Laud and Pankova *BMC Medical Genomics* 2018, **11**(Suppl 4):84
https://doi.org/10.1186/s12920-018-0400-8

BMC Medical Genomics

TECHNICAL ADVANCE Open Access

Privacy-preserving record linkage in large databases using secure multiparty computation

Peeter Laud[1*] and Alisa Pankova[1,2]

From iDASH Privacy and Security Workshop 2017
Orlando, FL, USA.14 October 2017

Abstract

Background: Practical applications for data analysis may require combining multiple databases belonging to different owners, such as health centers. The analysis should be performed without violating privacy of neither the centers themselves, nor the patients whose records these centers store. To avoid biased analysis results, it may be important to remove duplicate records among the centers, so that each patient's data would be taken into account only once. This task is very closely related to privacy-preserving record linkage.

Methods: This paper presents a solution to privacy-preserving deduplication among records of several databases using secure multiparty computation. It is build upon one of the fastest practical secure multiparty computation platforms, called Sharemind.

Results: The tests on ca 10 million records of simulated databases with 1000 health centers of 10000 records each show that the computation is feasible in practice. The expected running time of the experiment is ca. 30 min for computing servers connected over 100 Mbit/s WAN, the expected error of the results is 2^{-40}, and no errors have been detected for the particular test set that we used for our benchmarks.

Conclusions: The solution is ready for practical use. It has well-defined security properties, implied by the properties of Sharemind platform. The solution assumes that exact matching of records is required, and a possible future research would be extending it to approximate matching.

Keywords: Secure multiparty computation, Privacy-preserving record linkage, Deduplication, Privacy

Background

In this paper, we present a solution to the first track of iDASH 2017 competition [1], titled *De-duplication for Global Alliance for Genomics and Health (GA4GH)*. The goal is to develop privacy-preserving record linkage (PPRL) technique on top of existing European ENCCA Unified Patient Identifier (EUPID) framework to facilitate the deduplication task in GA4GH [2].

There is a large number of data providers, called *health-care centers*. They all have lists of patients. They want to know whether anyone else has the same patient in the list. This knowledge would allow them to exclude the duplicates from further data analysis, making it more efficient and less biased. Since each center uses its own local identities for its patients, they need to use so-called *quasi-identifiers* such as patient name and his date of birth to link the records, and this information may be sensitive. The centers could potentially agree on some kind of anonymization, replacing the quasi-identifiers with some random numbers. However, this is in general not sufficient, as e.g. seeing that some patient is present in exactly the centers A, B, C could already potentially leak more information about the patient. For example, if the observer already knew the patient records of A, he would in addition learn that these patients have also visited the

*Correspondence: peeter@cyber.ee
[1]Cybernetica AS, Ülikooli 2, 51003 Tartu, Estonia
Full list of author information is available at the end of the article

centers B and C. Mechanisms stronger than anonymization are needed to overcome this problem.

Technical details of the task

In the competition setting, the patient list of a health center is presented as a table, the columns of which are correspond to the patient's sensitive data, such as "first name", "last name", "date of birth", "sex", etc. Two records are considered the same if a certain combination of their attributes matches, e.g. concatenation of the name and the date of birth. For simplicity, it is assumed that the records are already pre-formatted, and possible typos should not be taken into account (we will discuss exact and approximate matching in the "Discussion" section). For each record, the centers compute the required combination of the attributes, and hash it. They then upload all hashes to the ideal computing machine, implemented with the help of Secure Multiparty Computation (SMC). The machine tells to each center, which of its records are also present in someone else's dataset, but it does not tell to whom exactly these records belong. The machine itself should be realized by $k \geq 3$ specially assigned servers, and no server should learn the values of uploaded hashes, since hashing is used only for shortness of representation and does not provide privacy by itself.

The targeted size of the computation is ca. 1000 centers, with ca. 10000 records each. The competition organizers have made available a test set with 1000 tables, each containing around 10000 records.

There are certain details about the data input and output, which make this task more specific:

- The data is not uploaded instantaneously by all centers. It is uploaded over time, so the ideal computing machine may have significant resources left over, while waiting for the next inputs.
- The first uploader of each record does not get it indicated as a duplicate, even if more copies of the same record are uploaded later. The record is indicated as a duplicate only to the next uploaders.

The system diagram is depicted in Fig. 1. There are n healthcare centers, indexed in the order they provide data. The data is uploaded to the servers in form of secret-shared hashes of records (1). After SMC servers compute the results on this data (2), they report to each center a list of booleans marking the duplicates (3), i.e. the records that have already occured in the centers with smaller indices. The results are returned to the centers in a secret-shared manner, so that no server will learn them, but the receiver center is able to reconstruct his answer from shares. In the end, each center knows which records it should discard to avoid repetitions with the other centers, and the remaining records may now be used for any other task, e.g. data analysis using other SMC protocols.

There can be two ways in which the final outputs should actually be delivered to the centers. These are different tasks with different challenges and solution possibilities. We will consider them both.

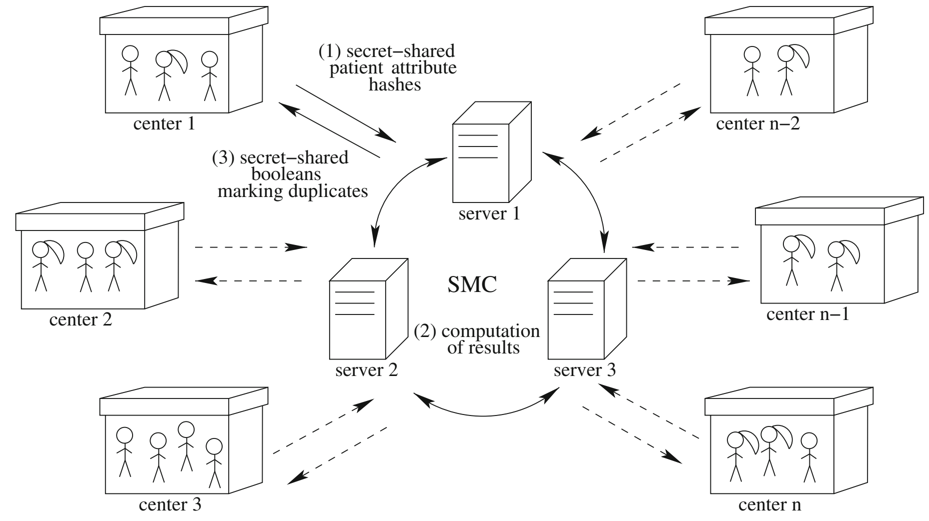
1. In our first solution, the results are sent out to the centers only after all uploads have finished. It is fine as far as all 1000 centers will agree to contribute their data more or less simultaneously, so that the first uploader will not have to wait for too long. We provide a faster and a more secure algorithm for this setting.
2. In our second solution, each upload of a bunch of records by a center is immediately followed by a response from the servers. This algorithm has to be used in the case when some centers may need more time to contribute their data, but the faster uploaders already want to get the results. It is more difficult to preserve privacy in this case.

Related work

There are numerous papers about PPRL, including (but not limited to) solutions based on Bloom filters for string approximate matching [3, 4], extensions of Bloom filters to matching of numerical data [5], as well as provably secure multiparty protocols [6]. Some practical implementations of PPRL include [7], describing a protocol where the linking is delegated to the patient himself, and [8], based on matching seeded hashes, which describes in details which preprocessing steps should be applied to the data to make it better linkable. An extensive state-of-the art overview of this topic in the beginning of 2017 has been given in [9].

Similarly to other privacy-preserving computation tasks, the two main approaches to PPRL are Secure Multiparty Computation (SMC) techniques and data perturbation techniques. The SMC approach is in general less efficient, but it provides clear security guarantees, while data perturbation methods can be severely broken due to informal security definitions that they provide [10]. For example, some cryptanalysis of Bloom filter based solutions has been performed in [11, 12]. They show some interesting attacks, including patient name frequency analysis. The iDASH 2017 competition has been focused on SMC methods.

SMC often uses the notion of three separate roles for parties [13]. There are *input parties* that provide the computation with data, which needs protection. There are *computation parties* that execute the privacy-preserving computation on provided data. The computation parties should not learn anything new from the execution, besides what is leaked through the design of the application. Finally, there are *result parties* who learn the results of the computation. These three sets of parties may intersect. In the context of the competition task, the computation



(1) secret−shared patient attribute hashes
(3) secret−shared booleans marking duplicates
center 1
center 2
center 3
center n−2
center n−1
center n
server 1
server 2
server 3
SMC
(2) computation of results

Fig. 1 System diagram. The setting to which the solutions proposed in this paper are applied

parties are specially assigned servers, whose number does not depend on the total number of health centers participating. The centers themselves are both input and result parties, each of them receiving a different subset of the outcomes.

In our solution, we assume three computation parties, since this is the setting for which the fastest SMC protocols are known, and mature computation platforms exist. The solution makes use of the following assumptions, allowed by the competition:

1. Honest majority: at most 1 of 3 servers can be corrupted.
2. Honest-but-curious (passive) adversaries: the corrupted server follows the protocol, but the attacker may get access to all messages received by this server.

As far as these assumptions are satisfied, the only things that may be inferred by the servers from the computation are the following. The first type of leakage is the "duplication pattern": *for each $n \in \mathbb{N}$, how many records are there, that occur in the lists of all data providers exactly n times?* In the context of our task, this leaks how many patients have been registered in exactly n hospitals, but this number is not correlated with particular records or hospitals in any way. The second type of leakage is: *how many duplicates have been reported to the health center?* It does not allow to correlate the duplicates of different centers in any way, and it only leaks that some n patients of this center have visited some other center. We note that, in our solutions, either the first or the second type of leakage takes place, but never both at once.

Some points for evaluating a PPRL algorithm have been mentioned in [9]. According to their framework, our solution could be assessed as follows:

- **Scalability:** The network communication complexity of the first algorithm is $O(n)$, where n is the total number of records that all the clients hold. e.g. if there are m clients holding k records each, then $n = mk$. The complexity of the second algorithm is formally $O(n^2)$, but we keep the constant small by using an elaborated filtering technique, which allow to reduce the number of comparisons up to 10 times compared to straightforward pairwise comparison, while slightly sacrificing the answer precision. We note that the running time of the second solution is actually more affected by heavy local computation than by network communication.
- **Linkage quality:** Our solution aims to achieve exact matching, as it was required by the test dataset. The loss in correctness is limited by 2^{-40} in our application. It happens mainly due to the initial application of a hash function to the inputs, which was a competition's requirement anyway, and also due to dropping some bits in order to match data types that Sharemind supports. In the second solution, some more correctness is sacrificed to gain better efficiency.
- **Privacy:** It is guaranteed that we leak no more than the total number of patients that have been registered in exactly n hospitals (the first solution), or the number of duplicated records in one client's data (the second solution).

Methods

The Sharemind platform

Our SMC protocols are implemented on top of the Sharemind® platform [14]. The platform provides a distributed virtual machine (VM) that must be installed at each of the computing servers. The machine interprets the description of a privacy-preserving application (in our case, the deduplication), where the cryptographic details are abstracted away. From the application developer's point of view, different pieces of data are merely labeled as "public" or "private". The private data will never be learned by any of the servers, unless it is deliberately *declassified* (i.e. opened to all servers) by the application. While the public values are stored at each server in plain, the private values are stored in a secret-shared manner, preventing a single server from learning its value. Underneath, the virtual machine executes cryptographic protocols to perform operations on private secret-shared data, which in general requires interaction between the servers. The underlying cryptographic protocols have been proven to be composable [15], meaning that the applications do not need to undergo any additional security proofs. Only deliberate declassification of values needs to be justified. This also concerns the deduplication that we present in this paper.

The main protocol set of Sharemind [16], denoted shared3p, is based on additive sharing among three parties. The private representation of a value $u \in R$ from a finite *ring* R is denoted by $[\![u]\!]$, and is defined as $[\![u]\!] = ([\![u]\!]_1, [\![u]\!]_2, [\![u]\!]_3)$, where the *share* $[\![u]\!]_i \in R$ is held by the i-th server. The shares are random elements of R, with the constraint that the sum up to u. Hence, if anyone manages to get only one or two shares of u, he cannot infer any information about u. In [16], the authors have presented protocols for a number of basic arithmetic, relational and logical operations, such as addition, multiplication, comparisons, etc., transforming shares of the inputs into shares of the output. The supported rings are $\mathbb{Z}_{2^n}$ and $\mathbb{Z}_2^n$, and several different rings may be in used simultaneously in the same application. There are special protocols for converting shares between different rings. This basic set of operations has been extended in numerous follow-up papers [17–22].

The protocols of shared3p are secure against one passively corrupted party. Our deduplication application is built on top of them. In the rest of this section, we present all algorithms that we used in our application. In the loops, we use **foreach** and **while** to denote parallelizable loops, whilst **for** loops are sequential. Parallelization is important due to the latency of all non-linear operations with private values, due to the need to exchange messages between computation servers. We denote vectors as $\vec{x} = \langle x_1, \ldots, x_n \rangle$.

The first solution: output all results once in the end

In out first solution, the results of computation are only available at the end, after all *clients* (the health centers) have uploaded the hashes of their records. The outline of the process is the following. In the first phase, the SMC servers are collecting input data from the clients, without performing any deduplication detection on it. When the uploads have ceased (e.g. the deadline or some other trigger event has happened), the servers stop collecting the data. They run the deduplication algorithm on all data they managed to collect so far, and give to each client its personal output. If a record is duplicated, then the client that first uploaded it will not be notified that it is a duplicate. All other clients that have uploaded the same record will receive a notification about it.

Cryptographic building blocks

We fix two cryptographic functions: the hash function $H : \{0,1\}^* \rightarrow \{0,1\}^\eta$, and a block cipher $E : \{0,1\}^\eta \times \mathcal{K} \rightarrow \{0,1\}^\eta$, where $\mathcal{K}$ is the set of possible keys for E. The block size of E should be the same as the output length of H, so that we could apply E directly to the output of H. The challenge is that we will apply E to secret-shared values, and hence the block cipher E has to be easily computable under SMC. We have picked AES-128 as E, and the privacy-preserving implementation of AES-128 is already available in Sharemind [19]. There exist newer, possibly more efficient implementations of AES [23], as well as proposals for SMC-optimized block ciphers [24], which have not been implemented on Sharemind yet, and could potentially speed the computation up.

In our solution, we have taken $\eta = 128$. We let H be the composition of SHA-256 cryptographic hash function and a universal one-way hash function family (UOWHF) [25].

Computation on the client side

The behaviour of a client is described in Algorithm 1. At start-up, each client queries the Sharemind servers for the random parameters of the UOWHF H, which the servers generate in the beginning. Each client takes the

Algorithm 1: Client computation

```
Data: Patient records $p_1, \ldots, p_n$
Result: Booleans $b_1, \ldots, b_n$, where $b_i = 1$ iff $p_i$ is a
        duplicate
begin client($p_1, \ldots, p_n$
    Receive parameters $a, r$ from the servers.
    $\eta \leftarrow 128$
    foreach $i = 0$ to $n - 1$ do
        Let $s_i$ be concatenation of key attributes of $p_i$.
        $x \leftarrow$ SHA256($s_i$)
        $h_i \leftarrow (a \cdot x + r) \bmod 2^\eta$
        Send $h_i$ to the servers.
    end
    foreach $i = 0$ to $n - 1$ do
        Receive a boolean value $b_i$ from the servers.
        If $b_i = 1$, then the $i$-th record is a duplicate.
    end
    return $\langle b_1, \ldots, b_n \rangle$
end
```

input records as soon as they become available, picks the necessary attributes from each record, applies H to them, and uploads the hashes to Sharemind servers in secret-shared manner. The Sharemind API readily supports that. At this point, no server learns the exact values of the hashes, since they are all secret-shared.

In the end, when the servers have finished the computation, the client queries them for its personal result. The servers respond with the shares of the output, which is a vector of booleans of the same length as the number of records from this client, indicating whether a record is a duplicate one. The client reconstructs the result vector. Again, the Sharemind API already has support for that.

Computation on the server side

We describe the work of the servers in phases.

Start-up. This short phase is given in Algorithm 2. The servers privately generate the following random values:

- parameters of the hash function H;
- a key $[\![K]\!]$ for the block cipher E.

All these values are generated in such a way that they remain secret-shared among the servers, and no server actually learns them. Sharemind API supports such shared randomness generation, and we denote the corresponding functionality as random.

The servers initialize a public variable $cnt \leftarrow 0$. It will be used for indexation of clients.

Algorithm 2: Start-up

```
begin serverStartup ()
    $[\![K]\!] \leftarrow \mathsf{random}(2^{128})$
    $a \leftarrow \mathsf{declassify}(\mathsf{random}(2^{128}))$
    $r \leftarrow \mathsf{declassify}(\mathsf{random}(2^{128}))$
    $cnt \leftarrow 0$
    while more clients are connecting do
        Send $a, r$ to the client.
    end
end
```

Upload During the upload phase, two different activity threads are essentially taking place in parallel.

One thread is the actual acceptance of data from the clients, described in Algorithm 3. The hashes of clients are stored into an private array $[\![v]\!]$, and the corresponding client identities into a public array s under the same indices. In principle, this algorithm could be invoked several times for the same client, if it intends to split up the upload of its data. Several clients may want to connect to the servers at the same time, so we need to make use of Sharemind's database support to avoid race condition on cnt.

Algorithm 3: Uploading data to servers

```
begin serverReceiveUploads ()
    while more data coming do
        Let the client upload the hash $[\![h]\!]$ of a record
        Let $id$ be the identity of the client
        $[\![v_{cnt}]\!] \leftarrow [\![h]\!]$
        $s_{cnt} \leftarrow id$
        $cnt \leftarrow cnt + 1$
    end
end
```

The other thread of activity is encrypting the elements $[\![v_i]\!]$. For each i, the servers evaluate E on $[\![v_i]\!]$ in a privacy-preserving way, using the same key $[\![K]\!]$ for each encryption, and take the first 64 bits of the result, as described in Algorithm 4. The only reason why we take only 64 of 128 bits is that the largest ring that Sharemind currently supports is $\mathbb{Z}_{2^{64}}$. There are no problems with taking only 64 bits, as long as there are no collisions. With the envisioned amounts of data (around $10^7 = 2^{23.5}$ records in total), collisions are unlikely; the birthday paradox puts their probability at around $2^{-64+2\cdot 23.5} = 2^{-17}$. We note that the communication complexity also reduces with the number of bits of the data types.

Algorithm 4: Computing encryptions of hashes

```
Data: Private values $[\![v_0]\!], \ldots, [\![v_{cnt-1}]\!]$
Result: Private values $[\![c_0]\!], \ldots, [\![c_{cnt-1}]\!]$, where $c_i$ is an
        encryption of $v_i$ for all $i \in \{0, \ldots, cnt-1\}$
begin encrypt($[\![v_0]\!], \ldots, [\![v_{cnt-1}]\!]$)
    foreach $i = 0$ to $cnt - 1$ do
        $[\![e_i]\!] \leftarrow E([\![K]\!], [\![v_i]\!])$
        $[\![c_i]\!] \leftarrow \mathsf{leftbits}_{64}([\![e_i]\!])$
    end
    return $\langle [\![c_0]\!], \ldots, [\![c_{cnt}]\!] \rangle$
end
```

In our implementation, evaluation of E can take place either immediately after the data upload, or it can be run in a separate thread, waiting until more data comes to process it all in parallel. In any case, the application of E is done in parallelized manner, batching up the hashes that are waiting for being processed so far.

Computation of results. When the upload has finished and the ciphertext $[\![c_i]\!]$ has been computed for each $i \in \{0, \ldots, cnt-1\}$, then the final results are computed as in Algorithm 5. First of all, all computed ciphertexts are privately shuffled, and then declassified. Since they are encrypted with a block cipher whose key remains secret-shared, it only reveals the "duplication pattern", i.e. how many values occur there exactly n times. The private shuffle [26] in the shared3p protocol set is highly efficient: the amount of communication is linear in the size of the shuffled vector, and the number of communication rounds is constant. It also allows to easily apply the shuffle inverse, which is as efficient as the shuffle itself.

Algorithm 5: Finding the duplicates

```
Data: Private encryptions $[\![c_0]\!], \ldots, [\![c_{cnt-1}]\!]$
Data: Public client identities $s_0, \ldots, s_{cnt-1}$, where the
      record $c_i$ belongs to $s_i$ for all $i \in \{0, \ldots, cnt-1\}$.
begin serverMain($[\![c_0]\!], \ldots, [\![c_{cnt-1}]\!], s_0, \ldots, s_{cnt-1}$)
    Generate a random permutation $[\![\sigma]\!]$ for $cnt$
    elements.
    $[\![\vec{w}]\!] \leftarrow$ shuffle($[\![\sigma]\!], [\![\vec{c}]\!]$)
    $[\![\vec{t}]\!] \leftarrow$ shuffle($[\![\sigma]\!], \langle 0, 1, \ldots, cnt-1 \rangle$)
    $\vec{w} \leftarrow$ declassify($[\![\vec{w}]\!]$)
    foreach $i \in \{0, \ldots, cnt-1\}$, such that $w_i$ unique in
    $\vec{w}$ do
        $r_i \leftarrow false$
    end
    foreach value $ct$ occurring in $\vec{w}$ several times do
        $\{i_1, \ldots, i_k\} \leftarrow$ positions, where $ct$ occurs in $\vec{w}$
        $\langle r_{i_1}, \ldots, r_{i_k} \rangle \leftarrow$ findmin($[\![t_{i_1}]\!], \ldots, [\![t_{i_k}]\!]$) //Alg. 6
    end
    $[\![\vec{b}]\!] \leftarrow$ inverse_shuffle($[\![\sigma]\!], \vec{r}$)
    for $i = 0$ to $cnt-1$ do
        output $[\![b_i]\!]$ to the data provider $s_i$
    end
end
```

At this point, the servers could already mark the duplicates with boolean values 1 and the remaining values with 0. After the obtained vector is privately shuffled back, the servers can return the secret shared bits to the clients, so that the i-th clients learns the bits indicating the duplicity of its own records. The problem is that the servers need to notify all clients except the first one, but because of shuffling they do not know which entry belongs to the first client. They cannot declassify the client indices $[\![s_i]\!]$ either since it would partially undo the shuffling.

In order to determine which value belongs to the first client, the servers run Algorithm 6. This algorithm is applied to each set of shuffled entries that have identical values, to determine the minimum amongst them. The minimum should be labeled *false* since it is not considered a duplicate, and all other elements should be labeled *true*. The idea behind this recursive algorithm is the following. The inputs are split into pairs, and a comparison performed within each pair. The element that is larger is definitely not the minimum of the entire set, so we can immediately write $b_i := true$. The indices of all elements that turned out to be smaller are stored into m_i, and the whole algorithm is now applied again to the elements indexed by m_i. The procedure is repeated until there are is only one element left, which is the minimum, so the algorithm returns *false* in one-element case.

Although Algorithm 6 works with private values, it reveals the ordering between certain elements of $\vec{t}$. Because of the random shuffle, the ordering of this vector is completely random, so it does not disclose any information. No server learns from the output more than it would from a random permutation.

Algorithm 6: Finding the minimum of private values with public ordering

```
Data: Private numbers $[\![x_1]\!], \ldots, [\![x_k]\!]$, all different
Result: Public booleans $b_1, \ldots, b_k$ where $b_i = false$ iff
        $x_i$ is the smallest among $x_1, \ldots, x_k$
begin findmin($[\![x_1]\!], \ldots, [\![x_k]\!]$)
    if $k = 1$ then return $\langle false \rangle$ $\ell \leftarrow \lfloor k/2 \rfloor$
    foreach $i \in \{1, \ldots, \ell\}$ do
        if declassify($[\![x_i]\!] < [\![x_{i+\ell}]\!]$) then
            $m_i \leftarrow i$
            $b_{i+\ell} \leftarrow true$
        else
            $m_i \leftarrow i + \ell$
            $b_i \leftarrow true$
        end
    end
    if $k$ is odd then
        $\ell \leftarrow \ell + 1$
        $m_\ell \leftarrow k$
    end
    $\langle b_{m_1}, \ldots, b_{m_\ell} \rangle \leftarrow$ findmin($[\![x_{m_1}]\!], \ldots, [\![x_{m_\ell}]\!]$)
    //Alg. 6 return $\langle b_1, \ldots, b_k \rangle$
end
```

The second solution: output results immediately

Our second solution considers the case where the client is immediately notified, which of its hashes of records have already been uploaded by some previous client. This task is more difficult than the previous one. We cannot simply declassify the encrypted hashes when they arrive, since it would leak which pairs of centers have intersecting sets of records.

The second solution has the same components E and H as the first solution, constructed in the same manner. Again, the computation servers agree on the key $[\![K]\!]$ in the beginning, as well as the parameters of H.

Computation on the client side

The clients perform pretty much in the same manner as in the first solution, according to Algorithm 1. The only technical difference is that the servers react to the upload immediately, so the client most likely does not interrupt the session with the servers, and stays connected until it receives the result.

Computation on the server side

Assume that the servers already store T encrypted hashes (zero or more), provided by the previous clients. These T hashes have been stored as *public* values $z_1, \ldots, z_T$, where z_i is equal to the first 64 bits of $E(K, v_i)$. Duplicates have already been removed among $z_1, \ldots, z_T$. Assuming that duplicates never end up among $z_1, \ldots, z_T$, since $[\![K]\!]$ remains private, $z_1, \ldots, z_T$ are computationally indistinguishable from T uniformly randomly distributed values, so it is safe to make them public.

The 64-bit values $z_1, \ldots, z_T$ are kept in 2^B *buckets*, numbered from 0 to $2^B - 1$. In our implementation, $B = 16$. Each bucket is just a vector of values. Each z_i is assigned to the bucket $\mathbf{B}_j$, where j is equal to the first 16 bits of z_i.

Now suppose that a center has uploaded the hashes $[\![v_1]\!], \ldots, [\![v_t]\!]$. The servers need to check whether these hashes has occurred before. The computation starts by encrypting each $[\![v_i]\!]$. Let $[\![z'_1]\!], \ldots, [\![z'_t]\!]$ be the results of encryption, computed by $[\![z'_i]\!] = \mathsf{leftbits}_{64}(E([\![K]\!], [\![v_i]\!]))$. We cannot immediately declassify them since it would leak which pairs of centers have intersecting sets of records. Instead, we should use privacy-preserving comparison.

We do not want to simply invoke the private comparison protocol for each z_i and $[\![z'_j]\!]$, because we consider their number to be too large. Indeed, as we are aiming to handle ca. 10 million records, this method would cause us to compare each pair of records, leading to ca. $5 \cdot 10^{13}$ invocations of the comparison protocol. An ℓ-bit comparison requires slightly more network communication than a ℓ-bit multiplication [16], with the latter requiring each computation server to send and receive two ℓ-bit values [17]. If $\ell = 64$ and there are $5 \cdot 10^{13}$ operations, then each server has to send out and receive at least $6 \cdot 10^{15}$ bits, which on a 100 Mb/s network (specified in the conditions of the competition task) would take almost two years.

We reduce the number of comparisons in the following manner. Let $[\![z']\!]$ be one of the private values $[\![z'_1]\!], \ldots, [\![z'_t]\!]$; all t values are handled in parallel. The comparison of $[\![z']\!]$ with $z_1, \ldots, z_T$ is described in Algorithm 7. In this algorithm, we let N be the maximum size of a bucket. We denote the j-th element of the i-th bucket by $\mathbf{B}_{i,j}$. We assume that each bucket has exactly N elements, adding special dummy elements if necessary.

The *characteristic vector* of an element $x \in \mathbb{Z}_M$ is a vector $\langle b_0, \ldots, b_{M-1}\rangle \in \mathbb{Z}_2^M$, where $b_x = 1$ and all other elements are equal to 0. The shared3p protocol set has a simple and efficient protocol for computing characteristic vectors, described in [27]. The protocol turns a private value into a private characteristic vector. The characteristic vector of $\mathsf{leftbits}_{16}([\![z']\!])$ marks the index of the bucket to which $[\![z']\!]$ belongs, and the expression $\bigoplus_{i=0}^{2^B-1}[\![b_i]\!] \cdot \mathbf{B}_{i,j}$ returns exactly the j-th element of that bucket, which we denote $[\![y_j]\!]$. This way, the values $[\![y_1]\!], \ldots, [\![y_N]\!]$ are the privately represented content of the bucket, into which z' would belong. The private comparison $[\![z']\!] \overset{?}{=} [\![y_j]\!]$ is performed for all j, thus comparing $[\![z']\!]$ against each element that belongs to the i-th bucket. Finally, $[\![b]\!] = \bigvee_{j=1}^{n}[\![c_j]\!]$ is the private OR of all comparisons, which tells whether there had been at least one match. The private bits $[\![b_1]\!], \ldots, [\![b_t]\!]$ resulting from applying Algorithm 7 to all $[\![z'_1]\!], \ldots, [\![z'_t]\!]$ are returned to the client.

Algorithm 7: Comparing $[\![z']\!]$ against $z_1, \ldots, z_T$

```
Data: Public numbers $z_1, \ldots, z_T$, all different
Data: A private number $[\![z']\!]$
Result: Private boolean $b$ where $b = true$ iff $z' = z_j$ for
        some $j \in \{1, \ldots, T\}$
begin compare($[\![z']\!], z_1, \ldots, z_T$)
    Let $z_i$ be partitioned to $2^B$ buckets $\mathbf{B}_j$ by first $B$ bits.
    Compute the characteristic vector
    $\langle [\![b_0]\!], \ldots, [\![b_{2^B-1}]\!] \rangle$ of $\mathsf{leftbits_B}([\![z']\!])$.
    foreach $j \in \{1, \ldots, N\}$ do
        $[\![y_j]\!] \leftarrow \bigoplus_{i=0}^{2^B-1}[\![b_i]\!] \cdot \mathbf{B}_{i,j}$
        $[\![c_j]\!] \leftarrow [\![z']\!] \overset{?}{=} [\![y_j]\!]$
    end
    return $[\![b]\!] = \bigvee_{j=1}^{n}[\![c_j]\!]$
end
```

After returning the answer to the client, the buckets have to be updated with $z'_1, \ldots, z'_t$. It is safe to declassify $[\![z'_i]\!]$ if $b_i = 0$, since in this case $[\![z'_i]\!]$ cannot be correlated to any of z_i and is indistinguishable from a random value. However, we cannot immediately declassify the vector $[\![\vec{b}]\!]$, because the positions of duplicated elements may give away information about the input data. Since we are allowed to leak the total number of duplicated entries per client, we can do as described in Algorithm 8:

1. randomly shuffle $\langle [\![b_1]\!], \ldots, [\![b_t]\!] \rangle$ and $\langle [\![z'_1]\!], \ldots, [\![z'_t]\!] \rangle$, using the same permutation;
2. declassify $[\![\vec{b}]\!]$;
3. declassify those $[\![z'_i]\!]$, where $b_i = 0$, and add these z'_i to the respective buckets.

In our implementation, $[\![z']\!]$ is shared over $\mathbb{Z}_2^{64}$, hence taking the first 16 bits of it is a local operation, resulting in a value in $\mathbb{Z}_2^{16}$. The characteristic vector protocol in [27] is easily adaptable to compute the characteristic vectors of elements of $\mathbb{Z}_2^B$, and its result is a vector over $\mathbb{Z}_2$ with

```
Algorithm 8: Updating $z_1,\ldots,z_T$ with $z'_1,\ldots,z'_t$
  Data: Buckets $\mathbf{B}_j$ of public numbers $z_1,\ldots,z_T$, all
        different
  Data: Private numbers $[\![z'_1]\!],\ldots,[\![z'_t]\!]$
  Data: Private duplicity booleans $[\![b_1]\!],\ldots,[\![b_t]\!]$
  Result: Buckets $\mathbf{B}_j$ updated with those $z'_i$, where $b'_i$ is
        false
  begin update($\mathbf{B}_1,\ldots,\mathbf{B}_B,[\![z'_1]\!],\ldots,[\![z'_t]\!]$)
      Generate a random permutation $[\![\sigma]\!]$ for $t$ elements
      $[\![\vec{b}]\!] \leftarrow$ shuffle($[\![\sigma]\!],[\![\vec{b}]\!]$)
      $[\![\vec{z}']\!] \leftarrow$ shuffle($[\![\sigma]\!],[\![\vec{z}']\!]$)
      $\vec{b} \leftarrow$ declassify($[\![\vec{b}]\!]$)
      foreach $i \in \{1,\ldots,t\}$ where $b_i = 0$ do
          $\vec{z}_i \leftarrow$ declassify($[\![z_i]\!]$)
      end
      Add declassified $z_i$ to the respective buckets $\mathbf{B}_j$.
      return $\mathbf{B}_1,\ldots,\mathbf{B}_B$
  end
```

length 2^B. The computation of the characteristic vector requires communication of two elements of $\mathbb{Z}_2^B$ and one element of $\mathbb{Z}_2^{2^B}$ (in *total*, not per party).

The computation of $[\![y_j]\!]$ in Algorithm 7 is again a local operation. The computations of $[\![c_j]\!]$ and their disjunction are straightforward using the protocols available in the shared3p set of Sharemind.

Speeding up local computations

In Alg. 7, the computation of $[\![y_j]\!]$ is local. Nevertheless, in practice it takes a major part of the entire effort of the protocol, taking up most of the time in it. If we knew something about the size of z', we could compute $\bigoplus$ not over all buckets, but only over a subset of them, covering only the range into which z' is guaranteed to fall, with a negligible error.

A bucket is defined by the most significant bits of elements that it contains. If we take any two buckets, then all elements in one of them will be strictly smaller than all elements in the other one. If we sort $[\![z'_1]\!],\ldots,[\![z'_t]\!]$ in ascending order (Sharemind has efficient protocols for sorting private values [28]), we know that the first elements more likely belong to the buckets with "smaller" bits, and the last elements more likely belong to buckets with "larger" bits. We can estimate these probabilities more precisely.

As the key $[\![K]\!]$ is secret, and the hashes $[\![v_1]\!],\ldots,[\![v_t]\!]$ are all different, the values $[\![z'_1]\!],\ldots,[\![z'_t]\!]$ can be treated as mutually independent, uniformly random elements of $\mathbb{Z}_2^{64}$. After sorting them, their likely ranges can be derived from the order statistics as follows.

Let $\mathcal{P}$ be a discrete probability distribution over values $x_1, x_2, \ldots$, such that the probability mass of x_i is p_i. Let $X_1,\ldots,X_n$ be random variables sampled from $\mathcal{P}$, and let $X'_1,\ldots,X'_n$ be obtained after sorting $X_1,\ldots,X_n$ in ascending order. We have $\Pr[X'_j \le x_i] = \sum_{k=j}^{n}\binom{n}{k}P_i^k\cdot(1-P_i)^{n-k}$, where $P_i = Pr[X_i \le x_i] = \sum_{k=1}^{i} p_k$. This quantity comes from summing up probabilities of all possible combinations, where at least j of n variables are smaller than x_i. For a fixed j, this expression is actually the cumulative density function of a binomial distribution $B(n, P_j)$.

In our case, $\mathcal{P}$ is the distribution over AES ciphertexts, i.e $p_i = 2^{-128}$ for all i. The sorted ciphertexts z'_i are instances of random variables X'_i. We want to find m_i and M_i, such that $\Pr[z'_i < m_i] \le \varepsilon$ and $\Pr[z'_i > M_i] \le \varepsilon$, where ε is the desired error probability. Since we are dealing with binomial distribution, we can use e.g. Höffding's inequality

$$\Pr[z'_i \le m_i] \le exp\left(-2\frac{(n\cdot P_i - m_i)^2}{n}\right)$$

and Chernoff's inequality

$$\Pr[z'_i \le m_i] \le exp\left(-\frac{1}{2P_i}\cdot\frac{(n\cdot P_i - m_i)^2}{n}\right) \ ,$$

where $exp(x) = e^x$ for Euler's number e. We can solve the equation $\epsilon = exp(-2\frac{(n\cdot P_i - m_i)^2}{n})$ if $P_i \le \frac{1}{4}$, and $\epsilon = exp(-\frac{1}{2P_i}\frac{(n\cdot P_i - m_i)^2}{n})$ if $P_i \ge \frac{1}{4}$, getting the value for m_i. The value for M_i can be obtained analogously, since $\Pr[z'_i \ge x_i] = \Pr[-z'_i \le -x_i]$.

By default, we use $\varepsilon = 2^{-40}$ as the probability of error. As the total number of hashes is expected to be around $1000\cdot 10000 \approx 2^{23.5}$ and we have two bounds to try for each hash, the probability of making a bounds check error during the whole run is not more than $2\cdot 2^{-40+23.5} = 2^{-15.5}$, which we consider acceptable, and which is also similar to errors due to the collisions in the first 64 bits of AES output.

The usefulness of these bounds increases together with t. If $t = 100$ (and $\varepsilon = 2^{-40}$), then we gain little, as the ranges $[m_i, M_i]$ still cover around half of the whole range. If $t = 10000$, then the sorted values can be much more tightly positioned — the ranges $[m_i, M_i]$ are less than 1/10 of the whole range.

Table 1 First solution

Encrypting the hashes	29 m
Shuffling the encrypted hashes	60 s
Find the first element in each duplicated set	30 s
Unshuffling the boolean results	12 s
Public sort, housekeeping	18 s
Total	32 m

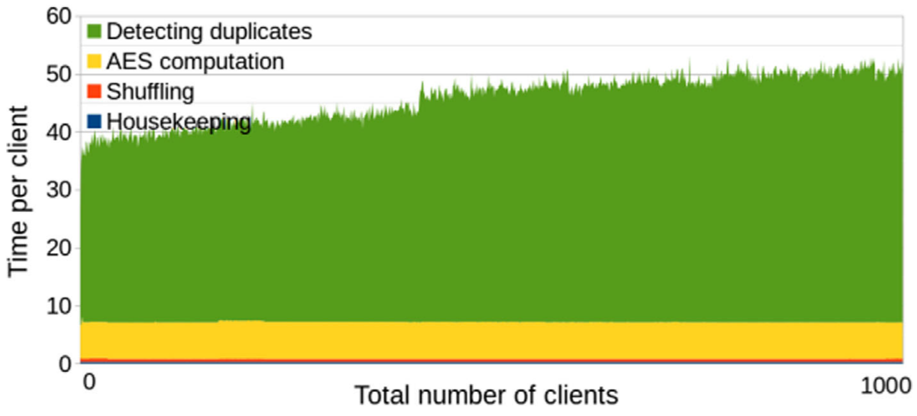


Fig. 2 Second solution (w/o polynomials). Efficiency graph of the second solution without polynomial optimization. The number of CPU threads is limited to 2

Alternative comparison

The communication costs of Algorithm 7 may be further reduced, ultimately turning them into a constant (assuming that B is constant), albeit with a further increase in the costs of local computation.

Consider the bucket $\mathbf{B}_i$ with elements $\mathbf{B}_{i,1},\ldots,\mathbf{B}_{i,N}$. The value z' is an element $\mathbf{B}_i$ iff it is a root of the polynomial $\mathbf{P}_i(x) = \prod_{j=1}^{N}(x - \mathbf{B}_{i,j})$. The polynomial is considered over the field $\mathbb{F}_{2^{64}}$. The elements of this field are 64-bit strings and their addition is bitwise exclusive or. Hence, an additive sharing over $\mathbb{F}_{2^{64}}$ is at the same time also sharing over $\mathbb{Z}_2^{64}$ and vice versa.

Let $\mathbf{P}_{i,0},\ldots,\mathbf{P}_{i,N}$ be the coefficients of the polynomial $\mathbf{P}_i$. It does not make sense to compute $\mathbf{P}_i([\![z']\!])$ in a straightforward way, because this would involve $N-1$ private multiplications for each bucket. A better way is given in Algorithm 9.

```
Algorithm 9: Comparing $[\![z']\!]$ against $z_1,\ldots,z_T$ by polynomial evaluation
Data: Public numbers $z_1,\ldots,z_T$, all different
Data: A private number $[\![z']\!]$
Data: Coefficients $\mathbf{P}_{i,0},\ldots,\mathbf{P}_{i,N}$ of the polynomial
      $\mathbf{P}_i(x) = \prod_{j=1}^{N}(x - \mathbf{B}_{i,j})$
Result: Private boolean $[\![b]\!]$ where $b = true$ iff $z' = z_j$
        for some $j \in \{1,\ldots,T\}$
begin compare_poly($[\![z']\!], z_1,\ldots,z_T$)
    Privately compute powers of $[\![z']\!]$: $[\![z'^2]\!],\ldots,[\![z'^N]\!]$
    foreach $i \in \{0,\ldots,2^B-1\}$ do
        $[\![d_i]\!] \leftarrow$
        $\langle[\![1]\!],[\![z']\!],[\![z'^2]\!],\ldots,[\![z'^N]\!]\rangle \cdot \langle\mathbf{P}_{i,0},\ldots,\mathbf{P}_{i,N}\rangle$
        $[\![c'_i]\!] \leftarrow [\![d_i]\!] \overset{?}{=} 0$
    end
    Return $[\![b]\!] = \bigvee_{i=0}^{2^B-1}[\![c'_i]\!]$
end
```

In Algorithm 9, $[\![d_i]\!]$ is computed as a scalar product of the private vector of the powers of z' with the public vector of the coefficients of $\mathbf{P}_i$. The powers of $[\![z']\!]$ have to be computed only once for all buckets. These powers are computed with the help of the usual multiplication protocol of Sharemind, but working over $\mathbb{F}_{2^{64}}$. The local operations in this protocol are multiplications in $\mathbb{F}_{2^{64}}$, which are relatively more expensive than ordinary bitwise operations. In our implementation we use the NTL library [29] for binary field operations. The computation of all $[\![d_i]\!]$ involves many multiplications in this field, so even though the computation does not require any communication between the parties, it is quite heavy on the local side.

As the computation of $[\![z'^2]\!],\ldots,[\![z'^N]\!]$ is done over a field, we can push its heaviest part to the *precomputation phase*, leaving just a single private multiplication to be done during runtime. The method is described in ([30], Algorithm 1). The precomputation consists of generating a random invertible element $[\![r]\!] \in \mathbb{F}^*_{2^{64}}$ together with its inverse $[\![r^{-1}]\!]$ and computing $[\![r^2]\!],\ldots,[\![r^N]\!]$. During the runtime, one computes $[\![z']\!]\cdot[\![r^{-1}]\!]$ and declassifies it. For an exponent k, the private value $[\![z'^k]\!]$ is then found as $[\![z'^k]\!] = \left(z'\cdot r^{-1}\right)^k \cdot [\![r^k]\!]$, which can be computed locally, without interaction.

Table 2 Second solution (w/o polynomials)

Housekeeping (avg per client)	0.28 s
Encrypting the hashes (avg per client)	6.3s
Shuffling the encrypted hashes (avg per client)	0.51 s
Detecting duplicates (min of 1000 clients)	20s
Detecting duplicates (max of 1000 clients)	46s
Total (1000 clients)	12 h 45 m

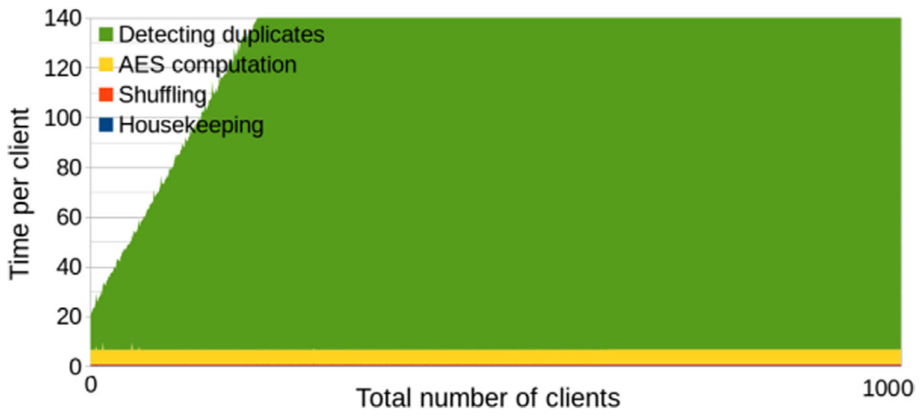


Fig. 3 Second solution (w/ polynomials). Efficiency graph of the second solution with polynomial optimization. The number of CPU threads is limited to 2

In binary fields, including $\mathbb{F}_{2^{64}}$, squaring of additively shared values does not require communication between the servers. This can be used to speed up precomputations. In ([30], Algorithm 4) it is shown how to reduce the *communication* cost of computing $[\![r^2]\!], \ldots, [\![r^N]\!]$ to that of approximately $\sqrt{N}$ multiplications.

The polynomial-based comparison method is also amenable to the order statistic related speedup described in the previous section. Both comparison methods have been implemented in our second solution.

Results

We have evaluated our solution on the test data provided by iDASH 2017 competition organizers. The data consists of 1000 CSV formatted files, ca. 10000 rows each. The evaluation has been performed on a local cluster of three 2 x Intel Xeon E5-2640 v3 2.6 GHz/8GT/20M servers, where one server has also played the role of all clients. The network bandwidth has been throttled to 100 Mb/s, and latency changed to 40 ms. The number of computation threads was limited to 2, as required by the competition rules. The data upload times have not been measured since they depend on the network connection between the server and the centers, and are not related to the particular SMC algorithms.

The number of incorrect answers in our benchmarks was 0, and it turns out that the error 2^{-40} was fine for 1000 clients with 10000 records each.

First solution

The benchmarks of our first solution are given in Table 1. The entire computation takes ca. 32 min, after which all centers receive back the list of entries that are duplicates. This is quite fast for privacy-preserving computation with such large inputs, and in practice setting up the servers and collecting the data may even take more time than the computation itself.

Second solution

If the centers do not provide their data simultaneously, and the time span between the first and the last contribution is long, then the first uploaders may probably not want to wait for the remaining ones. The 32 min of the actual computation may become insignificant compared to the time waiting for the other centers, so even a slower algorithm may give advantage if it returns the results immediately. The benchmarks of our second solution (*without* polynomial optimization) are given in Fig. 2 and summarized in Table 2, and they show that the waiting time for each server is between 20 and 45 s. The later the center joins, the longer will be his computation time. For all 1000 centers altogether, the second solution would give 12 h 45 min compared to the 32 min of the first solution. However, if the time span between the first and the last contributions exceeds at least 12 h 15 min, then the second method already gives advantage also in the total time.

The benchmarks of the second solution *with* polynomial optimization are given in Fig. 3 (the limit on Y-axis values is set to 140 to make it comparable to the other graphs) and summarized in Table 3. We see that, although the communication between parties decreases, the computation is slower. The time cost per client visibly grows linearly with the number of clients, so the total time is quadratic in the number of clients. This

Table 3 Second solution (w/ polynomials)

Housekeeping (avg per client)	0.28 s
Encrypting the hashes (avg per client)	5.8 s
Shuffling the encrypted hashes (avg per client)	0.53 s
Detecting duplicates (min of 1000 clients)	14 s
Detecting duplicates (max of 1000 clients)	666 s
Total (1000 clients)	88 h 8 m

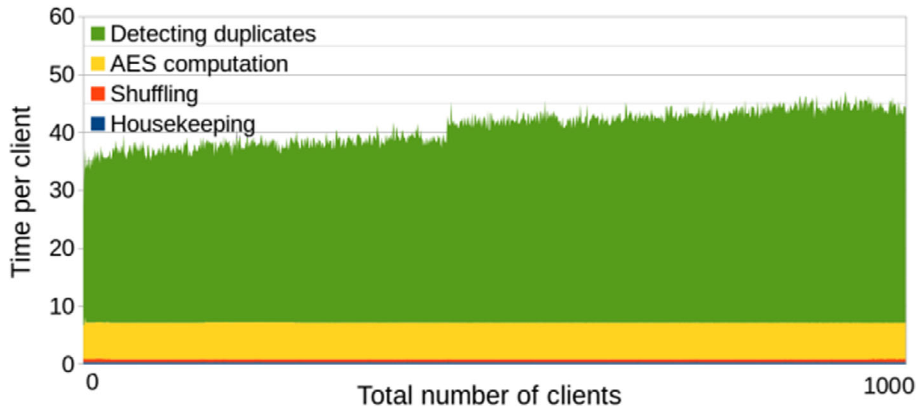


Fig. 4 Second solution (w/o polynomials, no thread limit). Efficiency graph of the second solution without polynomial optimization. The number of CPU threads is 32

means that the multiplication in $\mathbb{F}_{2^{64}}$ is too heavy, and even limiting network connection to 100Mbps and adding 40ms latency did not make its complexity less significant. The total time of processing 1000 clients was almost seven times slower than the solution without polynomials. Nevertheless, for small number of clients, this solution is actually more efficient, and total time is smaller for the first 55 clients. The time *per client* has started getting worse after the 30-th client, so it is reasonable to switch between comparison algorithms at this point. Since the alternative comparison has less network communication, it should be theoretically better in settings with very slow network connection and fast hardware. We decided to conduct one more experiment to verify this claim.

During the competition, the solutions were evaluated using only 2 parallel processor threads, but we have also tried to run the same algorithm without limiting their number (using all 32 cores), to see if we get any advantage. The results of the benchmarks with and without polynomials can be found in Figs. 4 and 5, summarized in Tables 4 and 5 respectively. We see that the polynomial solution was still slower, although all 32 processors on all three servers worked with 100% load during the main computation. However, while the time improvement of the solution without polynomials is not too significant (although 1.5 extra hours still make a difference), we see that the performance of polynomial solution is much better than it was with 2 threads, and the 1000 clients have been processed in 19 h 30 m instead of 88 h 8 m. This time, the polynomial solution was more efficient that the non-polynomial solution with up to 300 clients. The time *per client* has started getting worse after the 160-th client. We see that the alternative comparison indeed scales better w.r.t. parallelization.

Discussion

We think that the results are good enough and can be applied to real data. Depending on the requirements, either the first or the second solution can be used.

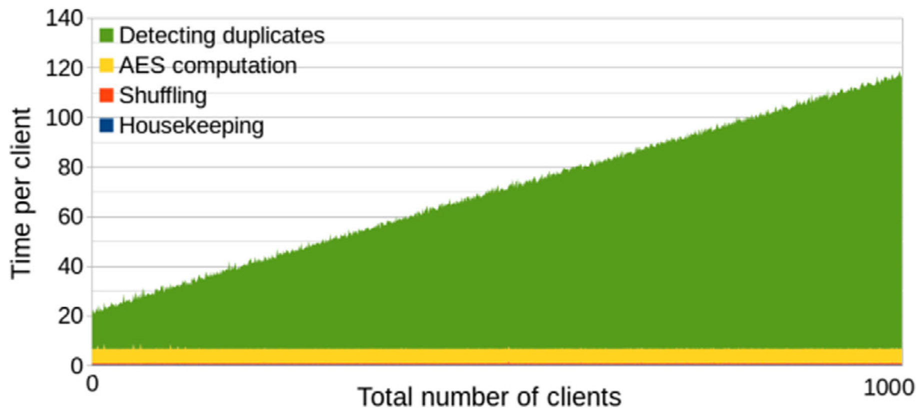


Fig. 5 Second solution (w/ polynomials, no thread limit). Efficiency graph of the second solution with polynomial optimization. The number of CPU threads is 32

Table 4 Second solution (w/o polynomials, no thread limit)

Housekeeping (avg per client)	0.28 s
Encrypting the hashes (avg per client)	6.3 s
Shuffling the encrypted hashes (avg per client)	0.50 s
Detecting duplicates (min of 1000 clients)	20 s
Detecting duplicates (max of 1000 clients)	40 s
Total (1000 clients)	11 h 20 m

The data that was provided by iDASH 2017 competition organizers was idealized, i.e. it was already preprocessed and assumed that e.g. the gender is labeled exactly "M" and "F" and there are no other possible encodings. In a practical application, we should think how much preprocessing we agree to delegate to the data owner, and how much we perform using secure multiparty computation.

We note that revealing "duplication pattern" is safe as far as there are sufficiently many clients, so that leaking how many patients have been present in exactly n centers would be safe. This leakage would be bad in some extreme cases, e.g with two medical centers with one patient, since then the server will know whether that particular patient has been in both centers or only one of them. However, if the data is small, then slower and more secure algorithms could be used for that anyway, such as finding privacy-preserving set intersection between each pair of medical centers to determine the duplicates.

One possible problem of Sharemind deployment is that its protocols work under assumptions of the honest majority and an honest-but-curious adversary. This means that one needs to find three hosts who agree to set up an SMC server, and who are trustworthy enough not to deliberately break the computation, and clean the server from all intermediate data after the computation has finished (some challenges of real-world application deployment are described e.g. in [31]). Sacrificing in efficiency, we could use protocol sets like SPDZ [32] or MASCOT [33], which are actively secure against a dishonest majority. In any case, as far as we use secret sharing, we need to assume that at least one server should remain uncorrupted, or otherwise the computation parties will collaborate to reconstruct the clients' secrets, regardless of the privacy-preserving algorithms used.

Table 5 Second solution (w/ polynomials, no thread limit)

Housekeeping (avg per client)	0.28 s
Encrypting the hashes (avg per client)	5.8 s
Shuffling the encrypted hashes (avg per client)	0.55 s
Detecting duplicates (min of 1000 clients)	14 s
Detecting duplicates (max of 1000 clients)	112 s
Total (1000 clients)	19 h 30 m

Possible future work

As discussed e.g. in [9], the quasi-identifiers are often inconsistent in real applications, so exact comparison of values is not sufficient to achieve accurate linkage results, and some approximated string matching can be preferred. Our algorithms cannot be extended to this setting directly, since AES encryption maps similar strings to completely different values. We could take some ideas from some previous works, and hash all n-grams of a string instead of the entire string. Different n-grams should be encrypted with different AES keys to avoid leaking similarities in the string patterns. Nevertheless, this would already leak more information, and a corrupted server would be able to perform e.g. analysis of bigram frequencies, and there could possibly be more attacks inspired by [12], even though the attacks are adapted to Bloom filters that we do not use in our solution. We conclude that the extension to approximate matching would require a more careful design and security analysis.

Conclusion

In this paper, we have proposed two methods for finding duplicates between several databases. The implementation has been benchmarked on data of 1000 medical centers with 10000 records each, giving quite positive results, which would be acceptable in real-world scenarios.

Abbreviations

AES: Advanced encryption standard; CSV: Comma separated values; PPRL: Privacy-preserving record linkage; SMC: Secure multiparty computation; SHA: Secure hash algorithm; UOWHF: Universal one-way hash function; VM: Virtual machine; WAN: Wide area network

Acknowledgements

The authors want to thank Marju Ignatjeva who helped to set up the Sharemind virtual machine for iDASH competition.

Funding

The research and the publication cost were funded by Estonian Research Council, grant no. IUT27-1.

Availability of data and materials

The test data itself is not available anymore on the competition site. The virtual machine of the implementation is not available due to Sharemind group's commercial interests.

About this Supplement

This article has been published as part of?BMC Medical Genomics?Volume 11 Supplement 4, 2018: Proceedings of the 6th iDASH Privacy and Security Workshop 2017. The full contents of the supplement are available online at?https://bmcmedgenomics.biomedcentral.com/articles/supplements/volume-11-supplement-4.

Authors' contributions

PL has developed the theory part and the corresponding algorithms. AP has implemented these algorithms on Sharemind platform, and performed the experiments. Both authors have read and approved the final manuscripct.

Ethics approval and consent to participate

Not relevant.

Consent for publication

Not relevant

Competing interests
The authors declare that they have no competing interests.

Publisher's Note

Author details
[1]Cybernetica AS, Ülikooli 2, 51003 Tartu, Estonia. [2]STACC, Ülikooli 2, 51003 Tartu, Estonia.

Published: 11 October 2018

References

1. iDASH 2017 competition . http://www.humangenomeprivacy.org/2017/. Accessed 7 July 2018.
2. Ebner H, Hayn D, Falgenhauer M, Nitzlnader M, Schleiermacher G, Haupt R, Erminio G, Defferrari R, Mazzocco K, Kohler J, Tonini G, Ladenstein R, Schreier G. Piloting the european unified patient identity management (eupid) concept to facilitate secondary use of neuroblastoma data from clinical trials and biobanking, vol. 223. In: Health Informatics Meets eHealth: Predictive Modeling in Healthcare - From Prediction to Prevention, Proceedings of the 10th eHealth 2016 Conference, Studies in Health Technology and Informatics. Vienna: IOS Press; 2016. p. 31–38. https://doi.org/10.3233/978-1-61499-645-3-31.
3. Schnell R, Bachteler T, Reiher J. Privacy-preserving record linkage using bloom filters. BMC Med Inform Decis Making. 2009;9(1):41. https://doi.org/10.1186/1472-6947-9-41.
4. Karapiperis D, Verykios VS. A fast and efficient hamming lsh-based scheme for accurate linkage. Knowl Inf Syst. 2016;49(3):861–84. https://doi.org/10.1007/s10115-016-0919-y.
5. Vatsalan D, Christen P. Privacy-preserving matching of similar patients. J Biomed Inform. 2016;59:285–98. https://doi.org/10.1016/j.jbi.2015.12.004.
6. Du W, Atallah MJ. Protocols for secure remote database access with approximate matching, vol. 2. In: Ghosh AK, editor. E-Commerce Security and Privacy. Advances in Information Security. Boston: Springer; 2001. p. 87–111. https://doi.org/10.1007/978-1-4615-1467-1.
7. Alhaqbani B, Fidge C. Privacy-preserving electronic health record linkage using pseudonym identifiers. In: HealthCom 2008 - 10th International Conference on E-health Networking, Applications and Services. Singapore: IEEE; 2008. p. 108–17. https://doi.org/10.1109/HEALTH.2008.4600120.
8. Kho AN, Cashy JP, Jackson KL, Pah AR, Goel S, Boehnke J, Humphries JE, Kominers SD, Hota BN, Sims SA, Malin B, French DD, Walunas TL, Meltzer DO, Kaleba EO, Jones RC, Galanter WL. Design and implementation of a privacy preserving electronic health record linkage tool in chicago. J Am Med Inform Assoc. 2015;22(5):1072–80. https://doi.org/10.1093/jamia/ocv038.
9. Vatsalan D, Sehili Z, Christen P, Rahm E. Privacy-preserving record linkage for big data: Current approaches and research challenges. In: Zomaya AY, Sakr S, editors. Handbook of Big Data Technologies. Cham: Springer; 2017. p. 851–95. https://doi.org/10.1007/978-3-319-49340-4.
10. Kargupta H, Datta S, Wang Q, Sivakumar K. Random-data perturbation techniques and privacy-preserving data mining. Knowl Inf Syst. 2005;7(4): 387–414. https://doi.org/10.1007/s10115-004-0173-6.
11. Kuzu M, Kantarcioglu M, Durham EA, Toth C, Malin B. A practical approach to achieve private medical record linkage in light of public resources. J Am Med Inform Assoc. 2013;20(2):285–92. https://doi.org/10.1136/amiajnl-2012-000917.
12. Niedermeyer F, Steinmetzer S, Kroll M, Schnell R. Cryptanalysis of Basic Bloom Filters Used for Privacy Preserving Record Linkage. J Priv Confidentiality. 2014;6(2):59–79.
13. Kamm L. Privacy-preserving statistical analysis using secure multi-party computation. 2015. PhD thesis, University of Tartu.
14. Sharmeind secure multiparty computation platform. https://sharemind.cyber.ee. Accessed 7 July 2018.
15. Bogdanov D, Laud P, Laur S, Pullonen P. From Input Private to Universally Composable Secure Multi-party Computation Primitives. In: Datta A, Fournet C, editors. IEEE 27th Computer Security Foundations Symposium, CSF 2014, Vienna, Austria, 19-22 July, 2014. Computer Society Press: IEEE; 2014. p. 184–98. https://doi.org/10.1109/CSF.2014.21.
16. Bogdanov D, Niitsoo M, Toft T, Willemson J. High-performance secure multi-party computation for data mining applications. Int J Inf Sec. 2012;11(6):403–18.
17. Randmets J. Programming languages for secure multi-party computation application development. 2017. PhD thesis, University of Tartu. http://hdl.handle.net/10062/56298.
18. Dimitrov V, Kerik L, Krips T, Randmets J, Willemson J. Alternative implementations of secure real numbers. In: Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security. CCS '16. New York: ACM; 2016. p. 553–64. https://doi.org/10.1145/2976749.2978348.
19. Laur S, Talviste R, Willemson J. From Oblivious AES to Efficient and Secure Database Join in the Multiparty Setting, vol. 7954. In: Applied Cryptography and Network Security. LNCS. Berlin: Springer; 2013. p. 84–101.
20. Krips T, Willemson J. Hybrid model of fixed and floating point numbers in secure multiparty computations, vol. 8783. In: Proceedings of the 17th International Information Security Conference, ISC 2014. LNCS. Cham: Springer; 2014. p. 179–97.
21. Kamm L, Willemson J. Secure floating-point arithmetic and private satellite collision analysis. Int J Inf Secur. 2015;14:531–48.
22. Krips T, Willemson J. Point-counting method for embarrassingly parallel evaluation in secure computation, vol. 9482. In: Garcia-Alfaro J, Kranakis E, Bonfante G, editors. Foundations and Practice of Security: 8th International Symposium, FPS 2015, Clermont-Ferrand, France, October 26-28, 2015, Revised Selected Papers. LNCS. Cham: Springer; 2016. p. 66–82.
23. Araki T, Furukawa J, Lindell Y, Nof A, Ohara K. High-throughput semi-honest secure three-party computation with an honest majority. In: Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security. CCS '16. New York: ACM; 2016. p. 805–17. https://doi.org/10.1145/2976749.2978331.
24. Albrecht MR, Grassi L, Rechberger C, Roy A, Tiessen T. Mimc: Efficient encryption and cryptographic hashing with minimal multiplicative complexity, vol. 10031. In: Cheon JH, Takagi T, editors. Advances in Cryptology - ASIACRYPT 2016 - 22nd International Conference on the Theory and Application of Cryptology and Information Security, Hanoi, Vietnam, December 4-8, 2016, Proceedings, Part I. Lecture Notes in Computer Science; 2016. p. 191–219. https://doi.org/10.1007/978-3-662-53887-6.
25. Naor M, Yung M. Universal one-way hash functions and their cryptographic applications. In: Johnson DS, editor. Proceedings of the 21st Annual ACM Symposium on Theory of Computing, May 14-17, 1989, Seattle, Washigton, USA. New York: ACM; 1989. p. 33–43. https://doi.org/10.1145/73007.73011.
26. Laur S, Willemson J, Zhang B. Round-Efficient Oblivious Database Manipulation. In: Proceedings of the 14th International Conference on Information Security. ISC'11, vol. 7001. Lecture Notes in Computer Science. Xi'an: Springer; 2011. p. 262–77.
27. Laud P, Randmets J. A domain-specific language for low-level secure multiparty computation protocols. In: Ray I, Li N, Kruegel C, editors. Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security, Denver, CO, USA, October 12-6, 2015. New York: ACM; 2015. p. 1492–503. https://doi.org/10.1145/2810103.2813664.
28. Bogdanov D, Laur S, Talviste R. A practical analysis of oblivious sorting algorithms for secure multi-party computation, vol. 8788. In: Bernsmed K, Fischer-Hübner S, editors. Secure IT Systems - 19th Nordic Conference, NordSec 2014, Tromsø, Norway, October 15-17, 2014, Proceedings. Lecture Notes in Computer Science. Cham: Springer; 2014. p. 59–74. https://doi.org/10.1007/978-3-319-11599-3.
29. NTL. A Library for doing Number Theory. http://www.shoup.net/ntl/. Accessed 7 July 2018.
30. Laud P. A private lookup protocol with low online complexity for secure multiparty computation, vol. 8958. In: Hui LCK, Qing SH, Shi E, Yiu SM, editors. Information and Communications Security - 16th International Conference, ICICS 2014, Revised Selected Papers. Lecture Notes in Computer Science. Cham: Springer; 2014. p. 143–57. https://doi.org/10.1007/978-3-319-21966-0.
31. Bogdanov D, Kamm L, Kubo B, Rebane R, Sokk V, Talviste R. Students and taxes: a privacy-preserving study using secure computation. PoPETs. 2016;2016(3):117–35.

32. Damgård I, Pastro V, Smart NP, Zakarias S. Multiparty computation from somewhat homomorphic encryption, vol. 7417. In: Safavi-Naini R, Canetti R, editors. CRYPTO. Lecture Notes in Computer Science. Berlin: Springer; 2012. p. 643–62.
33. Keller M, Orsini E, Scholl P. Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, October 24-28, 2016. In: Weippl ER, Katzenbeisser S, Kruegel C, Myers AC, Halevi S, editors. New York: ACM; 2016. p. 830–42. https://doi.org/10.1145/2976749.2978357.