, we will present a system for web search that uses the *private shuffle* protocol, while addressing the above concerns.

5.2 A Private Web Search System

Our solution is comprised of four phases that together enable private web search:

- **Phase 0:** Installation and initialization
- **Phase 1:** Ad-hoc group setup
- **Phase 2:** Private shuffle of the search queries
- **Phase 3:** Query submission and private response retrieval

We remark that an ad-hoc group can be used for many searches, and ideally would be used for a session of a reasonable amount of time. This enables us to reduce the overhead due to running phase 1.

Phase 0 – installation and initialization: Our *private shuffle* protocol requires a PKI and communication with a central server. A natural realization of this would be as an Add-on to a web browser that would supply a functionality which is similar to the search window in the most common web browsers. This Add-on would contain the address of a central server (or a list of servers). Regarding the PKI, since most users do not have a certificate for digital signatures, we have to generate one. The most practical way to do this would be to use a one-time activation of the Add-on after installation, in which a key pair is generated and a digital certificate then downloaded from a CA. Recall that without a PKI, the efficient verification in our *private shuffle* protocol does not guarantee that it was the honest parties in the group that sent `true` in the verification of the shuffle stage. We stress that a different certificate can be installed on every machine using the Add-on.

Phase 1 – ad-hoc group setup: As mentioned above, users group together with the help of a server $\mathcal{S}$ that aggregates the identities of users that wish to currently engage in private web search. Conceptually speaking, in terms of role and trust, the server should be no more than a bulletin board for anonymous users who wish to create an ad-hoc group. In [2], the server was assumed to be a trusted entity who does not collude with any of the users nor with the web search engines. However, the role of grouping users together carries with it a lot of power that can easily be abused. Specifically, consider a server that has $t \geq n$ machines at its disposal (or even a single machine that can pretend to be t different users), where n is the size of the group. Then, the server can always group some single honest user with $n-1$ of the t server-owned users. If an honest user runs a private shuffle in this way, then its privacy is completely lost because the server knows the search queries of all the users except for the honest one. Thus, at the end of the protocol when all queries are revealed, the server knows the exact query made by the honest user. We stress that this holds even if the mix carried out is *perfectly secure*.

In order to prevent the server from grouping the users as it wishes, we have all parties run a type of joint coin tossing protocol so that the t parties controlled by a malicious server are uniformly distributed within all the groups running the shuffle. Let N denote the overall number of parties in the system, let t denote the overall number of parties under the control of the malicious server, and let n be the size of each group running the shuffle. Our coin-tossing protocol uses two random oracles H_1 and H_2. Each party P_i sends $H_1(IP_i, PK_i, r_i)$ to the server to be posted (where r_i is a long random string). Then, the groups are formed by applying H_2 to all the values $H_1(IP_1, PK_1, r_1), \ldots, H_1(IP_N, PK_N, r_N)$. Denote the output of H_2 by $o = (o_1, \ldots, o_N)$ where each o_i is of length $\log N$. Letting o_i be the temporary name of party P_i, we have that the output of H_2 induces an order on the parties by taking the lexicographic ordering of the temporary names. Using this order the users are grouped into groups of size n. Observe that the server can choose the r_j values in $H_1(IP_j, PK_j, r_j)$ after it received all of the honest parties' H_1 values (where P_j is a party under its control). Furthermore, it can do so many times in an attempt to obtain a "bad group" in which all but one party are under its control We therefore need to make sure that the probability that a group is "bad" is very small (e.g., 10^{-40}). This will ensure that the server, after seeing the inputs from the honest users, still cannot find input values that would create a "bad group" in sufficient time. The reason that we use the random oracle H_1 in the process of sending the inputs, instead

of just having the parties send (IP_i, PK_i, r_i) is in order to protect the identities of the users. Specifically, the server $\mathcal{S}$ will send the relevant IP addresses only to the relevant group, and so only the server $\mathcal{S}$ providing the service knows the history of which party participated in each group. As we will see below, it is important to prevent this information from being leaked, especially to the web search engine. Otherwise, statistical attacks can be carried out; see below for more details. The group setup appears in Protocol 6.

Protocol 6 (Group setup protocol)
Let H_1 and H_2 be two random oracles where $H_1 : \{0,1\}^* \to \{0,1\}^k$ and $H_2 : \{0,1\}^* \to \{0,1\}^{N \cdot logN}$. Let n be the size of each group for the shuffle. We set the initial indexing of the parties according to the lexicographical order of their IP addresses.

1. Each P_i chooses a random r_i and sends $H_1(IP_i, PK_i, r_i)$ to the center.
2. After a short predefined time everyone queries the center for the list of parties who have registered.
3. Each party computes $o = H_2(H_1(IP_1, PK_1, r_1), \ldots, H_1(IP_N, PK_N, r_N))$ and divides the result o into chunks of size $log\ N$, denoted $o_1, \ldots, o_N$. Party P_i is associated with o_i and the list is sorted according to the o_i values.
4. Grouping is carried out by taking groups of n parties according to the sorting. That is, for $i = 1, \ldots, \lfloor N/n \rfloor$, the ith group G_i is set to be the parties associated with the values $(o_{n\cdot(i-1)+1}, \ldots, o_{n\cdot i})$.
5. The center sends the IP addresses of the group members to the members of each group (i.e. each member gets only the IP addresses of the members in its group).
6. Members of each group send each other their IP address, public key and randomness that were used when registering with the center.
7. Each group member computes $H_1(IP_j, PK_j, r_j)$ for every party P_j in its group and verifies that it matches what was recorded by the center during registration. In addition, it verifies that it received the IP address of all parties that are in its group, by the computation of H_2. If no, then it sends abort to all the parties in its group.

We now analyze the security of Protocol 6. Recall that in the random oracle model, the output of H_2 is uniformly distributed every time that it is applied to a new value. We begin by analyzing the probability that a bad grouping occurs for a *given* set of values $\{(IP_i, PK_i, r_i)\}_{i=1}^N$. (Below we will analyze what this means when the server is malicious.) We call a group "bad" if it consists of $n-1$ malicious parties together with a single honest party. Clearly, this is bad because the server $\mathcal{S}$ then learns the search query of that party. The cases that a group has only a few honest parties is also quite bad, but there is still ambiguity regarding each user's search term. Furthermore, in Section 5.3 we discuss how to further improve this.

Let bad_i denote the event that the ith group is bad as defined above. We begin by computing the probability that the first group is bad; i.e., that bad_1 occurs. Since the output of H_2 is uniformly distributed, we can compute this by counting the number of ways to choose $n-1$ parties out of t malicious ones times the number of ways to choose a single honest party, divided by the total number of ways to choose a group of size n out of N parties. That is, we have:

$$\begin{aligned}\Pr[\mathsf{bad}_1] &= \frac{\binom{N-t}{1}\cdot\binom{t}{n-1}}{\binom{N}{n}} = \frac{(N-t)\cdot\frac{t!}{(t-n+1)!(n-1)!}}{\frac{N!}{(N-n)!n!}} \\ &= \frac{\prod_{i=1}^{n-2}(t-i)}{\prod_{j=1}^{n-1}(N-j)}\cdot(N-t)\cdot n \\ &= \prod_{i=1}^{n-2}\frac{(t-i)}{(N-i)}\cdot\frac{N-t}{N-n+1}\cdot n\end{aligned}$$

Noting again that H_2 is a random function, it follows that the above calculation is true for any fixed group. Thus, the above gives the probability of bad_i for every $i = 1, \ldots, \lfloor N/n \rfloor$. As we have mentioned, a grouping is bad if there exists a bad group. Thus, applying the union bound over all $\lfloor N/n \rfloor$ groups we have that:

$$\Pr[\exists\ i\ :\ \mathsf{bad}_i] \leq \sum_{i=1}^{N/n}\Pr[\mathsf{bad}_i] = \frac{N}{n}\Pr[\mathsf{bad}_1] = \prod_{i=1}^{n-2}\frac{(t-i)}{(N-i)}\cdot\frac{N-t}{N-n+1}\cdot N$$

Assuming now that $N >> t$, we have that $\Pr[\exists\ i$ s.t. $\mathsf{bad}_i]$ is approximately $(\frac{t}{N})^{n-2}\cdot N$. Concretely, consider the case of millions of users running this protocol, a malicious server $\mathcal{S}$ that controls a few thousand of them, and a group size of about 20. In this case, we have that the probability that there exists a bad group for a given set of H_1 values is smaller than $10^6\cdot(\frac{10^3}{10^6})^{18}$, which is 10^{-48}.

We stress that the above analysis alone is not sufficient. This is due to the fact that, as we have mentioned, it is possible for a malicious server $\mathcal{S}$ to modify the H_1 values many times in the aim of obtaining a bad grouping. Specifically, once all honest parties have submitted their values, the server can repeatedly modify the r_j portion of party P_j's input to H_1, where P_j is a malicious user under its control. Since any change to any of the H_1 values results in a completely different ordering of the parties (because H_2 is a random function), we have that the probability of a bad grouping is T times the above, where T equals the number of

hashes that the server can compute in the required time interval. With the above example parameters where the probability of a bad grouping is 10^{-48}, the probability that a malicious server achieves a bad grouping within seconds is very small.

Phase 2 – private shuffle of the search queries: Once the users have been grouped together, they run the *private shuffle* protocol of Section 3. However, as we discussed earlier in Section 5.1 (item 2 at the end of the section), we would like to prevent the group members from learning all the search results. This seems problematic because the parties do not know whose query they have and they must therefore broadcast the result to everyone. We overcome this problem by instructing each party to first choose a random symmetric encryption key k_j and then input the pair $wk_j = (w_j, k_j)$ to the shuffle. As we will see next, k_j will be used to encrypt the search result.

Phase 3 – query submission and private response retrieval: After the shuffle protocol is completed, each party holds a pair (w', k'). Each party then submits the search query w' to the search engine and receives back the result. The search result along with the original search term is then encrypted using the key k' with a symmetric encryption scheme (e.g., AES) and broadcast to all group members. Each party attempts to decrypt all search results; the one that decrypts correctly is its own result. In this way, each party only learns its own result and the result of one other random user. Thus, privacy of the queries is better preserved.

5.3 Additional Considerations

We now address some of the issues that concern deployment of our scheme in the real world and discuss the privacy that it provides.

Blending into a crowd: The main idea of our scheme is blending into a crowd. The fact that millions of people from all over the world can participate in the protocol provides a strong sense of privacy, but consideration should be given to the way different populations are grouped together. If 20 people from all over the world are grouped together and all submit the query in their native language, then it is easy to learn the query of each party based on the geographic location of its IP address. When deploying such a system, consideration should be given to these issues and blending into a crowd should actually be blending into a crowd of people with similar characteristics.

The size of a group: Our *private shuffle* protocol provides anonymity with respect to the size of the group; thus the bigger the group the more anonymity one enjoys. Since the size of the group affects both the number of modular operations each party needs to perform and the number of rounds in the *private shuffle* protocol, the size of the group is bounded by the computing power of the users' computers and the acceptable latency. Nevertheless, it is possible to hide in a larger group at the expense of more modular exponentiations but without increasing the number of rounds, as follows. As we have described in remark 3 after Protocol 4, if we can assume that the number of malicious parties within a group is some $t' < n$, then it suffices to run the shuffle stage for $t' + 1$ rounds. Performing a similar analysis to the one above, we have that the probability of having 19 malicious parties within a group of size 50 is actually very close to the probability of having 19 malicious parties within a group of size 20 (when the total number of parties is about a million and the total number of malicious parties is several thousand). Thus, if one can afford the additional number of modular exponentiations that comes with increasing the group size, we can enhance privacy significantly by increasing the size of the group, without paying much more in latency. Observe that in this calculation a group is "bad" if there are $t' + 1$ malicious parties. Thus, if a group is not bad, each honest party's search query is guaranteed to be hidden amongst $n - t'$ other search queries.

Lifetime of a group: Our scheme creates ad-hoc groups that can be changed over time. In terms of efficiency, it is easy to see that remaining within a group for a while saves the cost of running the group selection process. However, users may submit a query to the search engine and logout. In this case the group size would shrink and if it is too small then privacy is compromised. This can be dealt with by starting with a larger group and regrouping once the group becomes too small.

Statistical analysis and changing groups: In terms of privacy, it may seem that the more often people change groups, the more privacy they gain. However, this actually may not always be the case. Consider a central server that colludes with the web search engine. The server $\mathcal{S}$ and search engine can then run a statistical analysis to group together queries that are likely to belong to the same user (e.g., by grouping together very low-probability queries). Now, if these queries are carried out in different groups, then the server $\mathcal{S}$ can find the (most likely) unique IP address that appears in all of the different groups, and conclude that the queries originated from this address. Thus, changing groups can be problematic. (Of course, without such collusion, this problem does not arise.)

An additional privacy enhancement: The system presented above has the property that each user's search query is revealed to one other *random* group member. However, in some cases a user may prefer to be able to say which user will submit and therefore learn their query (and which users will not learn their query). We can extend our system for private web search to allow this by adding one more layer of encryption to the messages, using the public key of the designated party. Specifically, if a party P_j wishes to have party P_i be the one who submits its query, then it encrypts wk_j along with some redundancy (to verify the correctness when opening) using g^{α_i}. Then P_j executes the private shuffle protocol with the encrypted wk_j. After the messages are shuffled, each party sends the message it received to everyone else, and all parties decrypt the results. In this way, only the designated party P_i is the one that can learn wk_j and it will send the query.

Acknowledgements

We would like to thank Gilad Asharov and Meital Levy for many helpful discussions, and the anonymous referees for helpful comments.

References

1. M. Bellare and P. Rogaway Entity Authentication and Key Distribution. In *CRYPTO93*, Springer-Verlag (LNCS 773), pages 232-249, 1994.
2. J. Castellà-Roca, A. Viejo and J. Herrera-Joancomarti Preserving User's Privacy in Web Search Engines. In *Computer Comm.*, 32(13-14):1541–1551, 2009.
3. D. Chaum. Untraceable Electronic Mail, Return Addresses, and Digital Pseudonyms. *Communications of the ACM*, 24(2):84–88, 1981.
4. B. Chor, O. Goldreich, E. Kushilevitz and M. Sudan. Private Information Retrieval. *Journal of the ACM*, 45(6):965–981, 1998.
5. B. Chor and M. Rabin. Achieving Independence in Logarithmic Number of Rounds. In the 6*th PODC*, pages 260–268, 1987.
6. Y. Desmedt and K. Kurosawa. How to Break a Practical MIX and Design a New One. In *EUROCRYPT'00*, Springer-Verlag (LNCS 1807), pp. 557–572, 2000.
7. R. Dingledine, N. Mathewson and P. Syverson. Tor: The Second-Generation Onion Router. In *Proceedings of the 13th USENIX Security Symposium*, pages 303–320 2004.
8. T. ElGamal. A Public Key Cryptosystem and a Signature Scheme Based on Discrete Logarithms. In *CRYPTO'84*, Springer-Verlag (LNCS 196), 1984.
9. A. Fiat and A. Shamir. How to Prove Yourself: Practical Solutions to Identification and Signature Problems. In *CRYPTO'86*, Springer-Verlag (LNCS 263), pages 186–194, 1986.
10. M. Jakobsson. A Practical MIX. In *EUROCRYPT'98*, Springer-Verlag (LNCS 1403), pages 448–461, 1998.
11. R. Ostrovsky and W.E. Skeith. A Survey of Single-Database PIR: Techniques and Applications. In *PKC*, Springer-Verlag (LNCS 4450), pages 393–411, 2007.
12. C.P. Schnorr. Efficient Identification and Signatures for Smart Cards. In *CRYPTO'89*, Springer-Verlag (LNCS 435), pages 239–252, 1989.