User Search with Knowledge Thresholds in Decentralized Online Social Networks

Benjamin Greschbach, Gunnar Kreitz, Sonja Buchegger

User Search with Knowledge Thresholds in Decentralized Online Social Networks

Benjamin Greschbach, Gunnar Kreitz, and Sonja Buchegger

KTH Royal Institute of Technology
School of Computer Science and Communication
Stockholm, Sweden
{bgre, gkreitz, buc}@csc.kth.se

Abstract. User search is one fundamental functionality of an Online Social Network (OSN). When building privacy-preserving Decentralized Online Social Networks (DOSNs), the challenge of protecting user data and making users findable at the same time has to be met. We propose a user-defined knowledge threshold ("find me if you know enough about me") to balance the two requirements. We present and discuss protocols for this purpose that do not make use of any centralized component. An evaluation using real world data suggests that there is a promising compromise with good user performance and high adversary costs.

Keywords: Decentralized Online Social Networks, Privacy, User Search

1 Introduction

Popular Online Social Networks (OSNs) are logically centralized systems. The massive information aggregation at the central provider inherently threatens user-privacy. Data leakages, whether intentional (e. g., selling of user data to third parties) or unintentional (e. g., by attacks from outsiders), happen regularly[1]. Motivated by this insight, Decentralized Online Social Networks (DOSNs) have been proposed to mitigate the threats. When decentralizing a system, two challenges have to be met: to implement equal functionality without centralized components, and to provide user privacy under a significantly different threat model.

Here, we look at the functionality of user search, i. e., the lookup of a system-specific user identifier (e. g., a URI of a profile) based on information about the user (e. g., name, city, affiliation). The ability to search for users, in conjunction with other ways of traversing the social graph (e. g., friendlist of friends), is a basic building block of an OSN that allows users to find each other and thereby establish links.

[1] To name only two examples: Twitter leaking data from 250K users in February 2013 (http://blog.twitter.com/2013/02/keeping-our-users-secure.html), Facebook selling user data (http://www.telegraph.co.uk/technology/facebook/8917836/Facebook-faces-EU-curbs-on-selling-users-interests-to-advertisers.html).

1.1 Our Contribution

We propose and evaluate protocols to support user search in a decentralized OSN that shield user data from parties who know less than a user-specified threshold amount of information about the target. To our knowledge, formalizing the use of this consideration is a novel application of knowledge-based access control. This type of restriction was inspired by an observation by Fong et al. [7] that being able to reach a user in an OSN is an integral part of access control in such systems.

We evaluate our protocols using real world data from the U.S. census to relate the performance for legitimate users to the costs of an adversary attempting to guess unknown information.

1.2 Related Work

To the best of our knowledge the privacy-findability tradeoff has not been formally investigated in this context. The closest example is user search in Skype. However, as far as we know, their protocol has not been described in detail, but only via external measurement studies, such as one by Baset and Schulzrinne [2].

Most user search functionalities, including ours, search for users within the global user database of the OSN, independently of who searches. In contrast, we note that recently, Facebook has debuted Graph search [5], which ties searching to the social graph, and where the goal is not only to find users, but also content. Several other approaches of personalized searching for content in an OSN have also be discussed, e. g., by Bai et al. [1] in a decentralized setting.

Although designed specifically to search for users in a DOSN, some challenges are shared with constructing a general purpose search in a peer-to-peer (P2P) setting. This has been studied by e. g., Li et al. [8], and Bender et al. [3]. There is also a commercial search engine using P2P, Faroo [6]. Two differences are our focus on access control and privacy, and the significantly smaller amount of information to be indexed in our setting. Similar to these proposals, we also build upon a Distributed Hash Table (DHT) as a core component to realize our functionality.

2 Decentralized User Search Protocol

As we design search protocols for a decentralized system, we cannot assume any trusted third party or central search provider to be available. Instead, we use a DHT to register and look up search terms, as it is a common component of DOSNs. As the DHT runs on nodes participating in the system, we must also protect the privacy of the participants against these nodes.

We propose two protocols, both designed to index and retrieve information in a DHT in a protected way. Our protocols provide two operations. A *register* operation, where users enter information that allows others to find them based on certain attributes, and a *search* operation that, given a set of search terms,

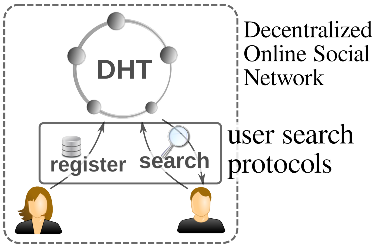

Fig. 1. System overview: The search protocols are one component of the DOSN and makes use of a DHT.

returns the set of matching user identifiers. In a next step, out of the scope of the search protocols described here, these user identifiers can be used to view public profiles, and to send a message or friend request to the found user. Figure 1 illustrates the search functionality.

2.1 Protocol Specification

We consider a searcher, who wants to find a searchee. The searchee registers searchable information about herself in the DHT by choosing a number n of attribute labels l_i (e. g., lastname, firstname, city) and assigning each one[2] value v_i. This label-value pair (denoted as attribute a_i) is mapped to a user identifier *uid* of the searchee. Upon registration the searchee specifies a threshold number t of attributes which the searcher must know in order to obtain the user identifier.

2.2 Storing Values in the DHT

The DHT holds a mapping from user attributes to user identifiers, but this mapping must be protected, also against the nodes in the DHT. To this end, we propose a protocol that alters how values are added and retrieved from the DHT. The required property is to retain standard DHT functionality, while nodes in the DHT do not learn plaintexts of keys or values.

When storing a *key-value* pair the *key* is fed into a Key Derivation Function (KDF) together with a global salt *gSalt*, yielding the DHT-key for the `put` and `get` operations of the DHT. The *value* is encrypted using a secret that is derived from a random salt *salt* and the *key* (the attribute information, in our case). The *salt* is stored together with the ciphertext on the right hand side of the mapping. In short, the mapping of a *key-value* pair in the DHT looks like this:

$$\text{KDF}(gSalt,key) \mapsto salt||\text{encrypt}_{\text{KDF}(salt,key)}(value)$$

[2] For simplicity we assume that each attribute can be assigned only exactly one value.

The *gSalt* has to be publicly available for all users to allow the lookup of any attributes. This invalidates the purpose of a salt, as pre-computing tables to reverse the left hand side becomes possible again. Nevertheless, we suggest to keep the *gSalt* as it at least requires the pre-computation attack to be targeted to each specific instance of our system and off-the-shelf pre-computed tables for the used KDF cannot be employed.

The *salt* is an individual random number different for every entry. Note that it in particular has to be different from *gSalt* as otherwise any DHT node could decrypt the *value* of items it stores, using the left hand side (without knowing the *key*).

2.3 Scheme 1: Storing all Allowed Attribute Combinations

We want a searcher to prove knowledge of a threshold number of attributes before obtaining the user identifier. One direct approach to achieve this is to map the user identifier only from attribute concatenations of the threshold length. If the searchee registered e. g., seven attributes and specified that at least four of them are necessary to find her *uid*, we would store the following $\binom{7}{4} = 35$ combinations:
$a_1||a_2||a_3||a_4 \mapsto uid$
$a_1||a_2||a_3||a_5 \mapsto uid$
...
$a_4||a_5||a_6||a_7 \mapsto uid$

where $a_i = (u_i, v_i)$, u_i attribute labels and v_i attribute values. We assume there is a canonical order of attributes (e. g., a lexicographic order of labels), and attributes are sorted by this order before concatenation.

Algorithm 1 Registration (Scheme 1)

```
$l_1, \ldots, l_n \leftarrow$ User.input("Choose searchable attribute labels (e. g., name,city,...)")
$v_1, \ldots, v_n \leftarrow$ User.input("Enter values (your name, your city,...)")
$a_i \leftarrow l_i||v_i$                                        // for $i = 1 \ldots n$
$t \leftarrow$ User.input("Enter threshold number of attributes necessary to find you.")
for all ordered sequences $a_p|| \ldots ||a_q$ of length $t$ do
    $key \leftarrow a_p|| \ldots ||a_q$
    $dhtkey \leftarrow$ KDF($gSalt$,$key$)
    $salt \leftarrow$ generateSalt()
    $value \leftarrow uid$
    $dhtvalue \leftarrow salt||$ encrypt$_{\mathrm{KDF}(salt,key)}(value)$
    DHT.put($dhtkey$,$dhtvalue$)
end for
```

Algorithms 1 and 2 describe the protocol in more detail. For registration, all attribute combinations of length t are mapped to the user identifier and stored in the DHT according to the procedure described in Section 2.2. When searching, all provided search attributes are ordered and used to query the DHT (after the Section 2.2 transformation). If the result is empty or does not contain what the user was looking for, all subsets of the provided search attributes are subsequently tried, ordered by decreasing number of elements. The final result will contain the user identifier of the searchee (and possibly more hits from other users that registered the same attributes) if the number of attributes searched for is greater or equal than the threashold specified by the searchee.

Algorithm 2 Search (Scheme 1)

```
$l_1, \dots, l_s \leftarrow$ User.input("Choose attribute labels to search for (e. g., name,city,...)")
$v_1, \dots, v_s \leftarrow$ User.input("Enter attribute values (a name, a city,...)")
$a_i \leftarrow l_i || v_i$                                              // for $i = 1 \dots s$
for $i \leftarrow s, \dots, 1$ do  // while result set is empty or the user requests more results
    for all ordered sequences $a_p || \dots || a_q$ of length $i$ do
        $key \leftarrow a_p || \dots || a_q$
        $dhtkey \leftarrow \mathrm{KDF}(gSalt, key)$
        for $salt, ciphertext$ in DHT.get($dhtkey$) do
            $uid \leftarrow \mathrm{decrypt}_{\mathrm{KDF}(salt,key)}(ciphertext)$
            add $uid$ to result set if decryption was successful
        end for
    end for
end for
```

One shortcoming of this scheme is that for sufficiently large numbers of n and t, the number of combinations might become infeasible for storage space constraints and KDF computation latencies during registration. Requiring e. g., 5 out of 20 registered attributes would yield 15504 combinations.

2.4 Scheme 2: Storing Each Attribute Individually

An alternative approach, overcoming the large number of combinations generated by Scheme 1, is to store each attribute individually. In order to require a threshold number of attributes to find the user identifier, a single attribute does not map directly to the *uid* but to an encrypted version. The key used for the encryption is based on a secret sharing scheme and one share is stored with each of the attributes. Instead of using the shared key directly, it is fed into a KDF together with an individual salt. This indirection allows us to independently tune the costs for requesting shares for one attribute (determined by the DHT latency and the KDF described in Section 2.2) and for trying to combine them (determined by the KDF used here). Furthermore, a bloom filter bf_i is attached to each share, to help finding the right shares to combine with, which is important for popular attributes with large response sets:
$a_1 \mapsto share_1 || bf_1 || salt_1 ||$ $\mathrm{encrypt}_{\mathrm{KDF}(salt_1, sk)}(\mathrm{uid})$
...
$a_n \mapsto share_n || bf_n || salt_n ||$ $\mathrm{encrypt}_{\mathrm{KDF}(salt_n, sk)}(\mathrm{uid})$
where sk can be recovered with t of the shares $share_1 \dots share_n$.

The bloom filter that is stored with each share is created using all other $n-1$ shares belonging to the same key sk. To avoid the case in which two bloom filters for a related set of shares look similar, we introduce an individual salt for each bloom filter, which is used to modify elements before insertion. Thus, with each bloom filter bf_i, we store a salt $bfsalt_i$, and when adding or querying for an element (a share in our case) in bloom filter bf_i, we first hash the element together with the $bfsalt_i$. E. g. instead of bf_i`.add(`$share$`)`, we do bf_i`.add(hash(`$bfsalt_i, share$`))`, where `hash()` is a cryptographically strong keyed hash function.

Algorithms 3 to 6 describe the protocol in more detail. When combining the shares in the search protocol, the bloom filter information is used to reduce the number of possible combinations. Note that for two sets of shares (and attached bloom filters) two reductions are possible: First a share in set one is fixed and its bloom filter is used to reduce set two. Then, for all remaining shares in set two, their bloom filters can be used to determine if they fit to the fixed share of set one. If not, they are removed from set two as well. This generalizes; for n sets, in expectancy the number of matches will be reduced by a factor of $\exp(bloomfactor, \sum_{i\in 1\dots n} 2(i-1))$, where $bloomfactor$ is the false positive probability of the bloom filter.

Algorithm 3 Registration (Scheme 2)

```
$l_1, \dots, l_n \leftarrow$ User.input("Choose searchable attribute labels (e. g., name,city,...)")
$v_1, \dots, v_n \leftarrow$ User.input("Enter values (your name, your city,...)")
$a_i \leftarrow l_i || v_i$                                        // for $i = 1 \dots n$
$t \leftarrow$ User.input("Enter minimum number of attributes necessary to find you.")
$sk \leftarrow$ generateKey()
$share_1, ..., share_n \leftarrow$ createShares($t$,$n$,$sk$)
for $i \leftarrow 1, \dots, n$ do
    $key \leftarrow a_i$
    $dhtkey \leftarrow$ KDF(gSalt,$key$)
    $bf \leftarrow$ createBloomFilter($\{share_j | j \neq i\}$) // using salted bloom filter (see text)
    $salt \leftarrow$ generateSalt()
    $k_E, k_S \leftarrow$ KDF($salt$,$sk$)                          // derive keys to encrypt and sign
    $ciphertext \leftarrow$ encrypt$_{k_E}$(uid)
    $value \leftarrow share_i || bf || salt || ciphertext ||$ MAC$_{k_S}(ciphertext)$
    $dhtsalt \leftarrow$ generateSalt()
    $dhtvalue \leftarrow dhtsalt ||$ encrypt$_{\mathrm{KDF}(dhtsalt,key)}$(value)
    DHT.put(dhtkey,dhtvalue)
end for
```

2.5 Extensions

Weighting of attributes. Some attributes might be easier to guess for an attacker than others because they have a lower entropy or represent more public infor-

Algorithm 4 Search (Scheme 2)

```
1: $l_1, \ldots, l_s \leftarrow$ User.input("Choose attribute labels to search for (e. g., name,city,...)")
2: $v_1, \ldots, v_s \leftarrow$ User.input("Enter attribute values (a name, a city,...)")
3: $a_i \leftarrow l_i || v_i$                                        // for $i = 1 \ldots s$
4: $setOfShareSets \leftarrow \emptyset$
5: for $i \leftarrow 1, \ldots, s$ do
6:     $key \leftarrow a_i$
7:     $dhtkey \leftarrow$ KDF(gSalt,$key$)
8:     $shareSet \leftarrow \emptyset$
9:     for each $(dhtSalt, dhtCiphertext) \in$ DHT.get(dhtkey) do // $> 1$ res. possible
10:        $share||bf||salt||uidCiphertext||mac \leftarrow \text{decrypt}_{\text{KDF}(dhtSalt,key)}(dhtCiphertext)$
11:        $shareSet$.add(($share, bf$)) // also remember $salt$, $uidCiphertext$ and $mac$
12:    end for
13:    $setOfShareSets$.add($shareSet$)
14: end for
15: $sk \leftarrow$ reduceAndCombineShares($setOfShareSets$, $\emptyset$)       // recovers $sk$ iff $s \geq t$
16: $salt, uidCiphertext, mac \leftarrow$ lookup values for successful shares       // see line 11
17: $k_E, k_S \leftarrow$ KDF($salt$,$sk$)
18: $uid \leftarrow \text{decrypt}_{k_E}(uidCiphertext)$                   // and validate $mac$ using $k_S$
```

Algorithm 5 reduceAndCombineShares (Scheme 2)

Input: $setOfShareSets, chosenShares$
Output: sk

```
if $|setOfShareSets| = 0$ then     // base case: try to recombine candidate shares
    $sk \leftarrow$ useShares($chosenShares$)
    if $sk$ valid then
        return $sk$                    // for simplicity, return only the first valid key
    end if
    return $None$
else                                                       // otherwise recurse
    $S \leftarrow setOfShareSets[0]$
    $SRest \leftarrow setOfShareSets \setminus S$
    for $(share, bf) \in S$ do
        $SRestReduced \leftarrow$ reduceShareSets($share, bf, SRest$)
        $result \leftarrow$ reduceAndCombineShares($SRestReduced, chosenShares||share$)
        if $result \neq None$ then
            return $result$
        end if
    end for
     // if nothing was returned yet, try not to pick any share from the current set
    return reduceAndCombineShares($SRest, chosenShares$)
end if
```

Algorithm 6 reduceShareSets (Scheme 2)

```
Input: $share, bf, setOfShareSets$
Output: $reducedShareSets$
$reducedShareSets \leftarrow \emptyset$
for $S \in setOfShareSets$ do
    for $share', bf' \in S$ do
        if not checkBloomFilter($share', bf$) then
            $S$.remove($s'$)
        end if
        if not checkBloomFilter($share, bf'$) then
            $S$.remove($share'$)
        end if
    end for
    $reducedShareSets$.append($S$)
end for
return $reducedShareSets$
```

mation that is easy to research from system external sources. We therefore want to give the users the ability to weight attributes, that is, differentiating their contribution for reaching the threshold number t. In Scheme 1, this is straightforward to implement: instead of registering all attribute combinations with a certain number of attributes, we only register combinations whose weighted sum meets the threshold.[3] For Scheme 2, more work has to be done, to implement this functionality. A possible approach is, to first pick a granularity number g for the weighting factor (the number of discrete values the weighting factor can take). Instead of storing only one share with each attribute, 1 to g shares will be stored with each attribute depending on the weight for this attribute. The threshold number will be adjusted accordingly (e. g., multiplied by g). To hide the weight of an attribute, all attributes with less than maximum weight will store dummy shares. Following a convention to first store the real shares and than append dummy shares, the additional work (for legitimate users as well as adversaries) – when trying combinations of share values – is guessing this split-point between real and dummy shares for each attribute (e. g., for $g = 10$ and 4 shares, a factor of 10000).

Dummy-attributes for Plausible Deniability. Introducing plausible deniability for leaked personal information can mitigate the consequences of privacy breaches. This can be accomplished by adding random dummy-attributes along with the real attributes. Thus, the adversary cannot be sure if an attribute that she found to be related to a user is a real one or a generated fake entry. Dummy-attributes come, however, with the trade-off of increasing false positive matches for legitimate users. Furthermore, they can be debunked by adversaries with

[3] More precisely: only those combinations where the weighted sum is greater or equal the threshold and the removal of any included attribute would yield a weighted sum less than the threshold.

background knowledge. Finally, they might make brute-force attacks easier, as dummy-attributes increase the total number of attributes but not the threshold number of required attributes.

3 Threat Model

All information that the user gives away or generates while interacting with the system has to be considered as possibly sensitive. This comprises general administrative information (existence in system, date of registration, user-identifiers), entered information during registration (attributes, i. e., label-value pairs), search query data (who searches for whom, which previously unknown attributes are used to specify search-target) and behavioural data (online times, frequency of searching/registering/updating information).

3.1 Adversaries and Their Capabilities

All agents in the system can possibly act in malicious ways. This comprises nodes involved in the DHT storage, passive traffic observers and active adversaries, i. e., malicious users that can perform search and register operations. Their capabilities range from sniffing traffic and performing traffic analysis (e. g., analyzing query sizes), crawling the DHT (performing massive search operations) or analyzing data they might store, to actively inserting data into the DHT.

Example instances of these adversary models are curious users of the system, targeted attacks from parties with background knowledge about the target user (e. g., testing specific attributes of this user, also learning from negative results), or crawling attacks that aim to harvest information for e. g., spammers, targeted advertisement or insurance companies. We cannot perform a comprehensive security and privacy analysis of the protocols, taking into account all mentioned user assets and adversary capabilities. Instead, we will focus on several specific attacks and present one of them in more detail.

3.2 Subset Crawling Attack Scenario

The proposed protocols are trying to balance findability and privacy. Thus, they cannot provide perfect protection. In the worst-case of a targeted attack, an adversary with profound background knowledge about the target user will likely succeed. For example protecting the user identifier cannot be accomplished if the adversary knows as many attributes about the target user as legitimate users do. At the same time we assume that both schemes protect the users fairly well from large-scale crawling attacks as the search space of all possible attribute combinations is too large to brute-force and the protocols transform the registered user data in a way that inferences from the publicly stored data are infeasible. If an adversary chooses to constrain her effort to only crawl the data of a specified subset of the user-base, her chances might be better. We therefore focus on what we call a Subset Crawling Attack. In this scenario, the adversary chooses a number of sensitive attributes and tries to identify all users of the system that registered that attributes. For example, the adversary could try to identify all users working at a specific company by fixing the attribute "workplace" and then brute-forcing a set of identifying attributes such as "name", "firstname" and "city".

4 Privacy Evaluation

In the following we will evaluate the costs for an adversary to perform a Subset Crawling Attack and compare this to the search costs of legitimate users. We assume that a person's first name, last name, and the city the person is located at are identifying attributes and at the same time among the most popular search attributes. These attributes might be rather public information or easy to research, so we assume that users combine them with other, less public attributes. According to the Subset Crawling Attack scenario we assume that the adversary fixes at least one of the other attributes and tries to brute-force the identifying attributes.

4.1 Data sources

To get evaluation results reflecting realistic distributions of values for identifying attributes, data from the U.S. census was used as input for the following calculations.

Distributions of U.S. *last names* were taken from [4, Table 1]. The data shows that there are 4 Million last names in total. 7 last names occur more than 1 Million times in the U.S. population. The top 3012 last names are shared by 55% of the population, the top 1 Million names are shared by 98.5%. The last name frequencies roughly resemble a power law distribution. Frequencies of U.S. *first names* were taken from [9]. The data is split into "male" and "female" first names. For our calculations we merged them assuming an equal distribution of the two categories. For U.S. *cities*, we used a dataset listing the population of all cities with more than 50000 inhabitants [10]. The data closely resembles a Zipf distribution. For the remaining population, we made a worst-case assumption of being distributed equally to cities with 50000 inhabitants (worst-case in the sense of getting less diversity for this attribute).

The validity of the evaluation results is therefore based on the assumption that the system's user base is a representative subsample of the U.S. population. In the following calculations we furthermore assume that all users registered all three attributes. A source of errors in our evaluation is that we treat these attributes as independent, because we were not able to find any statistics on joint distributions.

4.2 Brute-force Probabilities Scheme 1

We investigate the success probability of an adversary, when trying to guess identifying attributes by brute-force, i. e., searching the whole value space. We assume the adversary will try most likely values (those registered by most users according to the value distribution in the population) first. Figure 2 shows the number of combinations to test in order to cover a certain percentage of the user population. This corresponds to the costs of an adversary, as in Scheme 1, to try one combination, one KDF operation plus one DHT `get` operation are necessary. For single attributes between 180 and 3000 combinations are enough to find a target with 50% success probability (4600 to 60 Million combinations for 100%). When the combination of two attributes has to be guessed, this increases to around 10^7 combinations for 50% success probability and up to 10^{15} combinations to search the whole value space.

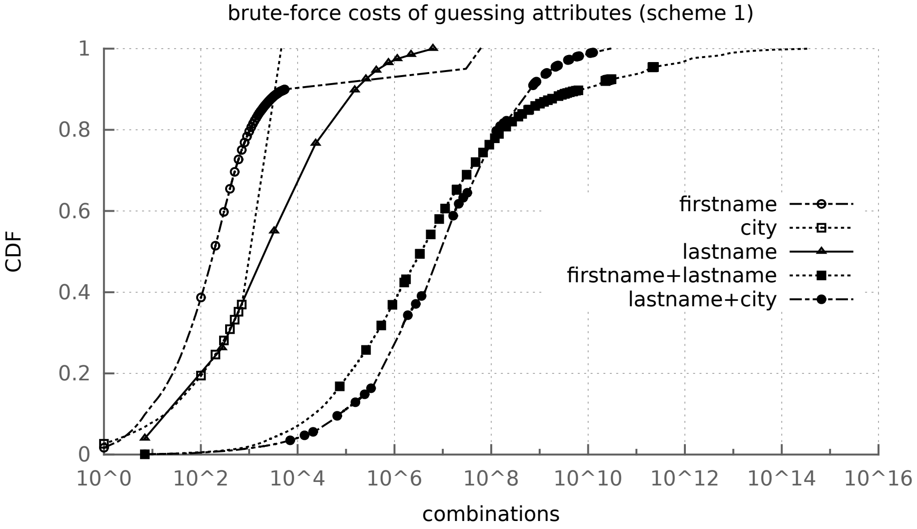

Fig. 2. CDF of brute-force success after trying a certain number of combinations (most likely ones first) for different attributes.

4.3 Brute-force Probabilities Scheme 2

In Scheme 2, bruteforcing works slightly differently. We assume, that the adversary knows some attributes (the fixed attributes that specify the subset to crawl, e. g., "workplace") and tries to guess other attributes (the identifying attributes). For each known attribute the adversary can issue a query and gets back a set of shares, each share having one bloom filter attached. Each share stems from a user who registered this specific attribute, i. e., a label-value combination (e. g., "workplace":"KTH") – several shares occur if several users registered the same combination (e. g., one from each user that registered their workplace as "KTH"). For an unknown attribute, the adversary will enumerate all possible values of the label-value combination (e. g., all possible lastname values for the attribute "lastname") and issue one DHT query each (after having performed a KDF operation to compute the DHT-key). This will result in one set of shares for each of the queries, again each share having one bloom filter attached. To know which share in each set should be picked, the bloom filters can be used to reduce the possible combinations. To test one combination, a second KDF operation (that might be tuned differently) has to be performed.

Figure 3 shows the work to be done for a legitimate user searching for three attributes and an adversary, who knows one attribute and tries to guess two unknown attributes. Additionally, the ratio of the legitimate user's cost to the adversary's cost is plotted, distinguishing two strategies of the adversary to search the value space: Either less popular values are tested first ("ratio") or more popular values first ("ratio biggest first").

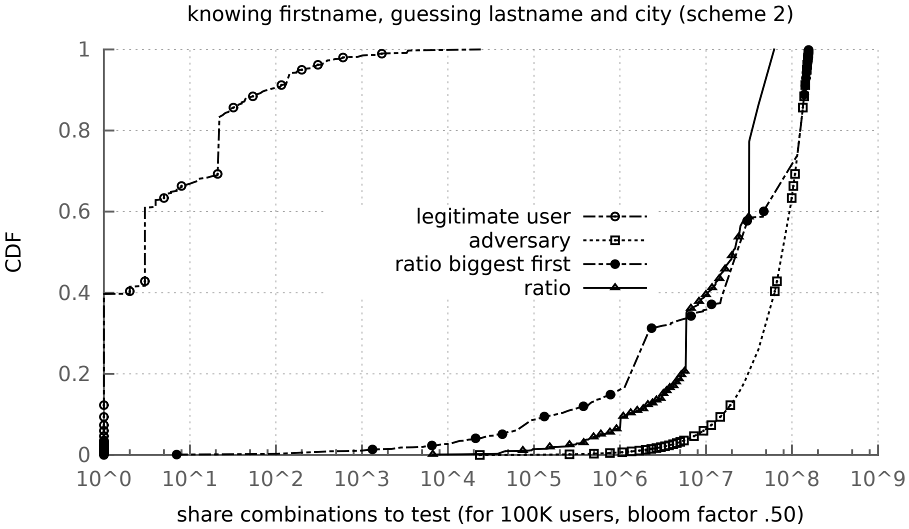

Fig. 3. Legitimate user searching for 3 attributes vs. adversary guessing 2 of them. Ratios depending on the adversaries strategy to search the value space.

4.4 Other Attacks

Existence Testing i. e., finding out if a user is registered in the system or not (without knowing enough attributes) is not possible in Scheme 1 and actively prevented in Scheme 2: Encrypting the user identifier under different keys (due to different salts) yields different ciphertexts and the bloom filters are salted differently. This is important as otherwise, searching e. g., for a certain firstname-lastname combination and getting the same ciphertext on the right hand side or similar bloom filters, reveals that there is a person with that firstname-lastname combination registered in the system, even if the person specified that more than two attributes are necessary to find her.

Search Query Data can give away information about the searcher (e. g., whom she is interested in) as well as previously unknown information about the searchee. A worst case example for the latter would be search queries that contain more information about the searchee than the searchee herself registered in the system. An adversary observing these queries can at least probabilistically learn more information about the searchee.

This attack does, however, require the adversary to reverse the KDF operation that transformed the plaintext attribute combination (denoted *key* in the pseudocode) into a derived *dhtkey*. For Scheme 1, the search protocol tries longer combinations first, which are harder to reverse. For successful search operations, this prevents the searcher from issuing queries with a lower number of attributes than specified by the searchee as threshold. Unsuccessful search operations will, however, issue eventually queries with only one attribute in the *key*. For Scheme 2, every DHT-query is derived from only one attribute, so successfully reversing the KDF might be more likely in this case. One mitigation would be to obfuscate the query origin (e. g., by using a different Tor circuit for each query), but time-correlation attacks could still be successful.

Replaying an observed search query does not help an adversary if she is not able to reverse the KDF operation (transforming a *dhtkey* value back in a *key* value), because without the *key* value, she cannot decrypt the result of the search query.

Impersonation is not prevented by our protocols as they do not try to solve the general authentication problem. Although the *uid* should be signed by the searchee and its signature validated by the searcher after it was found (not described by our protocols), this does not keep an adversary from setting up a fake profile for John Doe and register the attributes "firstname:john", "lastname:doe" into the DHT, mapping it to the *uid* of the fake profile.

5 Discussion

The results presented in the previous section describe the gap between the search effort of a legitimate user and the cost of an adversary trying to find user identifiers despite knowing fewer attributes than required. For Scheme 1, the former is constant in terms of DHT operations, the latter depends on the number and kind of unknown attributes, as shown in Figure 2. The adversary's costs for only one attribute are rather low, as expected. They can be tuned by KDF parameters but this will also affect the performance for legitimate users. The gap increases, however, combinatorially with the number of attributes the adversary has to guess. Already for two unknown attributes this might frustrate an attack: When tuning the KDF operations to take one second (delay for a legitimate user), an adversary with the same computational power as the user would need about 6 weeks to find the correct combination with 50% probability. The gap is not a global system parameter but can be tuned by each user individually (by choosing an individual threshold t for the registered information) but also depends on the adversary's knowledge about a target user. Scheme 2 can be tuned to achieve adversary costs comparable to that of Scheme 1, at the cost of slightly more work for legitimate users.

Apart from that, Scheme 1 has several advantages, compared to Scheme 2. It does not leak partial negative results, while Scheme 2, independently of any user thresholds, can reveal that a certain attribute combination is not registered in the system. For example, when searching for a certain lastname and workplace, and none of the shares of the two result sets are compatible according to the bloom filters, one learns that no user with this lastname registered this workplace. Furthermore, in Scheme 1 the adversary cannot make use of knowledge about other attributes of the user to decrease the search space for the identifying attributes. In Scheme 2, each additional attribute the adversary knows about the user, provides additional bloom filters to reduce the size of the result sets for the identifying attributes. Moreover, in Scheme 1 the user can specify even more fine-grained restrictions than only a minimum number of attributes. This makes weighting of attributes straightforward (see Section 2.5), but can even be used to explicitly exclude certain attribute combinations that the user does not want to be found by.

The advantage of Scheme 2 is the lower number of items to store in the DHT for each user. In Scheme 1, besides the higher storage load for the DHT, this is mainly a problem for registering a user, as for each of the attribute combinations also one KDF has to be computed. While this could be solved by accepting a longer delay for the registration operation and let it run in the background, the higher number of combinations might, however, also incur problems for search queries in certain cases. When over-specifying the search target in Scheme 1 (i. e., providing a number of attributes that is greater than the searchee's threshold t), successively all subsets of the attributes have to be queried while in Scheme 2 the number of DHT queries is always equal to the number of specified search attributes.

6 Conclusion and Future Work

We presented two approaches to realize a targeted user search in a DOSN. The search protocols implement a knowledge threshold, allowing the users to protect their user identifier from adversaries that do not possess enough information about them while legitimate users, who know enough about the searchee, are able to find her. We described the protocols in detail, sketched a threat model, and evaluated selected properties using real world data. The evaluation yielded insights into the brute-force costs of an adversary, which depend on the user defined knowledge threshold and the knowledge of the adversary about the target user. The results suggest that for a subset crawling attack, the proposed protocols offer promising protection against an adversary that tries to brute-force at least two or three identifying attributes.

One open problem to be investigated in future work is the possibility of combining the two presented approaches. Building on Scheme 2, several attributes that have a rather small value space could be combined in the way it is done in Scheme 1, thus avoiding the high number of combinations while still leveraging the advantages of Scheme 1.

7 Acknowledgements

Oleksandr Bodriagov and Guillermo Rodríguez Cano contributed to joint discussions of the ideas in Section 2. Some of the ideas were also discussed with Thomas Paul.

References

1. Bai, X., Bertier, M., Guerraoui, R., Kermarrec, A.M., Leroy, V.: Gossiping personalized queries. In: Manolescu, I., Spaccapietra, S., Teubner, J., Kitsuregawa, M., Léger, A., Naumann, F., Ailamaki, A., Özcan, F. (eds.) EDBT. ACM International Conference Proceeding Series, vol. 426, pp. 87–98. ACM (2010)
2. Baset, S., Schulzrinne, H.: An analysis of the Skype peer-to-peer internet telephony protocol. CoRR abs/cs/0412017 (2004)
3. Bender, M., Michel, S., Triantafillou, P., Weikum, G., Zimmer, C.: Minerva: Collaborative p2p search. In: Böhm, K., Jensen, C.S., Haas, L.M., Kersten, M.L., Larson, P.Å., Ooi, B.C. (eds.) VLDB. pp. 1263–1266. ACM (2005)
4. David L. Word, Charles D. Coleman, R.N., Kominski, R.: Demographic aspects of surnames from census 2000 (2000), http://www.census.gov/genealogy/www/surnames.pdf
5. Facebook: Introducing graph search (2013), https://www.facebook.com/about/graphsearch
6. Faroo: P2P search (2013), http://www.faroo.com/hp/p2p/p2p.html
7. Fong, P.W.L., Anwar, M.M., Zhao, Z.: A privacy preservation model for facebook-style social network systems. In: Backes, M., Ning, P. (eds.) ESORICS. LNCS, vol. 5789, pp. 303–320. Springer (2009)
8. Li, J., Loo, B.T., Hellerstein, J.M., Kaashoek, M.F., Karger, D.R., Morris, R.: On the feasibility of peer-to-peer web indexing and search. In: Kaashoek, M.F., Stoica, I. (eds.) IPTPS. LNCS, vol. 2735, pp. 207–215. Springer (2003)
9. U.S. Census Bureau, P.D.: Genealogy data: Frequently occurring surnames from census 1990 (1990), http://www.census.gov/genealogy/www/data/1990surnames/names_files.html
10. U.S. Census Bureau, P.D.: Table 1. annual estimates of the resident population for incorporated places over 50,000, ranked by july 1, 2011 population: April 1, 2010 to july 1, 2011 (sub-est2011-01) (2012), http://www.census.gov/popest/data/cities/totals/2011/tables/SUB-EST2011-01.csv