Configurable Private Querying: Lookup and Partial Matching under Homomorphic Encryption

Hamish Hunt
IBM Research, Hursley, UK
hamishun@uk.ibm.com

Jack Crawford
IBM Research, Hursley, UK
jack.crawford@ibm.com

Oliver Masters*
Squarepoint, London, UK
oliver.masters@squarepoint-capital.com

Enrico Steffinlongo
IBM Research, Hursley, UK
enrico.steffinlongo@ibm.com

Flavio Bergamaschi
IBM Research, Hursley, UK
flavio@uk.ibm.com

ABSTRACT

The ability to query a database privately is nowadays ubiquitous via an encrypted channel. With the advent of homomorphic encryption, there is a want to expand the notion of privacy in this context to querying privately on the database with the database learning as little to no information of the query data or its result. The ability to compute the intersection from at least two parties' sets that are kept private only to themselves is known as private set intersection (PSI) and should be considered a fundamental operation in several homomorphic computation scenarios to do useful work; not least for the ability to implement queries on a database. We outline in this paper a novel highly configurable PSI structure to be used in private querying providing the possibility that even the exact query itself can be protected from the database if required. As well as complex database lookups, there is also a more complex partial matching. The outline of the system design is discussed and we report preliminary results on some of the fundamental operations. We demonstrate that this technology is emerging as a viable given response to lookup queries and partially matching on an encrypted database with over a million entries in approximately 9 minutes.

KEYWORDS

homomorphic encryption, homomorphic computation, private set intersection, private information retrieval, partial matching, relational databases

1 INTRODUCTION

The storage of data has revolutionized human endeavors to an extent that the average person is now aware that large amounts of data are held in many modern systems in a subcomponent known as the database.

In the early days of databases, there were fundamental questions such as how a database should be structured and how should querying such a database work. Arguably the most successful solution was to model the database on the relations data had with each other [10]. Queries on a database then became the well understood operations on the relations themselves.

In the same era as early relational database research and development, modern encryption was also making breakthroughs for secure communication [12]. Soon thereafter, the use of encryption to protect channels of communication with databases to perform queries in private is now commonplace. The question now is how little information can a database (compromised or not) learn from a query request? Maybe computation directly on the encrypted data without the requirement to decrypt could answer this question.

The ability to perform computation on encrypted data directly is possible due to homomorphic encryption (HE) schemes that have been developed in the last few decades. There are various *partially* HE schemes which allow a single homomorphic operation, such as Paillier [28] and ElGamal [14] permitting addition and multiplication respectively. Despite these schemes being known for a few decades, it was not until Gentry's breakthrough [16] that proved *fully* HE schemes to be possible. These schemes enable the computation of both operations to an arbitrary circuit depth. However, Gentry's *fully* HE scheme proved to be impractical, thus leading to the introduction of numerous, more practical *somewhat* HE schemes supporting both operations to a fixed circuit depth coming to the fore. The most commonly used *state-of-the-art* schemes are BGV [2] and FV [15] schemes, which are suitable for work in the integers and the CKKS [6] scheme that is used to represent complex approximate numbers.

Several mature HE software libraries are now available [7, 21, 25, 29, 31] for developing algorithms under homomorphic encryption, more commonly referred to as *homomorphic* algorithms.

HE creates challenges for software development as it can be viewed as a new paradigm to programming such as not allowing early termination of all or part of an algorithm due to the data being encrypted. Thus, we must address how to achieve query operations homomorphically through Private Set Intersection (PSI). The basic concept of PSI can be viewed as two parties in possession of private datasets desiring to compute the set intersection. However, this computation must only reveal the members of the intersecting set and no additional information.

This work explores and presents a novel approach of utilizing HE to achieve PSI in a configurable manner. Most recent literature on homomorphic PSI has concentrated on optimizations to speed up the inherent computational complexity of HE. This paper proposes a method of homomorphic PSI that leads to *high configurability* and we show how it can be used to configure queries in Private Information Retrieval (PIR) and optionally without the server knowing the exact query being run. This furthers the goal towards a general and flexible HE database. The second use demonstrated is a partial matching algorithm that returns weighted scores based on

*Contributions by Oliver Masters to this work were made while he was an IBM Research UK employee.

user-defined inputs that can be configured. This type of partial matching has many applications such as, but not limited to, a scenario where a law enforcement agency is searching a database for a car, someone who is looking to buy an item, or identity verification for know your customer requirements. All these uses are where the client does not have exact information on what they are searching for and/or relative *importances* of the data have to be taken into account.

2 BACKGROUND

2.1 Private Set Intersection

Private set intersection (PSI) is a secure multiparty computation involving at least two parties. Each party holds a private set of data and wishes to compute the intersection between their set and the sets of all other parties involved. This must be achieved without revealing any information other than the elements that are contained in the intersecting set. One of the first secure PSI protocols was proposed by Meadows [26]. The concept utilized the homomorphic multiplicative property of the Diffie-Hellman key exchange and although having a reasonable communication cost, the running times did not scale well with the set sizes.

The last ten years has seen numerous applications that show PSI to be practical with two protocols in particular [24, 30] being considered to be the most efficient. These two protocols are based on *Oblivious Transfer* (OT) where the client sends a single OT request for each element in their set and learns a randomized encoding for each request obliviously; that is, without revealing the request or the client learning more than their request should provide. Thus, each OT is a comparison protocol performed in parallel that does not reveal any information outside of the intersection.

Notably, Chen *et al.* [4, 5] later took a similar approach to Pinkas *et al.* [30] but made use of fully homomorphic encryption. These schemes will be described in more detail below. Chen *et al.* improved on the efficiency of previous PSI schemes. However, each of these protocols include an emphasis on achieving the best possible efficiency whereas our contribution is a protocol that provides a high level of configurability to the queries that the client is permitted to send.

Chen *et al.* [5] proposed a basic PSI protocol utilizing fully homomorphic encryption with particular focus on optimizations that reduce both the communication cost as well as the circuit depth. This is achieved through several methods including the use of cuckoo hashing to reduce the size of the elements to be computed on. Additionally, they introduce the use of *partitioning* and *windowing* to reduce the computation circuit depth, albeit as a trade-off with communication cost. Computing the intersection of 5,000 32-bit strings and 16,000,000 32-bit strings, Chen *et al.* claimed a total communication cost of 12.5 MB. Furthermore, they achieved running times of 36 seconds for server-side computation and 71 seconds for client-side pre-processing using a protocol specifically optimized to provide the lowest running times. They used several techniques. *Windowing* attempts to reduce the number of multiplications needed to compute the difference of the query element and the entire set of elements of the server. Essentially, the client pre-computes the necessary powers of the query; the queries are raised to several powers, how many are traded off with other factors that impact resources. Then the client sends all of these values to the server encrypted. Clearly this causes the user to decide upon the trade-off between decreasing the circuit depth and increasing the communication cost. *Partitioning* attempts to reduce the circuit depth by separating the elements of the sets into smaller subsets and computing the same PSI protocol on these smaller subsets. However, this introduces a trade-off between the computation cost and the communication cost, as for every subset, the server receives one query element and sends back a reply.

There have been numerous works on using hashing to perform efficient PSI-based protocols such as the work from iDASH 2015 [22, 23]. In order to produce a secure comparison between genomic data via calculation of either the Hamming distance or edit distance, the teams used hashing to reduce the size of the data.

In this work, instead of attempting to achieve optimal performance of a somewhat limited query, we concentrate on providing flexibility of the queries that the client can make. Thus, the specific optimizations proposed by Chen *et al.* [5] are not considered suited to our purposes here.

2.2 Private Information Retrieval

Private information retrieval (PIR) was a notion first introduced by Chor *et al.* [9]. The concept is where a client uses a unique identifier to query a server to retrieve an item of data, typically contained within a database, without revealing the intent of the query.

There has been considerable work on achieving PIR through the use of HE in recent years [13, 27] where a *keyword* is provided for searching. This method of PIR using HE is clearly an extension of PSI as shown by Chen *et al.* [4] who built upon their previous PSI protocol [5] and extended it to perform *labeled PSI* which is synonymous with batched, single-server PIR by keywords. In this scenario, the server holds a single label per item in their set and the client receives the labels corresponding to the items that appear in the intersecting set. Both their works concentrate on the case of *unbalanced PSI* which is when there is a considerable difference between the sizes of the two sets being computed on. For sets of size 2^{20} and 2^9, a recorded running time was given of 1 second and a total communication cost of 4 MB using a single thread.

Typically, efficient PIR requires unique identifiers. Most notably, the work by Akavia *et al.* [1] proposed a method of achieving PIR with non-unique identifiers as well as a procedure that could be used for extracting each individual match from the database in the order that they occur.

At the time of writing, there has been limited research on the area of fuzzy or partial matching, especially in the HE domain. There has been work on fuzzy matching protocols [8, 32] using the additively homomorphic Paillier scheme. This was later shown to be possible using a fully HE scheme [3]. This protocol performed fuzzy database queries where the matches were not exact. It involved vector comparisons via the computation of the Euclidean distance. Vectors with shorter Euclidean distances were considered to be closer matches. A threshold function can be computed homomorphically through the evaluation of a polynomial. This is achieved with a mapping represented by a polynomial function which is evaluated on a ciphertext having the effect of the mapping applied on the data in the slots. If the query vector had a Euclidean distance within the threshold then it was considered to match the entry in the database and this entry would be returned to the client without revealing the intent of the query.

Our work furthers this concept of fuzzy matching by producing a weighted score of a partial match relative to a query with user-defined weights to provide a measure on importance.

2.3 BGV scheme

For our purposes, we make use of the BGV HE scheme [2] given by HElib [21]. Although, it is possible to use other alternative implementations of the BGV scheme or even the BFV scheme given its similar plaintext space for the same purposes [7, 25, 29]. In the BGV case, the plaintext space is $A = \mathbb{Z}_p[x]/\Phi_m(x)$ where p is a prime number and $\Phi_m(x)$ is the m^{th} cyclotomic polynomial for some natural number m. In this paper, we make particular use of the isomorphism $A \cong E^l$ for some l, where we have $E = \mathbb{Z}_p[x]/I(x)$ where $I(x)$ is some irreducible factor of $\Phi_m(x)$ in $\mathbb{Z}_p[x]$. Throughout this work, the non-negative form of integers modulo p is used, i.e. we take representatives from the set $\{0, 1, \cdots, p-1\}$, however other representations are possible.

For a more concrete explanation of E, let d be the order of p in the group of units $U(\mathbb{Z}_m)$, the group of elements in $\mathbb{Z}_m$ which have multiplicative inverses. Elements of E are polynomials of degree at most $d-1$ with modulo-p integer coefficients since $\deg(I(x)) = d$. Moreover, from these facts, we can compute $l = \phi(m)/d$.

The ciphertext space $S = \mathbb{Z}_q[x]/\Phi_m(x)$ is similar to the plaintext space where instead we have q as a large composite number and m is the same as that of the plaintext space. Typically $q \gg p$ and the natural embedding $A \longrightarrow S$ is frequently employed. There are numerous publications describing the implementation of the BGV scheme in more detail [17, 18] as well as algorithms available in HElib [19, 20].

3 IMPLEMENTATION

3.1 Basic Scenario

The implementation assumes a scenario where a *client* will encode and encrypt query data to be sent to another party, the *server*. The *server* will compute on the encrypted query data sent against an encrypted database and produce a result. In the scenario of our implementation and experimentation, the result will either be an encrypted *score* or encrypted *mask*. The configuration, which query to perform or partial matching score to provide, in this scenario is known by both parties; although it could be optionally encrypted or not encrypted. We also discuss below in further work (section 7.3) the scenario where the database is not encrypted.

3.2 Encoding

We describe how to encode the relevant data into the plaintext space A so that they can be encrypted and the necessary operations can be performed. We consider a database given by N entries and M columns. The nomenclature used here is for the collection of data rather than sets (or in reality multi-sets) for the uses of PIR and partial matching or scoring usages that we discuss in this paper. Each entry will have column values in the database that will be typically a *small* datum such as a fixed-length integer.

Using the approach in this paper, we are able to embed in a column $l = \phi(m)/d$ data entries into one plaintext of our space. For simplicity, we first describe the case where $l \geq N$, so that an entire column of our database can be mapped to one element of A. Thus, with M plaintext elements of A, we are able to encode the entire database. The database is then encrypted under the BGV scheme into M ciphertexts $\{D_i\}_{i=0}^{M-1}$.

Next, to encode a query which is a set of M integers, encoded into a set of M plaintexts $\{P_i\}_{i=0}^{M-1}$. Here each P_i is an N-fold repetition of the query datum corresponding to column i. These plaintexts are then encrypted under the same public key as the database into the query ciphertexts, $\{Q_i\}_{i=0}^{M-1}$. This means that if we have multiple queries then this process is repeated for each query producing M ciphertexts per query data. However, it is possible to pack multiple queries across the slots of the M ciphertexts; this requires the database to be in the plaintext space and is described in more detail and left as further work in section 7.3.

We have described the basic encoding scheme for the simple case where the entire column fits into one element of A. However, it would be more typical to have $N > l$ and therefore we need to break the database up into disjoint ciphertexts which will be rows in the database D. We continue with the basic case for now, but the reader should understand that the server may replicate the query several times and perform the entire operation on all rows combining the result at the end.

3.3 Query Expansion

Considering that the basic encoding scheme previously described is for the case where the entire column fits into a single element of A, the number of ciphertexts in both the query and database are equal. However, in the more typical case where $N > l$, since the database must be split across disjoint ciphertexts, this will create rows of ciphertexts in D. The encrypted query $\{Q_i\}_{i=0}^{M-1}$ is transmitted to the server which holds the encrypted database $\{D_i\}_{i=0}^{M-1}$. Both of which are encrypted under the same BGV public key created by the client. The query data received by the server is always a single row of ciphertexts. Therefore, the server must expand the query by creating as many replicas as there are rows of ciphertexts in the database D. For each of the query rows, the following algorithm is performed independently and the server returns a single ciphertext response for each ciphertext row in the database.

Some variants of the system may involve the database being known to the server and not necessarily be encrypted. This would relax the performance constraints and additional key management inherent to the current system described. More details on such a variant can be found in section 7. Once the server receives the query, its goal is to compute the *weighted set-intersection score* as defined by a configurable partitioning of the set of columns and weight set described in the following sections.

3.4 Mask Generation

The first step is to compute a mask. To do this, we compute ciphertexts $\{\Delta_i\}_{i=0}^{M-1}$ as follows $\Delta_i := Q_i - D_i$ which is an inter-ciphertext difference operation. Inter-ciphertext operations are entrywise operations.

Next, we must be able to perform an operation on the Δ_i values which maps all nonzero values to 0 and maps 0 to 1. Since $I(x)$ is irreducible, all nonzero elements of E are units, so $|U(E)| = |E \setminus \{0\}| = p^d - 1$ and we have $g(x)^{p^d-1} = 1$ for any nonzero $g(x) \in E$ by Lagrange's theorem. Clearly, $0^{p^d-1} = 0$ so exponentiation by $p^d - 1$ gives us this capability. This exponentiation can be naïvely implemented by squaring, resulting in a circuit with multiplication depth approximately $\log\left(p^d - 1\right)$. However, a more efficient implementation can be achieved via repeated application of the Frobenius automorphism $\sigma \colon A \longrightarrow A$ defined by $\sigma \colon x \mapsto x^p$. HE libraries supporting BGV such as HElib implement efficient maps on the ciphertext space which give the Frobenius automorphism on A. Observe that for any $g(x) \in A$, we have

$$f(g(x)) \coloneqq 1 - \prod_{i=0}^{d-1} \sigma^i\left(g(x)^{p-1}\right) = \begin{cases} 1, & \text{if } g(x) = 0 \\ 0, & \text{otherwise} \end{cases}$$

as desired, where σ^i denotes function composition rather than multiplication. Clearly, this requires only $O(\log(p-1))$ multiplication depth, followed by d Frobenius automorphisms and another $O(\log(d))$ multiplication depth.

If we denote by $f' \colon S \longrightarrow S$ the ciphertext analogue of f, then computing $f'(\Delta_i)$ for each i gives us a set of *masks* $\{K_i\}_{i=0}^{M-1}$ which we can use to generate the weighted set-intersection score.

3.5 Weighted Set-Intersection Score Computation

As previously mentioned, we define the computation of the set-intersection score which depends on:

- A collection of column subsets, which we will refer to as *data blocks*. Each data block will be written in the form F_i. We formalize this notion as a partitioning/cover of the set of columns, which we consider to be a multi-set $\{F_i\}_{i=0}^{k-1}$ where each $F_i \subseteq \{0, 1, \cdots, M-1\}$, and the F_is are not necessarily disjoint.
- A set of *weights* τ_{F_i} for each data block F_i, which we consider to be functions $\tau_{F_i} \colon F_i \longrightarrow \mathbb{Z}_p$ for convenience of notation, along with a constant term μ_{F_i}. These values μ, τ can also be encrypted and sent by the client or another third party to the server as ciphertexts instead of plaintexts; offering a greater level of privacy regarding the nature of the query, at the cost of more ciphertext to ciphertext operations and their corresponding resource overheads. Moreover, the server no longer has any ability to check the search policy implied by the values chosen. For the duration of this paper, we consider these values as not being encrypted.

Given these values, we define the weighted set-intersection score function α as follows

$$\alpha\left(\{K_j\}_{j=0}^{M-1}\right) = \prod_{i=0}^{k-1}\left(\mu_{F_i} + \sum_{j \in F_i} \tau_{F_i}(j) K_j\right) \tag{1}$$

For low values of k, this is relatively cheap to compute.

One must be careful to pick the τ and μ values, as well as the partitioning multi-set, so as to ensure that we do not overflow our modulo-p capabilities. A simple check is to compute α setting each $K_i = 1$ which would be the score obtained by a perfect database match and ensure that this value is at most $p - 1$. If not, this can be mitigated by increasing p, making the subsets F_i smaller, decreasing k, decreasing the offset values μ, or decreasing the weight values τ.

The interpretation of the score function α is essentially a weighted *geometric mean* of weighted offset *arithmetic* means of matches across subsets of the columns of the database. Note that our score function does not contain any root or division operations as in the calculation of the actual *geometric mean* and *arithmetic mean* respectively.

This has the property of allowing data blocks which give more *importance* to some columns than others according to the τ values used in the weighted offset *arithmetic mean* terms. Moreover, the μ terms can be configured in order to customize the *maximum impact* that said data block can have on the overall score α due to the multiplication performed. For example, if $\mu_{F_i} = 0$, this means that the data block F_i *must* match at least partially, otherwise the overall score will be 0. If μ_{F_i} is noticeably larger than the values of τ_{F_i}, on the other hand, that data block will not heavily impact the overall score even if the F_i columns do not match well.

Another property of the scoring function α is the fact that we may have duplicate data blocks such as some $i \neq j$ for which $F_i = F_j$. This results in the *geometric mean* becoming a *weighted geometric mean* instead, since the product in equation (1) can then contain powers of terms.

4 USAGE EXAMPLES

4.1 Database lookup

Rather than simply using a system which generates a score, we could instead use this score to perform PIR. Due to the inherent configurability of our implementation, the calculated score can provide more complex queries than a simple *keyword* lookup. Here we provide the reader with a usage example of how this works. Let us suppose a system in which our database has $M = 5$ columns. In this system, we might use $k = 4$ and take our multi-set of data blocks as the whole set repeated four times, i.e. $F_0 = F_1 = F_2 = F_3 = \{0, 1, 2, 3, 4\}$.

A family of database lookups can be implemented given this configuration; including logical operations corresponding to the additions and multiplications used to compute α given in equation 1 according to the client's desire which is encoded in the weights and offsets. For example, if a client would like to match column 0 and 2 and (3 or 4), the client would set weights as

$$\begin{aligned} \tau_{F_0}(j) &= \mathbb{1}_{j=0} \\ \tau_{F_1}(j) &= \mathbb{1}_{j=2} \\ \tau_{F_2}(j) &= \mathbb{1}_{j \in \{3,4\}} \\ \tau_{F_3}(j) &= 0 \end{aligned}$$

where $\mathbb{1}$ is an indicator function defined by

$$\mathbb{1}_P = \begin{cases} 1, & \text{if } P \text{ is true} \\ 0, & \text{if } P \text{ is false} \end{cases}$$

In this example, we would also need to set the offset values as

$$\mu_{F_0} = 0$$
$$\mu_{F_1} = 0$$
$$\mu_{F_2} = 0$$
$$\mu_{F_3} = 1.$$

This ensures that the first three terms in the product will correspond to the three logical conditions imposed with the last one being a constant true value (a 1) due to the μ values.

Now, we are able to use α not simply as a score function, but as a *matching mask* which may be raised to the $p-1^{\text{th}}$ power using Fermat's little theorem to result in 1 for matching database rows and 0 for non-matching database rows. This can be used to retrieve the matching database entries as a 0 multiplied with the database zeroes out the entry and a 1 multiplied with the database returns the data. Note that this step can be avoided in the case where only AND operations are permitted. If the entries can be assumed to be unique then a known optimization is to add the retrieved values; this reduces the communication cost as only one row has to be returned. Alternatively, one could add the entries of the mask without ever multiplying by the values, thereby performing a count operation as opposed to value retrieval so the client can learn how many matches there were.

Smaller sets for the F_is may be used for different possibilities of database lookups. For example, one may simply pick the F_is to be singletons if no logical disjunctions are required. Clearly, the set of possible logical operations which can be performed by reconfiguring the F_is is too large to be described here.

Suppose we have a database, D, against which we want to perform this lookup operation, in the earlier notation, we have the i^{th} column $(d_{0i}, d_{1i}, \cdots, d_{(N-1)i})^\intercal$ encoded and encrypted as D_i.

Likewise, we would also have a query Q. Linking back to the earlier notation again, we would have the i^{th} column $(e_i, e_i, \cdots, e_i)^\intercal$ encoded and encrypted as Q_i.

Now, we perform the subtraction operation which results in a matrix of differences with columns equal to the Δ_i values mentioned earlier given by $\Delta \coloneqq Q - D$.

Next, the scoring function f' mentioned earlier is applied to the columns of this matrix giving another matrix of the same format

$$K \coloneqq f'\left(\{\Delta_0, \Delta_1, \Delta_2, \Delta_3, \Delta_4\}\right)$$

with columns earlier referred to as K_i.

Using the values of τ and μ mentioned before, the scoring function α becomes

$$\alpha\left(\{K_0, K_1, K_2, K_3, K_4\}\right) = (K_0)(K_2)(K_3 + K_4)(1)$$

Recalling that all of these operations occur entrywise, we have the i^{th} entry of this expression as follows

$$\alpha\left(\{\kappa_{i0}, \kappa_{i1}, \kappa_{i2}, \kappa_{i3}, \kappa_{i4}\}\right)_i = \begin{cases} 2, & \text{if } \kappa_{i0} = \kappa_{i2} = \kappa_{i3} = \kappa_{i4} = 1 \\ 1, & \text{if } \kappa_{i0} = \kappa_{i2} = \kappa_{i3} = 1 \text{ and } \kappa_{i4} = 0 \\ 1, & \text{if } \kappa_{i0} = \kappa_{i2} = \kappa_{i4} = 1 \text{ and } \kappa_{i3} = 0 \\ 0, & \text{otherwise.} \end{cases}$$

Thus, using Fermat's little theorem, we raise this score to the $p-1^{\text{th}}$ power to obtain

$$\alpha\left(\{\kappa_{i0}, \kappa_{i1}, \kappa_{i2}, \kappa_{i3}, \kappa_{i4}\}\right)_i^{p-1} = \begin{cases} 1, & \text{if } \kappa_{i0} = 1 \wedge \kappa_{i2} = 1 \wedge (\kappa_{i3} = 1 \vee \kappa_{i4} = 1) \\ 0, & \text{otherwise.} \end{cases}$$

which is the mask that we require for the logical condition requested by the client. Clearly, multiplying this by the database values or the original row will extract the value desired by the client. If it is known that the values fulfilling the logical condition will be unique, then the server can accumulate with addition and return only one value instead of a set of values with the rest zeroed out.

We note that using these results, it is possible to extend our algorithm to also perform an SQL-esque inner join with the query and database if we consider the query as the left table and the database as the right table. This is discussed as further work in section 7.2.

4.2 Direct set-intersection score usage

Once $\alpha\left(\{K_j\}_{j=0}^{M-1}\right)$ has been computed by the server, it can be returned to the client immediately without performing a lookup operation, who may decrypt it.

This score can be used for a type of partial matching where the client can choose the *importance* of data to search against or the importance of combinations of data to search against. This is achievable due to configurability of the client setting the weights and offsets. Below we describe how this can be used in an example scenario of a law enforcement agency wishing to inquire about a vehicle based on an eyewitness account of a car. The law enforcement agency may be issuing queries against various small car park ANPR (Automatic Number Plate Recognition) databases to determine which may offer leads. Of course, this eyewitness account could naturally give rise to differing *importances* of attributes since the eyewitness may have differing degrees of certainty about the attributes of the sighted car, e.g. he or she did not see the registration plate but is certain about the make and fairly confident about the model of the car.

The client will be able to see how well each database entry matches with their query, according to the importances defined by the values τ and μ. Note that this does leak some information about the database, namely the position of each match in the database. In some variants, one might wish to protect this information to varying extents. Randomizing the order of the values returned may be performed by the server using the *slot permutation* capabilities present in typical implementations of the BGV scheme, which would prevent the client from learning this information.

We will now work through the example in which the weights are set differently, so that the score is more granular and would be of interest when returned immediately.

Let the database have $M = 5$ columns where the columns correspond to data in Table 1 after any required preprocessing. Moreover, the client wishes to perform queries which answer questions of the form 'how well does this query match entries of the database?' in a non-binary fashion. We may then choose $k = 4$ and the data blocks

Table 1: Mapping between column index and what the column name.

Column index	Column meaning
0	Make
1	Model
2	Registration number
3	Engine size
4	Color

may be set up-front as,

$$\begin{aligned} F_0 &= \{0, 1, 2, 3, 4\} \\ F_1 &= \{3, 4\} \\ F_2 &= \{0\} \\ F_3 &= \{1\}. \end{aligned}$$

The client now has the capability of defining a rich class of scoring importances via the τ and μ values. In this case, the law enforcement client might set,

$$\begin{aligned} \mu_{F_0} &= 1, \tau_{F_0}(j) = \begin{cases} 0, & \text{if j = 0} \\ 7, & \text{if j = 1} \\ 0, & \text{if j = 2} \\ 1, & \text{if j = 3} \\ 3, & \text{if j = 4} \end{cases} \\ \mu_{F_1} &= 5, \tau_{F_1}(j) = \begin{cases} 1, & \text{if j = 3} \\ 2, & \text{if j = 4} \end{cases} \\ \mu_{F_2} &= 0, \tau_{F_2}(0) = 1 \\ \mu_{F_3} &= 1, \tau_{F_3}(1) = 1 \end{aligned}$$

Our score function would therefore become

$$\alpha(\{K_0, K_1, K_2, K_3, K_4\}) = (1 + 7K_1 + K_3 + 3K_4)(5 + K_3 + 2K_4)(K_0)(1 + K_1)$$

where this score, a highly customized aggregate of set-intersection scores, has desirable properties, such as,

- non-matching make results in zero score due to the K_0 term;
- no other non-matching feature will zero the score;
- non-matching model drastically impacts the score, but does not zero it;
- registration plate has no impact whatsoever;
- engine size has minimal but non-zero impact;
- and, color has low impact, but larger impact than engine size.

In the case where τ and μ are encrypted, the configuration or the weights and offsets will not be revealed to the server – in the scoring the server will not know what importance the client has placed on different data for the weighted partial match.

The reader should be aware that the scoring operation in general requires a larger p than the lookup operation due to the larger numbers appearing in the τ and μ values. However, this is not too problematic since we make savings in never having to exponentiate by $p - 1$ unlike the scenario described in the previous section. This score is returned directly to the client once calculated, without needing to be operated on further for anything such as value retrieval.

5 RESULTS

This section presents and discusses the preliminary results shown in tables 3 and 4 on various lookup and weighted partial match operations which can be performed using this protocol. The timings presented in table 3 do not include Input/Ouput (IO) operations and only measure the computation time of the algorithms themselves. The code used for the implementation of the experiments presented in this paper will be provided in a future public release of HElib [21].

We assume a scenario where both the query data and database are always encrypted. We note that if these requirements are relaxed, then the database can be encoded plaintexts instead, which is expected to decrease the total computation times as well as the memory usage. Additionally, we assume this scenario allows for the configuration of the query to be known by both the client and the server. It is acknowledged that this can also be encrypted, but we leave the testing of this scenario for further work.

There are chosen parameters for two algebras, A-1 and A-2, shown in table 2. A-1 has over 128 bits of security and enables the packing of up to 55440 8-bit numbers into a single ciphertext when choosing an m value of $m = 56083$. The number of bits of the modulus Q of a freshly encrypted ciphertext was initially chosen to be $Q = 900$, which provides us with an adequate number of bits relative to the depth of the evaluation circuit. Note however that HElib actually uses a modulus of $Q = 1382$ bits. The plaintext prime is chosen to be $p = 257$, which combined with the Hensel lifting parameter $r = 1$, enables the representation of 8-bit numbers as the coefficients of the polynomials in the ciphertext slots. The maximum value that can be represented without exceeding our modulus and causing wrap-around is $p^r - 1 = 256$. However, having a polynomial of degree $d = 4$ for each slot enables the packing of four 8-bit values per slot. Thus, it is possible through packing of slots to represent a 32-bit number in each slot of our ciphertext through splitting the number across the slot coefficients. The parameter $l = \phi(m)/d$ determines the number of slots of our ciphertext. This number signifies that we can represent up to 13860 32-bit numbers per ciphertext.

Algebra A-2 differs to A-1 by choosing a large $p = 1278209$ and an $m = 159776$ giving a $d = 1$. Algebras giving a $d = 1$ can be beneficial to HE algorithms as the ciphertext slots only contain a single number in $\mathbb{Z}_p$ not a polynomial. After performing the operations outlined in this paper, the client may wish to perform further processing without having to implement HE expensive procedures forcing polynomials to perform integer operations such as in the work of Crawford *et al.* [11]. However, due to the larger p, noise management is affected so we require a larger $Q = 1800$ (HElib actually uses $Q = 2811$) which reduces the security level meaning that we need a larger $\phi(m)$ to compensate. This larger $\phi(m)$ in combination with $d = 1$ means that there are far more slots available in a ciphertext $l = 79872$. Moreover, the security we could achieve with these parameters is lower that A-1. In this case, the capacity is lower in the slots since $\lfloor \log_2(p) \rfloor = 20$ bits.

Table 2: BGV parameters used for scoring and lookup. *HElib actually generates a $Q = 1382$ bits and $Q = 2811$ bits for algebras A-1 and A-2 respectively.

Parameters	A-1	A-2
m	56083	159776
p	257	1278209
r	1	1
Q	900 bits*	1800 bits*
$\phi(m)$	55440	79872
d	4	1
λ	> 128 bits	> 80 bits
l	13860	79872

For the algebra A-1, the database is initially created to have 84 rows of ciphertexts. Each row has 4 columns where each column in a row is represented using a single ciphertext. Using the parameters in table 2, our database holds a maximum of $lN = 1164240$ 32-bit entries per column and thus $lNM = 4656960$ values in total. The query data is a single row containing the same number of columns as the database. However, if both the query and database are encrypted as in this scenario, then the query can only hold a single entry. To overcome this limitation, see discussion in section 7.3.

In contrast, A-2 has 8 rows of 4 columns of ciphertexts. This means our database holds a maximum of $lN = 638976$ 20-bit entries per column and thus $lNM = 2555904$ values in total. The reason for fewer entries is that during experimentation we were not successful in finding an algebra that had enough bits in the modulus chain and achieved a reasonable security level (at least above 80).

5.1 Hardware

An IBM z14® was used to gather the reported preliminary results. Our OS (Ubuntu 18.04.04 LTS) was on a single logical partition (LPAR) with the following configuration:

- 32 cores @5.2 GHz;
- 10 Cores per chip;
- Cache L1 (I-128 KiB, D-128 KiB)/core;
- Cache L2 (I-2 MiB, D-4 MiB)/core;
- Cache L3 128 MiB shared per chip;
- 1 TiB Memory;
- 1.5 TiB Disk Storage.

5.2 Basic Lookup Operations

To show the configurability of lookup queries using this protocol, we provide the results of four basic types of queries that are available to the user. The first query type is the “same” query which is a query that returns a match if a single (specified) column of the query matches the same column of an entry in the database. The second query type is the “and” query which returns a match if both column a and column b of the query data match an entry of the database. The third query type is the “or” query which returns a match if either column a or column b of the query match an entry of the database.

The final query type is a combination of an AND inside an OR. The query that we use is a OR (b AND c), which we call the “expand” query type. The reason for this is that the implementation requires the query to be in conjunctive normal form. We need to expand the query so that we have the AND operation as the outermost operation resulting in the reinterpretation of the aforementioned query as a OR (b AND c) $\equiv$ (a OR b) AND (a OR c). This query returns a match if column a of the query matches an entry of the database or if both column b and column c match the database.

The timing results presented in table 3 appear to be promising for practical use, albeit within an *acceptable* time frame. We understand that ‘acceptable time frame’ is a gray area, but here we consider a useful operation or task done under HE within a few hours generally acceptable. The logically simplest “same” query produce the fastest times out of the different query types, followed by (on average) the “and” query and the “or” query. In contrast, the more complex “expand” query produced the slowest times as expected. It is evident from the table that the time taken seems to scale well by decreasing with increasing number of threads used; That is to say, the time taken for a lookup approximately halves as the number of threads double.

Achieving times of 456 seconds (7 minutes 36 seconds) and 10200 seconds (2 hours 50 minutes) with 32 threads to 1 thread respectively to perform an “expand” lookup for parameters of A-1. Similarly, for the same operation of parameters of A-2 we achieve times of 746 seconds (12 minutes 26 seconds) to 15462 seconds (4 hours 4 minutes 22 seconds) with 32 threads to 1 thread, respectively.

Due to the query configuration being publicly known, it is possible to perform extra optimizations based on the type of query being calculated. The resultant mask of the lookup operation is always a mask of encrypted 0s and 1s, representing the location of matching rows in the database. However, if the query contains an OR, then there is a possibility of obtaining repeated matches. To remove the repetitions we use Fermat’s little theorem $a^{p-1} \equiv 1 \mod p$ to map any non-zero results to 1. Note however that this is only necessary when the query contains an OR and since the configuration of the query is known; our implementation checks the query to decide if the extra Fermat’s little theorem operation is needed. This explains why the “same” and “and” queries produce faster times than the queries containing an OR.

The results from table 4 show that the maximum resident set size of the program ranges between ≈ 26 GiB to ≈ 32 GiB for A-1 and ≈ 65 GiB to ≈ 115 GiB for A-2 depending on the number of threads being utilized. The increase in memory usage relative to the increase number of utilized threads is understandable as each process will produce its own temporary results thus increasing the number of threads. For completeness in understanding some of this memory consumption, the HElib serialized objects produced by parameters of A-1 in table 2 that had to be loaded were the context and public key 2.8 GiB; the encrypted query data 54 MiB; and the encrypted database 4.4 GiB. Additionally, in A-2 these objects had the following sizes, context and public key 8.7 GiB; the encrypted query data 152 MiB; and the encrypted database 1.2 GiB. The results from table 4 clearly show that memory usage for the same operations are considerably higher for A-2 than A-1. This can be attributed to the considerably larger $\phi(m)$ and Q of A-2.

Table 3: Timing results for scoring and lookup operations for both A-1 and A-2.

Threads	A-1 time (s)					A-2 time (s)				
	scoring	lookup				scoring	lookup			
		same	or	and	expand		same	or	and	expand
32	527	368	401	420	456	552	485	730	493	746
16	783	644	722	704	775	939	873	1122	888	1138
8	1374	1227	1404	1292	1481	1787	1718	1958	1737	1968
4	2541	2392	2543	2374	2600	3499	3365	3933	3456	3950
2	5138	4606	5202	4739	5254	6926	6754	7845	6828	7905
1	9750	9274	10194	9497	10200	13777	13516	15400	13551	15462

5.3 Weighted Partial Match

The results of the weighted partial match showed comparable performance to the basic lookup experiments. From table 3, the scoring times appear to also scale in similar fashion to the lookup operations as reducing with increasing number of threads. We achieved a weighted score calculation in 9750 seconds (2 hours 42 minutes 30 seconds) when run single-threaded however this reduced to 527 seconds (8 minutes 47 seconds) while using 32 threads for parameters of A-1. Similarly, for parameters of A-2 we achieved a time of 13777 seconds (3 hours 49 minutes 37 seconds) when run single-threaded and 552 seconds (9 minutes 12 seconds) when using 32 threads.

The times for calculating the weighted scores were higher than the times of some of the basic lookup operations, most notably, the "same" and "and" queries. This is due to the query used for scoring being more complex as it contains both OR and AND operations. It is clear to see that for this reason the times for the weighted partial match is similar to the times for the "expand" query type.

6 CONCLUSION

Private queries on a database are required where little to no information is leaked to the database itself. Maturing HE technology provides the primitives for constructing a homomorphically encrypted database.

We have described how to create a highly configurable PSI implementation. With this fundamental implementation, we can achieve several useful operations. Two such operations were explored in this paper, namely complex query lookups and weighted partial matching on a database. With this implementation, there is no longer a need for bespoke programs for each type of operation. A configuration can be passed to a generic implementation instead.

Table 4: Memory usage results for scoring and lookup operations.

Threads	A-1 (GiB)		A-2 (GiB)	
	scoring	lookup	scoring	lookup
32	29.8	31.7	114.0	115.0
16	27.4	29.6	89.2	89.9
8	26.2	28.4	76.7	77.2
4	25.3	27.4	69.9	70.7
2	23.8	25.8	66.7	67.2
1	23.7	25.8	64.8	65.2

Preliminary results demonstrate that the database lookup generating the necessary mask or a weighted partial match generating a score can be performed on a z14 in approximately 9 minutes on over a million entries. It is quite clear that a multithreaded solution reduces the time for this operation substantially and the time appears to scale very well with increasing number of threads. More precisely, the time taken is approximately halving when doubling the number of threads.

Further work is required to leverage this technology and we provide some direction in section 7. This work provides the basic operations required to develop future HE relational databases.

7 FURTHER WORK

7.1 Identity Matching for Entity Resolution

One other use we considered for implementation is an operation of identity matching for entity resolution. Let us assume a scenario in which the server holds a graph $G = (V, E)$, with V as the set of vertices and $E \subset V \times V$ the set of edges. We begin by describing this undirected, unweighted case. In this case, the goal is for the client to ask the question 'how similar is this vertex v to all of the existing vertices in the graph?' without revealing v to the server.

In order to resolve this, we begin by fixing an encoding of the graph into our database structure described in section 3.2 as follows. Let $N = M = |V|$, and fix a public ordering $v_0, v_1, \ldots, v_{N-1}$ of the vertex set V. That is to say, both parties know this ordering. Now, we let the i^{th} row of the database have j^{th} entry equal to 1 if $v_i v_j \in E$ and 0 if $v_i v_j \notin E$.

Now that the (binary) database is encoded, the query can be encoded in the exact same way. For a query vertex v_*, the client must form the binary vector given by 1s in the places corresponding to the neighbors of v_*, and 0s elsewhere. In the notation of section 3.2, this query will consist of $M = |V|$ plaintexts $\{P_i\}_{i=0}^{M-1}$, where each P_i is just a repetition of 1 if $v_* v_i$ is a connection, and a repetition of 0 otherwise. The client may then decide upon a set of *importances* for the connections, encoding these into τ and μ values as before. Then, the exact same process as described in sections 4.1 and 4.2 can be applied, resulting in a score or database retrieval based on a graph-wise similarity.

Several minor modifications to this setup are possible in order to extend and generalize this scenario. Firstly, note that the database previously described is symmetric, since the graph G was taken to be undirected. However, this can easily be generalized to a directed graph by following precisely the same rules of encoding as previously mentioned. We merely relax the requirement of $v_i v_j \in E \iff v_j v_i \in E$ which we had previously assumed, thus it no longer applies.

The next obvious generalization is to enable the use of weighted graphs. For example, if the graph corresponded to a social network of people who might know each other, we might take the weight of an edge to be equal to 0 if the people do not know each other, 1 if they are acquaintances, and 2 if they are friends. In this scenario, we simply enter that weight as the database entry, so during the query phase we have a match if and only if the same edge exists with the exact same weight. Typically, we might keep the weighting of the graph rather coarse-grained, as in this $\{0, 1, 2\}$ example, since 'near misses' cannot be identified.

7.2 Development of HEQL

A method for computing basic query operations is described in sections 4.1 and 4.2. This work can be further expanded to perform various SQL-esque operations such as table joins. We propose that the operations defined in this paper are the way forward to develop an SQL-esque query language for a homomorphic enabled database, or better referred to as a Homomorphic Encryption Query Language (HEQL).

For example, consider the query as a table which we will refer to as the left table and refer to what we have so far called the database as the right table. Using the output of the database lookup operation described in section 4.1, we can use this mask to perform an inner join of the left and right tables.

Aggregate masks are required, ϵ_L and ϵ_R, for the left and right tables, respectively for data extraction or retrieval. The ϵ_R can then be multiplied with the right table columns to extract the desired row from the right table as $R_j = \epsilon_R D_j$, where R_j is the j^{th} column element in the join result of the right table. The same can be done with the left table to obtain the left part of the join $L_\rho = \epsilon_L Q_\rho$ for the ρ^{th} row of the left table.

The aggregate masks can be computed from aggregating the masks computed for the lookup operation in section 4.1. For example, performing a "same" lookup query, which returns a matching entry if the first column of both tables match, produces the mask α_i, a ciphertext for each i^{th} row,

$$\alpha_i(\kappa_{i0}) = \begin{cases} 1, & \text{if } \kappa_{i0} = 1 \\ 0, & \text{otherwise.} \end{cases}$$

In order to retrieve the corresponding matching row from the left table, the rows of ciphertexts of these masks can be summed to produce a single aggregate mask for each row of the left table. Using Fermat's little theorem, we raise the aggregated mask ϵ to the $p^d - 1$, $\epsilon_L = (\Sigma_i \alpha_i)^{p^d-1}$. Similarly, for the rows of the right table the aggregate mask can be computed from the summation of the masks $\epsilon_R = \left(\Sigma_\rho \alpha_\rho\right)^{p^d-1}$.

After extraction of both left and right tables the resultants can be concatenated together. Note that this will produce a repeated entry for the column that was used to find the matches. If the queried column is known then it would be trivial to remove this from the entire result set of the inner join by the client receiving the result. Attention should be drawn to the fact that the procedure described does not result in an inner join that one would expect from SQL but rather it is a complex PSI of the two tables by a query.

Moreover, it is possible to perform more complex joins through differing the query type used to obtain the mask as described in section 5.2. This is similar to performing binary operations within an *ON* command in SQL. The joining operation itself should be the same regardless of the query aside from removing potential duplicate columns from the two tables.

Due to the method in which the query data is encoded, the left table will contain a great deal of duplicated data. However, if the database is not encrypted, then this duplication of the query data can be optimized away. This is discussed in more detail in the following section 7.3.

7.3 Optimizations

The immediate work is to make improvements to the implementation in terms of multi-threading capabilities and computational efficiency at both the application and HElib level. Subsequent to this, more empirical data can be gathered for other likely scenarios such as the database not being encrypted but only encoded if the server actually owns the data which would speed up computation and reduce memory usage. In section 5, we assume that the configurations of the lookup query and partial matching are in the clear and known to both client and server. This however can change to where the configurations can themselves be encrypted meaning that the client can configure but not allow the server to know the configuration and thus not know exactly what computation is performed. Moreover, various other parameters and security levels can be explored and used to gain real timings of the full pipeline including the IO costs. This can be used to further analyze the pipeline, enabling further performance improvements.

Since the majority of the computation time is spent on the mask generation, it is possible to incorporate some of the previously implemented optimizations such as partitioning [5]. This involves splitting the query data set into smaller subsets and computing the private query independently for each subset in parallel. A single mask will be produced per subset which can simply be aggregated before being sent back to the client. This will reduce the total computation depth to that of each subset, however it will increase the total communication cost as each computer will require access to the database and public keys.

Once the mask has been computed, it can be used to extract the data from the database by multiplying the mask across the database entrywise. If the data is guaranteed to be unique, then we can trivially reduce the communication cost by summing the resultant rows of the database and return a single row instead. However, it can be argued that in most use cases for database queries there will be several results per column across the rows. Despite this, we could explore using techniques such as outlined by Akavia *et al.* [1] or statistical techniques via binning to lower the communication cost regardless of receiving several results per column.

In this paper, the encrypted query data was a row of M ciphertexts where each ciphertext contained an N-fold repetition of the query datum corresponding to a single column of the database. It is clear that in the scenario where the client has multiple queries, then the server will receive M ciphertexts for each query. This can however be optimized if the database is not encrypted. In this latter scenario where the database knows its own data, each entry can be represented as a single plaintext element as plaintexts are considerably less expensive both computationally and in terms of memory usage. This enables the client to pack each individual query across the slots of each ciphertext in a similar manner to the database as described in section 3.2. Therefore, if the client has $N < l$ queries, these can be packed into a single row of ciphertexts, thus greatly reducing the communication cost, memory usage, and computation cost.

ACKNOWLEDGMENTS

This research is based upon work supported by the Office of the Director of National Intelligence (ODNI), Intelligence Advanced Research Projects Activity (IARPA), via 2019-19-020700006. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies, either expressed or implied, of ODNI, IARPA, or the U.S. Government. The U.S. Government is authorized to reproduce and distribute reprints for governmental purposes notwithstanding any copyright annotation therein.

REFERENCES

[1] Adi Akavia, Dan Feldman, and Hayim Shaul. 2018. Secure Search via Multi-Ring Fully Homomorphic Encryption. *IACR Cryptology ePrint Archive* 2018 (2018), 245.

[2] Zvika Brakerski, Craig Gentry, and Vinod Vaikuntanathan. 2014. (Leveled) Fully Homomorphic Encryption without Bootstrapping. *ACM Transactions on Computation Theory* 6, 3 (2014), 13. https://doi.org/10.1145/2633600

[3] Capstone 2016. S2 - Secure Information Processing. http://nis-ita.org/capstone.

[4] Hao Chen, Zhicong Huang, Kim Laine, and Peter Rindal. 2018. Labeled PSI from fully homomorphic encryption with malicious security. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. 1223–1237.

[5] Hao Chen, Kim Laine, and Peter Rindal. 2017. Fast private set intersection from homomorphic encryption. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. 1243–1255.

[6] Jung Hee Cheon, Andrey Kim, Miran Kim, and Yong Soo Song. 2017. Homomorphic Encryption for Arithmetic of Approximate Numbers. In *ASIACRYPT (1) (Lecture Notes in Computer Science)*, Vol. 10624. Springer, 409–437.

[7] Ilaria Chillotti, Nicolas Gama, Mariya Georgieva, and Malika Izabachène. 2016. TFHE: Fast Fully Homomorphic Encryption Library. https://tfhe.github.io/tfhe/.

[8] Lukasz Chmielewski and Jaap-Henk Hoepman. 2008. Fuzzy private matching. In *2008 Third International Conference on Availability, Reliability and Security*. IEEE, 327–334.

[9] Benny Chor, Oded Goldreich, Eyal Kushilevitz, and Madhu Sudan. 1995. Private information retrieval. In *Proceedings of IEEE 36th Annual Foundations of Computer Science*. 41–50. https://doi.org/10.1109/SFCS.1995.492461

[10] E. F. Codd. 1970. A Relational Model of Data for Large Shared Data Banks. *Commun. ACM* 13, 6 (June 1970), 377–387. https://doi.org/10.1145/362384.362685

[11] Jack L. H. Crawford, Craig Gentry, Shai Halevi, Daniel Platt, and Victor Shoup. 2018. Doing Real Work with FHE: The Case of Logistic Regression. In *Proceedings of the 6th Workshop on Encrypted Computing & Applied Homomorphic Cryptography, WAHC@CCS 2018*, Michael Brenner and Kurt Rohloff (Eds.). ACM, 1–12. https://doi.org/10.1145/3267973.3267974 https://eprint.iacr.org/2018/202.

[12] W. Diffie. 1988. The first ten years of public-key cryptography. *Proc. IEEE* 76, 5 (1988), 560–577.

[13] Yarkın Doröz, Berk Sunar, and Ghaith Hammouri. 2014. Bandwidth Efficient PIR from NTRU. In *Financial Cryptography and Data Security*, Rainer Böhme, Michael Brenner, Tyler Moore, and Matthew Smith (Eds.). Springer, Berlin, Heidelberg, 195–207.

[14] Taher El Gamal. 1985. A Public Key Cryptosystem and a Signature Scheme Based on Discrete Logarithms. In *Proceedings of CRYPTO 84 on Advances in Cryptology*. Springer-Verlag, New York, NY, USA, 10–18. http://dl.acm.org/citation.cfm?id=19478.19480

[15] Junfeng Fan and Frederik Vercauteren. 2012. Somewhat Practical Fully Homomorphic Encryption. *IACR Cryptology ePrint Archive* 2012 (2012), 144.

[16] Craig Gentry. 2009. Fully Homomorphic Encryption Using Ideal Lattices. In *Proceedings of the 41st Annual ACM Symposium on Theory of Computing (STOC '09)*. ACM, New York, NY, USA, 169–178. https://doi.org/10.1145/1536414.1536440

[17] Craig Gentry, Shai Halevi, and Nigel P Smart. 2012. Homomorphic evaluation of the AES circuit. In *Annual Cryptology Conference*. Springer, 850–867.

[18] Shai Halevi and Victor Shoup. 2013. Design and implementation of a homomorphic-encryption library. *IBM Research (Manuscript)* 6 (2013), 12–15.

[19] Shai Halevi and Victor Shoup. 2014. Algorithms in HElib. In *CRYPTO (1) (Lecture Notes in Computer Science)*, Vol. 8616. Springer, 554–571.

[20] Shai Halevi and Victor Shoup. 2015. Bootstrapping for HElib. In *EUROCRYPT (1) (Lecture Notes in Computer Science)*, Vol. 9056. Springer, 641–670.

[21] HElib 2020. HElib (release 1.0.1). https://github.com/homenc/HElib.

[22] iDASH 2015. 2015 iDASH Healthcare Privacy Protection Challenge Morning Session. https://youtu.be/lfte4TE0qvA.

[23] iDASH 2015. iDASH Security and Privacy Workshop 2015. http://www.humangenomeprivacy.org/2015.

[24] Vladimir Kolesnikov, Ranjit Kumaresan, Mike Rosulek, and Ni Trieu. 2016. Efficient batched oblivious PRF with applications to private set intersection. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. 818–829.

[25] Lattigo 2020. Lattigo 1.3.1. Online: http://github.com/ldsec/lattigo. EPFL-LDS.

[26] Catherine Meadows. 1986. A more efficient cryptographic matchmaking protocol for use in the absence of a continuously available third party. In *1986 IEEE Symposium on Security and Privacy*. IEEE, 134–134.

[27] Jakob Naucke, Hamish Hunt, Jack Crawford, Enrico Steffinlongo, Oliver Masters, and Flavio Bergamaschi. 2019. Homomorphically Securing AI at the Edge. In *Proceedings of the First International Workshop on Challenges in Artificial Intelligence and Machine Learning for Internet of Things*. 32–38.

[28] Pascal Paillier. 1999. Public-Key Cryptosystems Based on Composite Degree Residuosity Classes. In *Advances in Cryptology — EUROCRYPT '99*, Jacques Stern (Ed.). Springer, Berlin, Heidelberg, 223–238.

[29] PALISADE 2020. PALISADE (release 1.9.2). https://gitlab.com/palisade/palisade-release.

[30] Benny Pinkas, Thomas Schneider, and Michael Zohner. 2018. Scalable private set intersection based on OT extension. *ACM Transactions on Privacy and Security (TOPS)* 21, 2 (2018), 1–35.

[31] SEAL 2020. Microsoft SEAL (release 3.5). https://github.com/Microsoft/SEAL. Microsoft Research, Redmond, WA.

[32] Qingsong Ye, Ron Steinfeld, Josef Pieprzyk, and Huaxiong Wang. 2009. Efficient fuzzy matching and intersection on private datasets. In *International Conference on Information Security and Cryptology*. Springer, 211–228.