Efficient Privacy-Preserving User Matching with Intel SGX

Junwei Luo(✉), Xuechao Yang, Xun Yi, Fengling Han, and Andrei Kelarev

School of Computing Technologies, RMIT University,
Melbourne, VIC 3000, Australia
{junwei.luo,xuechao.yang,xun.yi,fengling.han,
andrei.kelarev}@rmit.edu.au

Abstract. User matching is one of the most essential features that allows users to identify other people by comparing the attributes of their profiles and finding similarities. While this facility enables the exploration of friends in the same network, it poses serious security concerns over the privacy of the users as the prevalence of modern cloud computing services, companies outsource computational power to untrusted cloud service providers and confidential data of the users can be exposed as the data storage is transparent in the remote host server. Encryption can hide the user data, but it is difficult to compare the encrypted profiles. While solutions utilising the homomorphic encryption can overcome such limitations, they incur significant performance overhead, which is impractical for large networks. To overcome these problems, we propose an efficient privacy-preserving user matching protocol with Intel SGX. Other techniques such as oblivious data structure and searchable encryption are deployed to resolve security issues that Intel SGX has suffered. Our construction relies on secure hardware which guarantees the integrity and confidentiality of the code execution, which enables the computation of similarities between the profiles of the users. Moreover, our protocol is designed to provide protection against several types of side-channel attacks. The security analysis and experimental results presented in this paper indicate that our protocol is efficient, secure, practical and prevents side-channel attacks.

Keywords: Privacy-preserving user matching · Intel SGX · Oblivions data structure · Searchable encryption · Social network security

1 Introduction

Modern social networks such as Facebook and Instagram allow people to connect with each other in a virtual space, enabling them to make new friends in different parts of the world. While this facility enables the exploration of friends in the same network, it poses serious security concerns over the privacy of the users. With the increasing demands on modern cloud computing services, companies outsource computational power to untrusted cloud service providers and

W. Xiang et al. (Eds.): BROADNETS 2021, LNICST 413, pp. 92–111, 2022.
https://doi.org/10.1007/978-3-030-93479-8_6

confidential data of the users can be exposed as the data storage is transparent in the remote host server. Social network service providers have become targets of adversaries who try to exploit the service and attempt to steal sensitive information, data breaches for social networks have greatly impacted the society in a negative way. A group named "The Impact Team" breached a dating social network Ashley Madison and released over 30 million of users' data to the public in July 2015. Several unconfirmed suicides that occurred in the following days were believed to be linked to the cyber incident as some victims were not able to undertake the social impacts and public shaming that implied to them.

Such data breaches raise concerns about the privacy of social networks as the network holds sensitive information that users might want to keep it secret. While encryptions seem to be an answer to prevent such cyber incidents, encrypted data are difficult to process and features such as user matching that allows the discovery of people with similar interests are hard to implement. So far, many efforts have been made to overcome such limitations. Agrawal et al. [1] proposed a model that allows certain information to be exchanged without disclosing others. Freedman et al. [7] proposed a solution using polynomial evaluation to allow user profile matching to be done securely. The paper [7] was extended later by [10] that adds supports for set intersection matching. These solutions support binary matching, in which matching result shows whether or not users have a matching. Yi et al. [28] introduced a user matching scheme that uses multiple parties to compute the shared secret via homomorphic encryption, and later, extended in [29] to improve efficiency. The solution was implemented in [12] using Intel SGX to improve security.

However, solutions mentioned above are far from practical as they impose expensive computational overhead, due to the constraints of the involved cryptosystems. In [12], the authors proposed a user matching protocol that takes advantage of Intel SGX. This has made a reduction to the computation overhead compared to the use of traditional asymmetric cryptosystems. The results of [12] show that the system remains impractical and further improvements to security and performance remain desirable.

The present paper proposes a novel approach to the design of a privacy-preserving user matching protocol by simultaneously employing Intel SGX, oblivious data structure, searchable encryption and other techniques for enhancing security and performance. Our novel solution aims to provide user profile matching with better accuracy and ensure protection against several types of side-channel attacks while minimising the computational cost.

It is known that SGX is vulnerable to many side-channel attacks (see, [4, 8, 13, 26, 27]). Side-channel attacks such as cache-timing attack can be used to extract the secret key from a running program by observing the behaviour of a cryptographic algorithm to determine the secret [8]. Therefore, it is crucial to include protection against side-channel attacks. Our new protocol is designed to address these issues.

A brief overview of our solution is presented as follows: a user on a social network platform has a profile with one or more attributes representing the characteristics or interests of the user. All the attributes of the user are encrypted

with the own symmetric key of the user that was generated during the registration process, and is known to the user and the secure enclave thereby guaranteeing integrity. All users communicate with Intel SGX via secure communication channel. Intel SGX securely places data into the untrusted domain after encryption to minimise leakages. All sensitive computation is done in the secure enclave where the integrity and privacy of code execution are guaranteed. We strengthen the confidentiality of our system by introducing oblivious mechanisms to prevent unnecessary leakage via side-channel attacks.

To summarise, our contributions include the following:

1. We design a novel privacy-preserving user matching protocol, which has several stages incorporating Intel SGX, oblivious data structure, searchable encryption and other security-strengthening techniques to expedite the computation of privacy-preserving profile matching on social networks.
2. Our protocol is designed to provide protection against several types of side-channel attacks. While one component in searchable encryption leaks information about users, we incorporate a data oblivious scheme to mitigate the leakage.
3. We implement a prototype of proposed protocol and evaluate the performance of our protocol in terms of memory usage, matching time, and compare its effectiveness with previous alternative options.

The organisation of this paper is as follows: Sect. 2 presents the related work, followed by the overview of system architecture in Sect. 3. The proposed protocol is presented in Sect. 4. Section 5 details the security analysis. Section 6 discusses the performance of the proposed protocol and Sect. 7 concludes the paper.

2 Related Work

Since the introduction of Intel SGX in 2014, many researchers have been exploring the possibility of utilising such feature to enhance security of their products [2,3,6,15,17,25]. Pbsx [9] is a secure boolean query retrieval model implemented using Intel SGX to protect the privacy of users data in an outsourced cloud environment. Pbsx corporates with components such as Bloom Filter, ORAM and Bitmap to provide query matching efficiently with various techniques to protect against cache-timing attacks [8]. Lightbox [6] is a software middleware that acts as a firewall to provide a pattern matching for detecting malicious packets within the secure hardware, allowing it to be outsourced to the cloud infrastructure where it can provide computational power as requested instead of building a server in-house.

Apart from the works that focus on data matching aforementioned, Intel SGX has also been applied to various industries where a trusted computation is needed. SCONE [3] is a OS-level virtualisation container project that utilises Intel SGX to preserve the privacy of the program running within in a cloud environment. Container such as Docker is a container-as-a-service that facilitates the development process by packing any dependencies required for a product to be

deployed into a container and share between developers. VC3 [17] is a data analysis framework based on MapReduce that utilises Intel SGX for outsourced big data analysis with strong security guarantees. All data is encrypted and sent to the secure enclave where the integrity and confidentiality are held, using Remote Attestation to establish secure communication channel for key exchange as well as returning final results to the client, all is done without leaking information about what is being computed.

While researchers take advantage of secure enclave to enhance overall security of their proposed solutions, SGX has been known to be vulnerable for many side-channel attacks [4,8,13,26,27] that can leak information inside the enclave. Side-channel attacks such as cache-timing attack can be used to extract the secret key from a running program by observing the behaviour of a cryptographic algorithm to determine the secret [8]. Therefore, many works that focus on mitigating the side-channel attacks have been proposed [14,16] that provide mechanisms against various side-channel attacks such as cache-timing attack by hiding the access pattern with the help of Oblivious Random Access Memory (ORAM), or randomising the memory location whenever an enclave is created [18] using Address Space Layout Randomization technique (ASLR). Other side-channel attacks such as power analysis [26] has also been proven possible but is less practical compared to other attacks aforementioned.

3 System Architecture

3.1 System Design

Our new privacy-preserving matching protocol consists of three components: Users, a matching server and a secure enclave that co-exists within the matching server. Secure enclave is employed with Intel SGX to facilitate the cost of creating a Trusted Execution Environment. Figure 1 demonstrates the system architecture of proposed matching protocol. Users will be communicating directly with secure enclave via secure communication established using remote attestation. User data will be processed within the enclave to ensure its confidentiality and the matching server is solely responsible for persisting data for the enclave.

Matching Server. There exists a matching table MT in matching server that is constructed on basis of a SSE scheme [22]. The purpose of MT is to facilitate the process of encrypted queries about user matching in untrusted cloud environment, whereas sensitive information such as encryption keys for users are preserved within secure enclave as it guarantees integrity and confidentiality of the data within. As the profile of a user is processed within the enclave and sent back to matching server, it is persisted into the database for long-term storage.

Secure Enclave. The secure enclave will be responsible for computing sensitive information such as profile matching and constructing search tokens to enable

SSE scheme. There exists a key table KT within the secure enclave where it denotes all symmetric keys of users registered to the network. As the memory limitation implied to Intel SGX, performance will inevitably be hindered to the system, once the memory usage has reached to the point where the enclave triggers paging to encrypt and swap out unused memory to untrusted domain. We adopt an ORAM scheme to our key table KT later in our *Extended* scheme, where it adds communication overhead in flavour of mitigating memory paging issues. A simplified system workflow below demonstrates our matching protocol.

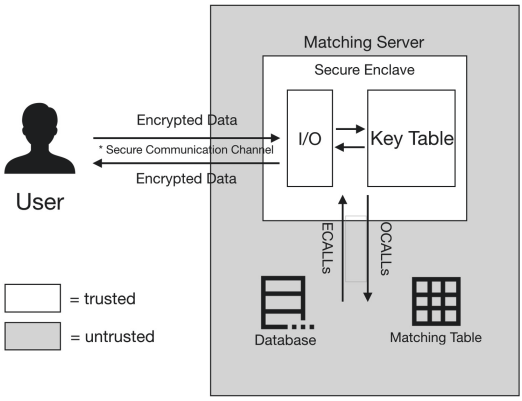

Fig. 1. System architecture.

Initialisation: Users begin by registration with secure enclave via secure communication channel after running remote attestation with the enclave to ensure the authenticity and validity of the system. A symmetric key for the registered user is randomly generated within the enclave and sent to the user. The symmetric key is managed within the secure enclave using a key table KT that contains tuples (U, K_U) where U is a unique identifier for the registered user, and K_U is the symmetric key for user U. A master key for secure enclave K_{SGX} is generated and managed within the enclave for encrypted querying. After registration, a profile P_U for the registered user is created, where P_U might contain one or more attributes A_U that represent the registered user U. Upon completing profile creation, P_U is then encrypted with K_U and sent to the enclave.

Processing: Once the profile is received by the secure enclave, it processes the profile by decrypting each attribute within the profile, and computes a search token that corresponds to the attribute. We adopt the SSE scheme [22] as a data structure that allows sensitive information to be stored securely in the matching server due to several constrains that imply to the secure enclave. Each search token corresponds to an attribute and acts an index to the SSE scheme, followed by encrypted user identifiers that can be used to query database which stores all encrypted profiles. However, the scheme that we adopt does not offer mechanisms against timing analysis, a technique that has been widely studied

to exploit secure enclaves. We solve this issue by introducing a data-oblivious access scheme into a component within the SSE scheme where it is affected by timing analysis.

Matching: As the user matching begins, user specifies one or multiple attributes that describe her preferences in friend matching, attributes are encrypted and transmitted to the secure enclave. Upon receiving the preference, secure enclave decrypts and computes the corresponding search token for each attribute. Search tokens are then passed to matching server where the matching table exists and queries databases based on result sets from the matching table. Secure enclave is able to evaluate the result set from matching server to determine if one shares similar interests to the user who requests a profile matching. While our proposed protocol supports different matching algorithms based on the type of attributes, such as Euclidean Distance for numerical values, exact matching and Hamming Distance for others. Other schemes that allow semantic matching between two words can also be adopted to evaluate similarity of attributes.

3.2 SSE Construction

Searchable Symmetric Encryption (SSE) allows clients to outsource encrypted data to the cloud service providers while maintaining the ability to search for the encrypted data (cf. [5,22]). Such constructions have been adopted to enable secure pattern matching such as deep packet inspection (DPI) [19], mobile cloud networks [11] and so on.

A SSE scheme defines the following operations:

Init: A security parameter λ, $K : \{0,1\}^{\lambda}$, a pseudorandom function that generates search token, $\langle k1, k2\rangle = \mathsf{H}(K, w)$ where the search token $ST = \langle k1||k2\rangle$ and $||$ denotes concatenation.

Create: Client computes an inverted index for every word w within a document set DB, where $Inv(DB) = (w_i, (id_{i_0}, id_{i_1}, ...), ...)$ that denotes the occurrence of word w in id_i document. Client then builds an encrypted list L as follows: for every word w within the $Inv(DB)$, a counter c for word w is initialised and set to 1 and computes the following: $\langle k1, k2\rangle = \mathsf{H}(K, w)$, $\ell = \mathsf{H}(k1, c)$ and $V = \mathsf{Enc}_{k2}(id_i)$. The process continues for every id that links to the word w, and the counter c is incremented by 1 for every computation until all id has been processed. Tuple $\langle \ell, V\rangle$ is stored in the list L and uploaded to the cloud. An index map manages all c for every word w to facilitate insertion when a new document is added.

Search: The client inputs K, w to generate $\langle k1, k2\rangle = \mathsf{H}(K, w)$, the pair is sent to the server. Having received it the server computes $V = L.get(\mathsf{H}(k1, c))$, $id = \mathsf{Dec}_{k2}(V)$ and returns the appropriate documents with id attached to. The update step is usually performed by the clients downloading the list L stored in the cloud and reconstructing a new L'.

3.3 Design Choices

We adopt the SSE construction from [22] as to reduce overall computational overhead implied using traditional homomorphic encryptions [12,29] while being able to search for information needed for matching. Original work of adopted SSE assumes that the client is honest and does not leak information that enables timing and pattern analysis, whereas in our protocol, secure enclave acts as the client which is situated in an untrusted environment. An adversary may observe the access pattern to the index map to deduce information based on access distribution. Works have been made to extract information simply by observing access patterns [4,8,21,27]. We mitigate the issue by introducing a pattern-hiding technique that obfuscates access patterns of the index map, which will be discussed in the next section. Our construction allows such table to be outsourced in matching server due to the memory constrains implied to the secure enclave, while ensuring that adversaries cannot learn information by observing the access pattern to the index map.

3.4 Adversary Model

We define a powerful adversary controlling the host environment, where the secure enclave is deployed, including OS resource scheduling for the underlying applications. The adversary can intercept, record, and monitor the use of hardware resources and attempt to disclose the secret information running in the secure enclave via side-channel attacks. Only the code running in the secure enclave is trusted, whereas the rest remain untrusted. We assume that the execution environment that Intel SGX introduces guarantees the integrity, confidentiality and consistency of code execution. The adversary can behave maliciously by sending false requests to both the matching server and the secure enclave trying to reveal some information about the user and corresponding keys. However, we assume that the adversary cannot extract the data from a running processor via physical attacks. Other side-channel attacks such as power analysis are outside the scope of this paper.

4 Privacy-Preserving User Matching Protocol

4.1 Construction of Improved Index Map

As discussed aforementioned that information can be deduced by observing access patterns of the data structure, we adopt the idea of data obliviousness to components of the SSE that could potentially suffer from such attacks. More specifically, we modify the index map to enable oblivious operations in order to prevent timing attacks. In our matching protocol, the index map is to group users based on their interests, where the interests will serve as an index, followed by the group that contains user ids. Originally the index map is constructed similarly to a hash table, which offers better performance in the expense of memory

usage and pattern leakage, we present our data oblivious scheme on basis of binary search tree, similar to [20,23].

Construction: The original work of index map is constructed using a data structure similar to a hash table, we modify this by adopting a binary search tree T with additional mechanisms to tighten security. Unlike an ordinary binary search tree, each node within the tree is a bucket whose size is the multiple of a search token and its corresponding counter. This ensures that the size of each bucket in the tree is the same. Given a full binary tree T of depth D, we have 2^D leaf nodes and $P(l)$ is used to denote the path from root of the tree T to the leaf node l.

Within the secure enclave, there is a deterministic function $DF()$ that takes as input a search token ST and decides a leaf node l which the search token ST is assigned to. When a record is inserted, secure enclave first computes $l = DF(ST)$ which returns a leaf node indicating where the record is randomly inserted into one of the bucket along the way from root to the leaf node. Matching server then performs a scan from the root of the tree to the leaf node l, and stores every bucket along the way to the leaf node as $P(l)$.

To achieve obliviousness, every bucket will be examined in order to prevent attacks that utilise cache timing analysis. Searching and deletion also perform similarly to insertion, where $l = DF(ST)$ is computed in secure enclave and sent to matching server, $P(l)$ is then obliviously retrieved and sent to the secure enclave, which then the secure enclave iterates all buckets and retrieves or removes c that corresponds to the search token ST. It is worth noting that in order to prevent timing attacks, there is no early termination when performing either insertion or searching, meaning that regardless of whether or not the desirable result is found, the system continues to go through the rest of the bucket. If the operation is deletion, secure enclave simply returns the $P(l)$ after deletion, and lets matching server to overwrite its stale path.

Algorithm 1: Pattern-hiding Index Map Insertion

Secure enclave:
    Computes $l = DF(ST)$ and sends l to the matching server.
Matching server:
    Retrieves all buckets in path $P(l)$ from T
    Sends $P(l)$ back to secure enclave.
Secure enclave:
    $r = UniformRandom(\{0,1\}^D)$
    foreach bucket **in** $P(l)$ **do**
        If *bucket.depth* equals to r **then**
            $bucket = bucket.add(ST, c)$
    end
    sends $l, P(l)$ to matching server.
Matching server:
    Overwrites $P(l)$ back to the l leaf of tree T.

Algorithm 2: Pattern-hiding Index Map Search

```
1  Secure enclave:
2      Computes l = DF(ST) and sends l to the matching server.
3  Matching server:
4      Retrieves all buckets in path P(l) from T
5      Sends P(l) back to secure enclave.
6  Secure enclave:
7      foreach bucket in P(l) do
8          If bucket.contain(ST) then
9              c = bucket.getAndRemove(ST)
10     end
11     Runs code 7 - 14 in Algorithm 1 to overwrite the previous path.
```

4.2 Construction of Privacy-Preserving Matching Protocol

Initialisation. At initialisation, matching server initialises components such as database, matching table and index map for matching server, as well as setting up a secure enclave with Intel SGX that co-exists within matching server. Inside the secure enclave there exists a master key K_{SGX} and a key table KT that stores tuples consisted of $\langle U_i, K_{U_i} \rangle$, where U_i is the identity of a user and its encryption key K_{U_i} respectively.

Preprocessing. When a user U_i signs up on the network, a remote attestation is executed with the secure enclave to ensure the validity and authenticity of a running enclave. This process establishes a secure communication channel between the enclave and U_i. A symmetric key K_{U_i} is created and stored in the key table that exists in the enclave. The key K_{U_i} is sent back to the user U_i, and U_i can send her basic information along with her profile P_{U_i} to the enclave via secure communication channel.

A profile P_{U_i} of U_i in modern social networks is a digital representative of the user U_i that usually contains one or more attributes A_j^i, such as age and interests. The U_i can upload one or more A_j^i to the enclave every time, when finishing creating a profile, P_{U_i} is encrypted using the key K_{U_i} received previously to encrypt her profile,

$$\mathsf{Enc}_{K_{U_i}}(P_{U_i}) = (\mathsf{Enc}_{K_{U_i}}(A_1^i), \mathsf{Enc}_{K_{U_i}}(A_2^i), \cdots, \mathsf{Enc}_{K_{U_i}}(A_m^i)) \tag{1}$$

where m denotes the number of attributes. The encrypted profile $\mathsf{Enc}_{K_{U_i}}(P_{U_i})$ is then sent to secure enclave via the secure communication channel established after successful remote attestation. Preprocessing begins when the enclave receives encrypted profile $\mathsf{Enc}_{K_{U_i}}(P_{U_i})$ from U_i. A key for decrypting this profile is retrieved from key table using user's identity.

As the content of an attribute within the profile is vetted and padded, secure enclave generates a search token as follows: $ST_j = \mathsf{H}(K_{GSX} \,||A_j^i)$, where K_{SGX}

is the master key for the enclave, and A_j^i is the j-th attribute of a profile P_{U_i}. This continues for every attribute within the profile P_{U_i}. For instance, the attribute that describes user's age is $A = 'AGE : 21'$, such attribute is appended to the master key of the enclave and digested using a Pseudorandom function $ST = \mathsf{H}(K_{SGX} \mid\mid A)$, the result of the digest function gives us a search token ST for an attribute that describes age of 21. The secure enclave then follows Algorithm 2 to computes a leaf node l for index map for searching and/or insertion.

Once a search token is computed, secure enclave queries the index map stored in matching server to check if a counter c can be found for this specific search token. Index map stores tuples of $\langle ST, c\rangle$, where ST is the search token and c is the number of users with the same attribute settings. For example, the token computed above $ST = \mathsf{H}(K_{SGX} \mid\mid A)$ describes an attribute of age with the value of 21, and c in this context is the number of users that shares the same attribute settings, namely age of 21.

When the leaf node l is given to the index map, it performs oblivious search over the map to check if c is present in the map. If c is not found, secure enclave runs codes 7 to 14 from Algorithm 1 to create a new tuple $\langle ST, c\rangle$, where the value of c is set to 1. This indicates that there exists one user who shares an attribute setting of age of 21 on the network. Such mechanisms allow secret to be stored and indexed in an untrusted environment without letting the server know about the content of the search token.

Following the previous example, if a tuple is found in the index map, secure enclave replaces the tuple with dummy data, updates the counter by $c = c + 1$ if the enclave receives a new profile that contains the same attribute settings, shifts the tuple into different position of the path and overwrites the entire path back to the index map.

Upon completion, secure enclave then encrypts the user id as follows: ST is broken down into two, where $ST = (ST_a \mid\mid ST_b)$, a new record A new record $\langle CA, CB\rangle$ is computed and inserted into the matching table MT, where $CA = \mathsf{H}(ST^a||c)$ and $CB = \mathsf{Enc}_{ST^b}(U_i)$, where c is the counter retrieved from the index map using the corresponding ST.

This process continues for every attribute that exists in P_{U_i} until no remaining attribute requires processing, and the encrypted profile itself $\mathsf{Enc}_{K_{U_i}}(P_{U_i})$ is sent back to the matching server.

To summarise, any new attributes submitted by users are classified to a group if the attribute settings are the same (e.g., same age) as each attribute results in the same search token ST. The index map is updated based on the search token ST generated, and returns a counter c that relates to ST, which is used to generate secret record $\langle CA, CB\rangle$ for the matching table. ST_x is split into ST_a and ST_b, $ST = (ST_a||ST_b)$ and the secret record $\langle CA, CB\rangle$ can be computed based on the counter c returned from the index map, where $CA = \mathsf{H}(ST_a||c)$ and $CB = \mathsf{Enc}_{ST_b}(U_i)$ respectively. Each $\langle CA, CB\rangle$ is sent to the matching server where the record persists in the matching table. Algorithm 3 demonstrates this procedure.

Algorithm 3: User registration.

User U_i:
  Remote attestation with the SGX enclave. Retrieves K_{U_i}.
  Computes $\mathsf{Enc}_{K_{U_i}}(P_{U_i}) = (\mathsf{Enc}_{K_{U_i}}(A_1^i), \cdots, \mathsf{Enc}_{K_{U_i}}(A_m^i))$.
  Sends $\mathsf{Enc}_{K_{U_i}}(P_{U_i})$ to the enclave.
SGX enclave:
  Receives $\mathsf{Enc}_{K_{U_i}}(P_{U_i})$ from U_i.
  Reveals $P_{U_i} = A_1^i, A_2^i, \cdots, A_m^i$ using K_{U_i} ▷ see Sect. 4.2.
  for $j = 1$ **to** m **do**
    Computes $ST = \mathsf{H}(K_{SGX}||A_j^i))$
    $l = DF(ST)$
    If l exists in the index map **then**
      Inserts $(ST, c+1)$ for this ST
    Else
      Inserts $(ST, 1)$ in the index map.
    end
    Obtains the latest counter c for the ST.
    Splits ST as $ST_a||ST_b = ST$.
    Computes $\mathsf{H}(ST_a||c)$, $\mathsf{Enc}_{ST_b}(U_i)$, sends to the matching server.
  end
The matching server:
  Stores $\mathsf{H}(ST_a||c)$, $\mathsf{Enc}_{ST_b}(U_i)$ received from the enclave.
  Stores $\mathsf{Enc}_{K_{U_i}}(P_{U_i})$ in the user profile table

Matching. When user U_i wants to find other users who have similar attributes (e.g., age), U_i submits his/her own attribute value (e.g., age = 21) to the SGX, and a list of corresponding users is returned by the matching server. To do that, U_i specifies one or more attributes indicating the group of users U_i is looking for, encrypts it with K_{U_i} and sends the encrypted attributes along with U_i's profile to the enclave.

Once the enclave receives a matching request from U_i, each attribute can be revealed in the secure enclave by using the corresponding key of U_i from the key table (refer to Sect. 4.2). After that, it is similar to the registration process where a search token is computed based on the attribute A_j^i from the request and stored in a list $ST[]$. When the computation completes, $ST[]$ is sent to the matching server. In order to find the related users that contain the same attribute, the matching server splits the search token ST into two, ST^a and ST^b respectively, such that $ST = (ST^a||ST^b)$, for every ST in $ST[]$. A counter c is initialised and set to 0, and CA can be computed as $\mathsf{H}(ST^a||c)$ where c increments until $\mathsf{H}(ST^a||c)$ is not found in the matching table. Each $\mathsf{H}(ST^a||c)$ that were found in the matching table returns an encrypted user identity Enc_{ST^b}(user ID), which can be decrypted using the key ST^b. This process continues for every ST that exists in the list $ST[]$, when the computation completes, the matching server queries its database to retrieve data using the decrypted user identity, a list

of encrypted profiles $RST[]$ is then returned from the database and is sent to enclave for similarity computation.

Algorithm 4: User matching.

User U_i:
  Computes $\mathsf{Enc}_{K_{U_i}}(A_1^i), \cdots$ as PT, send PT and $\mathsf{Enc}_{K_{U_i}}(P_{U_i})$ to SGX.
SGX enclave:
  Receives PT and $\mathsf{Enc}_{K_{U_i}}(P_{U_i})$ from U_i.
  Reveals $PT = \{A_1^i, \cdots\}$ using K_{U_i} as PT'. ▷ see Sect. 4.2
  Decrypts P_{U_i} as P'_{U_i} using K_{U_i}.
  Creates an empty list of search tokens $ST[] = \emptyset$.
  foreach A_j^i in PT' **do**
    Computes ST (e.g., $ST = \mathsf{H}(K_{SGX}||'Age : 21')$).
    Adds the computed ST to the list $ST[]$.
  end
  Sends the list $ST[]$ to the matching server.
The matching server:
  Receives $ST[]$ from the SGX.
  Creates an empty list of user IDs $U[] = \emptyset$.
  foreach ST in $ST[]$ **do**
    Sets counter $c = 0$.
    Splits $ST = ST^a||ST^b$.
    while $\mathsf{H}(ST^a||c)$ exists in the matching table **do**
      Reveals user ID U from $\mathsf{Enc}_{ST^b}(U)$ using ST^b.
      Adds revealed U to the user list $U[]$.
      Updates counter $c = c + 1$.
    end
  end
  Query the database with user IDs in $U[]$ and stored in $RST[]$.
SGX enclave:
  Receives $RST[]$ from the matching server.
  Defines an empty set $R[] = \emptyset$ of results.
  foreach P in $RST[]$ **do**
    Obtains K_p for profile P.
    $P' = \mathsf{Dec}_{K_p}(P)$.
    Computes the similarity score between P' and P'_{U_i}, stores it in $R[]$.
  end
  Sorts $R[]$ and sends it to the requesting user U_i.

As the list $RST[]$ consists of users with the similar attribute settings as U_i requests (e.g., same age), it is imperative to realise what users that potentially match better with U_i. To find out, similarity computation is required. Once the enclave receives the list $RST[]$, each profile is decrypted using the corresponding key from the key table and the decrypted profile is compared with user who requests the profile matching using several distance measurements algorithm.

For attributes that are not in numerical form, Levenshtein Distance is used to measure the similarity, and Euclidean Distance is used to measure the distance of two numerical attributes. Results of similarity score of two profiles are stored in $R[]$, which is sorted when the process completes. The enclave can optionally include some public information about the user in the list $R[]$ (e.g., the name) before sending the list back to user U_i.

5 Adversary Model and Security Analysis

Our protocol relies on the trusted execution environment provided by Intel SGX. Assuming that Intel SGX guarantees the integrity of code execution and content isolation, an attacker cannot directly expose the content of the protected memory section. Potentially, an adversary could try to exploit the system using various side-channel attacks. Our proposed user matching protocol remains secure against several types of side-channel attacks.

Theorem 1. *Improved Index Map is pattern-hiding and secure against timing analysis.*

The security of improved index map is defined as a game between an adversary $\mathcal{A}$ and a challenger $\mathcal{C}$. Let $\mathcal{P}$ be the access pattern which $\mathcal{A}$ can measure. The goal of the adversary is to evaluate access pattern of the data structure in order to learn what elements are visited and used. Adversary $\mathcal{A}$ starts the game by sending two search tokens ST_0 and ST_1 to the challenger. The challenger $\mathcal{C}$ then runs Algorithm 2 and generates two access patterns $\mathcal{P}_0$ and $\mathcal{P}_1$ that $\mathcal{A}$ is able to measure. In the end, the adversary wins the game if for all measured patterns, the probability of correctly identifying whether $\mathcal{P}_0$ belongs to the search token ST_0 or ST_1 by observing the pattern generated during the runtime is non-negligible.

Proof. According to Algorithm 1, for each insertion, there exists a deterministic function DF which decides which path $P(l)$ that the data should be inserted at the position chosen uniformly at random $r = UniformRandom(\{0,1\}^D)$. More specifically, when an element is accessed, either by searching or insertion, it is moved to another position r somewhere on the way from the root of the tree to the leaf l, where r is chosen uniformly at random. Moreover, all buckets on the path l are accessed as the search begins, this forces the processor to load all data into caches and prevents cache misses when processing.

Security of the SSE Scheme: As our proposed user matching protocol employs SSE scheme [22], it inherits all security guarantees from the original work, except for the index map where our improved index map mitigates issues with timing analysis. Most SSE schemes introduce a concept of leakage function, which defines the amount of information leaked at certain stages. Our adopted SSE scheme defines leakages as follows:

$leak_s(ST, S, t)$, where ST denotes the search token that is being searched for, S denotes a set of results related to the search token ST and the time t when the search is requested.

Such SSE holds forward privacy in $leak_s()$, this means that the result set S relates to the elements added prior to occurrence of the leakage only, meaning the items added after the leakage cannot be related until a new leakage is captured.

6 Experiments and Performance Analysis

Our system was developed in C and deployed on a device equipped with Intel Core i7-7700HQ with 32 GB of DDR4 2400 MHz memory and the host OS is Ubuntu 18.04 LTS with Intel SGX SDK 2.10 and OpenSSL 1.1.3c. For symmetric encryption scheme, we choose standard AES-NI implementation as it comes with the SGX SDK that prevents against various side-channel attacks, and the message digest algorithm for computing search token uses SHA256 from OpenSSL. Table 1 presents the overall complexity of our proposed system and compares with other alternative protocols.

Table 1. Comparisons with the protocols of [29] using asymmetric encryption with multiple servers, and [12]. The number of servers involved in profile matching is S. The number of profiles is n.

	Our protocols	[12]	[29]
Profile matching cost	$\mathcal{O}(2n)$	$\mathcal{O}(2n * S)$ (Exp.)	$\mathcal{O}(2n * S)$ (Exp.)
Profile storing cost	$\mathcal{O}(n)$	$\mathcal{O}(n * S)$ (Exp.)	$\mathcal{O}(n * S)$ (Exp.)
Communication cost	2 rounds	n * S rounds	n * S rounds
Cryptosystem used	Symmetric	Asymmetric	Asymmetric
Num of party participated	Two parties	Multi-party	Multi-party
SGX-enabled	Yes	Yes	No

While Intel SGX provides a secure environment for sensitive computation at minimum cost, systems that utilise secure enclave technology usually require to minimise the attack surface, that is, to reduce the memory usage to the point where only data that is absolutely necessary for computation is placed in Enclave Page Cache (EPC). EPC is a set of memory that is reserved and protected by CPU, any programs that attempt to read the protected memory is blocked in hardware level. Currently the maximum amount of memory allocated to secure enclave is 128 MB, approximately 90 MB of which is available for programs that run in the enclave mode, and the rest is reserved by hardware. If the program consumes more than the amount of EPC RAM allocated, an EPC paging occurs resulting in a huge performance penalty. Depending on the use case, the performance could be several times slower as the enclave encrypts and evicts the previously used data into untrusted memory section to make room for new data.

In our proposed protocol, the secure enclave is the component that runs in the matching server where the integrity and confidentiality are guaranteed. Inside

the enclave there exists a key table with the main purpose of holding the keys of all users on the platform, as the enclave is the only trusted entity in our system. Although placing the key table inside the EPC memory grants security benefits, the limitation of EPC size eventually renders the system unusable once the registered users reach to a point where it exceeded the amount of EPC memory allocated, which triggers the EPC paging that mentioned above. To handle this, the key table needs to be stored in unprotected memory while ensuring that the privacy of users is not compromised.

A tree-based data structure that enables obliviousness without sacrificing too much performance was proposed in [23]. It was implemented using SGX in [16]. Using this data structure, the risk of exceeding the EPC memory is mitigated, with a small cost of reducing performance. In the experimental section, our protocols implements both data structures for the key table that exists in the enclave to evaluate the performance and space trade-off between two data structures.

We implement two different variants of our user matching protocols: the *Vanilla* version aims to provide maximum performance, whereas the *Extended* version takes advantage of various techniques to optimise the EPC memory usage in the enclave, as well as adding resistance to protect against some side-channel attacks, which Intel SGX is known to suffer for years. In the next sections, we discuss the overall EPC usage between $Vanilla$ and *Extended* versions, as well as the overall performance of running user matching with given parameters. Finally, we discuss the improvements of our proposed protocol and compare it with the previous work [12] that also uses SGX as a trusted computation environment but applied expensive homomorphic encryption.

6.1 Space Complexity

Intel SGX ensures that the content stored in the EPC memory is protected. This allows sensitive information such as symmetric keys to be stored within. However, simply storing the keys in the EPC memory is suboptimal as the current limit of the EPC memory is 128 MB, and only around 90 MB of which is available for developers, a paging issue that heavily impacts the performance occurs if the system consumes more than the designed EPC memory. The $Vanilla$ implementation stores the key table in the EPC memory, which gives great performance as the computational complexity of each operation such as search and add is always constant, with performance penalty once it exceeds the EPC memory. To facilitate this, the *Extended* version utilises Oblivious Binary Tree [16,23] that allows data to be stored securely in unprotected memory region in exchange for a small performance overhead.

In Fig. 2, the *Vanilla* implementation of our user matching protocols exceeds the maximum amount of EPC memory when the registered users reach 1 million, whereas the *Extended* implementation performs roughly 10 times better than *Vanilla* implementation, with around 9 MB of EPC memory usage even if the number of registered users reaches 2 million. The significance of the cost of EPC memory is obvious, that is, the *Extended* version certainly enables more users

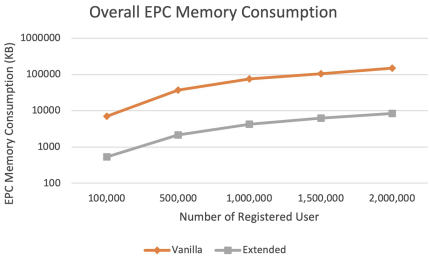

Fig. 2. EPC memory consumption.

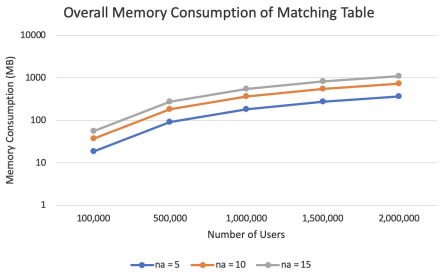

Fig. 3. MT memory consumption for various numbers na of attributes in the profiles.

to sign up on the platform before reaching the limit of one enclave. However, the *Extended* implementation also comes with performance penalty whenever a user requests for user matching, because of the overhead of utilising untrusted memory and switching between trusted and untrusted environment, which can also be addressed by integrating switchless calls [24].

Memory consumption of the matching table MT in our proposed protocols is highly dependent on the number of attributes in a profile. In our experiments we choose the configuration that each profile contains 5,10 and 15 attributes, which result in 5, 10 and 15 secret pairs CA, CB that is stored in the matching table MT. Given the fixed numbers of users, we measure the memory consumption as described in Fig. 3. A profile requires more space to store if the number of attributes increase accordingly, with more than 2 million registered users and more than 30 million secret pairs, it consumes roughly around 1000 MB of RAM allowing a fast lookup when a profile matching is requested.

Note that the space complexity of the matching table is approximate without counting the overhead of the data structure and its objects as they are not easily measurable. The space consumption of the actual encrypted profile is dependent on database implementation, thus it cannot provide a reference to how much disk space the encrypted profiles take.

6.2 Performance of User Matching Protocols

When a user U wishes to look for new friends, U begins by sending a request to the enclave that indicates the type of people U is looking for, and the server receives and process the request. The secure enclave first decrypts the request and compute corresponding search token for the matching server to locate the targeted users in the matching table. The experimental variables are as follows: Given a request which contains 5 attributes indicating the type of people U is looking for, the enclave computes the corresponding search tokens for those 5 attributes, and let the matching server search the matching table and return the top rst of results, where $rst = [25, 50, 100]$. In Fig. 4, the x-axis indicates the number of profiles returned, whereas the y-axis indicates the time spent

to complete each operation. We also generated three datasets of different sizes to assess how the performance changes when the number of registered users increases. These datasets contain 100k, 500k and 1m records, respectively.

As described in Fig. 4, user matching takes about 1 ms to complete on 3 different configurations for the *Vanilla* implementation, where the size of registered users varies, and 5 ms when the number of attributes goes up to 100. *Vanilla* implementations take advantage of hashing table which provides constant lookup time regardless of the size of the table, which greatly improves the performance of the matching, albeit with some compromises when it comes to leakage, which is discussed below. The increased time required for computing the secret corresponds to the amount of data needed to look up and compute, which believes to be reasonable and expected with the data structures involved.

The *Extended* implementation takes extra countermeasures to prevent potential data leaks via side-channel attacks, using oblivious data structure is the first counter-measurement that is used to resist cache-timing attacks, implementation of such data structure is presented in [16] with features that reduce the EPC usage by securely placing the data into untrusted regions after encryption. We measure the performance drop by up to 5 times compared to the *Vanilla* implementation that provides no protection against such attacks, from 5 ms to complete user matching with 25 returned users in the size of 100k users, all the way to 25 ms when the number of returned users is 100. This is due to the fact that oblivious data structure usually throttle the performance by a significant amount for security reason, and the most efficient oblivious data structure results in logarithmic complexity as supposed to constant found in *Vanilla* implementation. Context switching in the enclave also contributes to the longer execution time as the element in the key table needs to be encrypted before placing it outside the EPC memory section. In exchange for this degraded performance, we measure a significant drop in consumption of EPC memory in Fig. 2, allowing more users to sign up to the website while ensuring the enclave is resistant to some side-channel attacks.

Finally, we compare the performance of our new protocols with [12], another user matching scheme that utilises homomorphic encryption with multi-party settings that corporate with other servers to store and compute data for profile matching. The idea behind [12] is to split data into multiple servers to ensure the indistinguishability of data in the situation where some servers are compromised, while ensuring the data that stores across multiple servers can be securely computed using homomorphic encryption scheme, which imposes a huge computational overhead. Both [12,29] support matching profiles with numerical attributes only.

We measure the performance difference between our *Extended* implementation and [12] in Fig. 4 as an average in 30 tests, as [12] splits the secret into pieces, encrypts and distributes them to different servers using asymmetric encryption scheme, and combining them together when required by using the homomorphic property of the ElGamal encryption. When a user profile is requested by users, the user also submits their preferences to one of the servers, each server computes

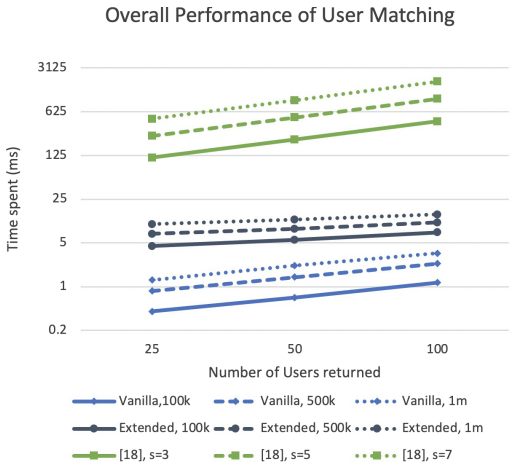

Fig. 4. Comparison of our protocol and alternatives using multiple servers and asymmetric encryption. The number of servers is denoted by s.

the partial similarity of the data and sends the intermediate result to the next server until all servers have completed computation, all intermediate results are then combined and revealed in secure hardware.

Finally, the users with the highest similarity score are returned to the requesting user. According to the experimental results in [12], it takes around 125 ms to compare 25 profiles, each profile with 5 attributes, on a configuration where the number of servers S is 3, and the number goes up to around 600 ms when S is 7. This computational overhead is mostly due to the complexity of computing exponentiation over large integers, and the experiments were conducted in a local network where the latency is minimised during the data transmission between multiple servers. Our new implementations outperform the solution [12] and in addition prevent several types of side-channel attacks.

7 Conclusion

In this paper, we design a privacy-protecting user matching protocol for modern social networks. It allows users to look for new friends without leaking any private information, while achieving reasonable performance so that it can be deployed in day-to-day use. Security analysis shows that the protocol is secure against various side-channel attacks and remains secure even when deployed in a public cloud. The security analysis and the outcomes of experiments presented in this paper demonstrate that our protocol is efficient, secure, practical and provides protection against side-channel attacks.

References

1. Agrawal, R., Evfimievski, A., Srikant, R.: Information sharing across private databases. In: Proceedings of 2003 ACM SIGMOD International Conference Management of Data, pp. 86–97. ACM (2003)
2. Ahmed, K.W., Al Aziz, M.M., Sadat, M.N., Alhadidi, D., Mohammed, N.: Nearest neighbour search over encrypted data using Intel SGXs. J. Inf. Secur. Appl. **54**, 102579 (2020)
3. Arnautov, S., et al.: SCONE: secure linux containers with Intel SGX. In: 12th USENIX Symposium on Operating Systems Design and Implementation, (OSDI 16), pp. 689–703 (2016)
4. Brasser, F., Müller, U., Dmitrienko, A., Kostiainen, K., Capkun, S., Sadeghi, A.R.: Software grand exposure: SGX cache attacks are practical. In: 11th USENIX Workshop on Offensive Technologies, WOOT 17 (2017)
5. Cash, D., et al.: Dynamic searchable encryption in very-large databases: data structures and implementation. In: NDSS, vol. 14, pp. 23–26. Citeseer (2014)
6. Duan, H., Wang, C., Yuan, X., Zhou, Y., Wang, Q., Ren, K.: Lightbox: full-stack protected stateful middlebox at lightning speed. In: Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security, pp. 2351–2367 (2019)
7. Freedman, M.J., Nissim, K., Pinkas, B.: Efficient private matching and set intersection. In: Cachin, C., Camenisch, J.L. (eds.) EUROCRYPT 2004. LNCS, vol. 3027, pp. 1–19. Springer, Heidelberg (2004). https://doi.org/10.1007/978-3-540-24676-3_1
8. Götzfried, J., Eckert, M., Schinzel, S., Müller, T.: Cache attacks on Intel SGX. In: Proceedings of the 10th European Workshop on Systems Security, pp. 1–6 (2017)
9. Jiang, Q., Qi, Y., Qi, S., Zhao, W., Lu, Y.: Pbsx: a practical private Boolean search using Intel SGX. Inf. Sci. **521**, 174–194 (2020)
10. Kissner, L., Song, D.: Privacy-preserving set operations. In: Shoup, V. (ed.) CRYPTO 2005. LNCS, vol. 3621, pp. 241–257. Springer, Heidelberg (2005). https://doi.org/10.1007/11535218_15
11. Li, H., Liu, D., Dai, Y., Luan, T.H.: Engineering searchable encryption of mobile cloud networks: when QoE meets QoP. IEEE Wirel. Commun. **22**(4), 74–80 (2015)
12. Luo, J., Yang, X., Yi, X.: SGX-based users matching with privacy protection. In: Proceedings of the Australasian Computer Science Week Multiconference, pp. 1–9 (2020)
13. Moghimi, A., Irazoqui, G., Eisenbarth, T.: CacheZoom: how SGX amplifies the power of cache attacks. In: Fischer, W., Homma, N. (eds.) CHES 2017. LNCS, vol. 10529, pp. 69–90. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-66787-4_4
14. Oleksenko, O., Trach, B., Krahn, R., Silberstein, M., Fetzer, C.: Varys: protecting SGX enclaves from practical side-channel attacks. In: 2018 USENIX Annual Technical Conference, USENIX ATC 18, pp. 227–240 (2018)
15. Priebe, C., Vaswani, K., Costa, M.: Enclavedb: a secure database using SGX. In: 2018 IEEE Symposium on Security and Privacy (SP), pp. 264–278. IEEE (2018)
16. Sasy, S., Gorbunov, S., Fletcher, C.W.: ZeroTrace: oblivious memory primitives from Intel SGX. IACR Cryptol. ePrint Arch. 2017, 549 (2017)
17. Schuster, F., et al.: Vc3: trustworthy data analytics in the cloud using SGX. In: 2015 IEEE Symposium on Security and Privacy, pp. 38–54. IEEE (2015)

18. Seo, J., et al.: SGX-shield: enabling address space layout randomization for SGX programs. In: NDSS (2017)
19. Sherry, J., Lan, C., Popa, R.A., Ratnasamy, S.: BlindBox: deep packet inspection over encrypted traffic. In: Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication, pp. 213–226 (2015)
20. Shi, E., Chan, T.-H.H., Stefanov, E., Li, M.: Oblivious ram with $O((\log N)^3)$ worst-case cost. In: Lee, D.H., Wang, X. (eds.) ASIACRYPT 2011. LNCS, vol. 7073, pp. 197–214. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-25385-0_11
21. Spreitzer, R., Plos, T.: Cache-access pattern attack on disaligned AES T-tables. In: Prouff, E. (ed.) COSADE 2013. LNCS, vol. 7864, pp. 200–214. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-40026-1_13
22. Stefanov, E., Papamanthou, C., Shi, E.: Practical dynamic searchable encryption with small leakage. In: NDSS, vol. 71, pp. 72–75 (2014)
23. Stefanov, E., et al.: Path ORAM: an extremely simple oblivious RAM protocol. In: Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security, pp. 299–310 (2013)
24. Tian, H., et al.: Switchless calls made practical in Intel SGXs. In: Proceedings of the 3rd Workshop on System Software for Trusted Execution, pp. 22–27 (2018)
25. Tsai, C.C., Porter, D.E., Vij, M.: Graphene-SGX: a practical library OS for unmodified applications on SGX. In: 2017 USENIX Annual Technical Conference, USENIX ATC 17, pp. 645–658 (2017)
26. Van Bulck, J., et al.: Foreshadow: extracting the keys to the Intel SGX kingdom with transient out-of-order execution. In: 27th USENIX Security Symposium, USENIX Security 18, pp. 991–1008 (2018)
27. Wang, W., et al.: Leaky cauldron on the dark land: understanding memory side-channel hazards in SGX. In: Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, pp. 2421–2434 (2017)
28. Yi, X., Bertino, E., Rao, F.Y., Bouguettaya, A.: Practical privacy-preserving user profile matching in social networks. In: 2016 IEEE 32nd International Conference on Data Engineering (ICDE), pp. 373–384. IEEE (2016)
29. Yi, X., Bertino, E., Rao, F.Y., Lam, K.Y., Nepal, S., Bouguettaya, A.: Privacy-preserving user profile matching in social networks. IEEE Trans. Knowl. Data Eng. **32**, 1572–1585 (2019)