Achieving GWAS with Homomorphic Encryption

Jun Jie Sim[1], Fook Mun Chan[1], Shibin Chen[1], Benjamin Hong Meng Tan[1], and Khin Mi Mi Aung[1]

[1]Institute of Infocomm Research, Singapore

February 12, 2019

Abstract

One way of investigating how genes affect human traits would be with a genome-wide association study (GWAS). Genetic markers, known as single-nucleotide polymorphism (SNP), are used in GWAS. This raises privacy and security concerns as these genetic markers can be used to identify individuals uniquely. This problem is further exacerbated by a large number of SNPs needed, which produce reliable results at a higher risk of compromising the privacy of participants.

We describe a method using homomorphic encryption (HE) to perform GWAS in a secure and private setting. This work is based on a semi-parallel logistic regression algorithm proposed to accelerate GWAS computations. Our solution involves homomorphically encrypted matrices and suitable approximations that adapts the original algorithm to be HE-friendly. Our best implementation took 24.70 minutes for a dataset with 245 samples, 4 covariates and 10643 SNPs.

We demonstrate that it is possible to achieve GWAS with homomorphic encryption with suitable approximations.

Background

Genome-wide association study (GWAS) compares genetic variants, single-nucleotide polymorphisms (SNP), to see if these variants are associated with a particular trait. The model used in GWAS is essentially logistic regression, evaluated one SNP at a time, corrected with covariates like age, height and weight. The number of SNPs analyzed can easily grow up to 30 million. It is estimated that it can take around 6 hours for 6000 samples and 2.5 million SNPs [1].

Some suggest that cloud computing could offer a cost-effective and scalable alternative that allows research to be done, given the exponential growth of genomic data and increasing computational complexity of genomic data analysis. However, privacy and security are primary concerns when considering these cloud-based solutions.

It was shown in 2004 by Lin *et. al.* [2] that as little as 30 to 80 SNPs could identify an individual uniquely. Homer *et. al.* [3] further demonstrated that even when DNA samples are mixed among 1000 other samples, individuals could be identified. In light of these discoveries, regulations concerning biological data are being updated [4]. The privacy and security of DNA-related data are now more important than ever.

Homomorphic Encryption (HE) is a form of encryption where functions, f, can be evaluated on encrypted data $x_1, \ldots, x_n$, yielding ciphertexts that decrypt to $f(x_1, \ldots, x_n)$. Putting it in the context of GWAS, genomic data can be homomorphically encrypted and sent to a computational server. The server then performs the GWAS computations on the encrypted data, before sending the encrypted outcome to the data owner for decryption. We argue that this would ensure the privacy and security of genomic data: Throughout the entire process, there is no instance where the server can access the data in its raw, unencrypted form, preserving the privacy of the data. Additionally, since the data is encrypted, no adversary would be able to make sense of the ciphertexts. The data is thus secured on the computational server.

Motivated by these concerns, the iDASH Privacy & Security Workshop [5] has organized several competitions on secure genomics analysis since 2011. The aim of these competitions is to evaluate methods that provide data confidentiality during data analysis in a cloud environment.

In this work, we provide a solution to Track 2 of the iDASH 2018 competition – *Secure Parallel Genome-Wide Association Studies using Homomorphic Encryption*. The challenge of this task was to implement the semi-parallel GWAS algorithm proposed by Sikorska *et. al.* [6], which outperforms prior methods by about 24 times, with homomorphic encryption.

We propose a modification of the algorithm by Sikorska *et. al.* [6] for homomorphically encrypted matrices. We developed a caching system to minimize memory utilization while maximizing the use of available computational resources. Within the constraints of the competition, including a virtual machine with 16GB of memory and 200GB disk space, a security level of at least 128 bits and at most 24 hours of runtime, our solution reported a total computation time of 717.20 minutes. Our best implementation using a more efficient HE scheme, which was not available during the competition, achieved a runtime of 24.70 minutes.

In the following section, we will first define some notations used in this paper. We will begin by describing the CKKS homomorphic encryption scheme that was used to implement the GWAS algorithm. We describe our methods for manipulating homomorphic matrices that are crucial to our solution. We start with our implementation of logistic regression with HE. Following that, we adapted the GWAS algorithm using suitable approximations to simplify the computations for HE, while preserving the accuracy of the model. We also detail some optimizations that were used to accelerate the runtime. Finally, we present our results and provide some discussion about our results.

Notation

Notation for HE

Let N be a power-of-two integer and $\mathcal{R} = \mathbb{Z}[x]/\langle x^N + 1\rangle$. For some integer ℓ, denote $\mathcal{R}_\ell = \mathcal{R}/2^\ell\mathcal{R} = \mathbb{Z}_{2^\ell}[x]/\langle x^N + 1\rangle$. We let λ be the security parameter where attacks on the cryptosystem require approximately $\Omega(2^\lambda)$ bit operations. We use $z \leftarrow D$ to represent sampling z from a distribution D. We initialize the following parameters with the scheme, p, L, N.

Notation for GWAS

The number of samples, covariates and SNPs are denoted as n, d and k respectively. Matrices are denoted in bold font uppercase letters. Let the covariates matrix be denoted as $\mathbf{X}$ and the SNP matrix as $\mathbf{S}$. The rows of $\mathbf{X}$ or $\mathbf{S}$ represent the covariates or SNPs from one sample respectively. We denote the rows as $\mathbf{x}_i$. Vectors are denoted in bold font lowercase letters. Let the response vector be denoted as $\mathbf{y}$. The vector of weights from the logistic model is denoted as $\boldsymbol{\beta}$ and the corresponding vector of probabilities is denoted as $\mathbf{p}$. The vector of SNP effects is denoted as $\mathbf{s}$. Let $\mathbf{W}$ be a n by n diagonal matrix whose entries are $p_i(1-p_i)$ for $i = 1, \ldots, n$. The transpose of a vector $\mathbf{v}$ is denoted as $\mathbf{v}^\intercal$. We let $\lceil\cdot\rceil_{\text{PO2}}$ denote rounding up to the nearest power-of-two.

Methods

Homomorphic Encryption

HE was first proposed by Rivest *et. al.* [7] more than 40 years ago while the first construction was proposed by Gentry [8] only a decade ago. For this work, we adopt the HE scheme proposed by Cheon et al. [9] (CKKS), which enables computation over encrypted approximate numbers. As GWAS is a statistical function, the CKKS HE scheme is the prime candidate for efficient arithmetic.

Most HE schemes are based on "noisy" encryptions, which applies some "small" noise to mask messages in the encryption process. For HE, a noise budget is determined when the scheme is initialized and computing on ciphertexts depletes this pre-allocated budget. Once the noise budget is expended, decryption would return incorrect results. The CKKS scheme [9] treats encrypted numbers as having some initial precision, with the masking noise just smaller than the precision. However, subsequent operations on ciphertexts increase the size of noises and reduce the precision of the messages encrypted within. Thus, decrypted results are approximations of their true value.

The noise budget for the CKKS scheme is initialized with the parameter L. For every multiplication, the noise budget is subtracted by the integer p. The noise budget for a given ciphertext is denoted as ℓ. When the message is just encrypted, $\ell = L$. When $\ell < p$, the noise budget is said to be depleted.

We provide a brief description of the CKKS scheme and highly encourage interested readers to refer to [9] for the full details.

- **KeyGen(1^λ):** Let 2^L be the initial ciphertext modulus and choose appropriate distributions χ_{key}, χ_{err}, χ_{enc}. Sample a secret $s \leftarrow \chi_{key}$, random $a \leftarrow \mathcal{R}_L$ and error $e \leftarrow \chi_{err}$. Set the secret key as $sk \leftarrow (1, s)$, public key as $pk \leftarrow (b, a) \in \mathcal{R}_L^2$ where $b = -a \cdot s + e \pmod{L}$. Finally, sample $a' \leftarrow \chi_{enc}$, $e' \leftarrow \chi_{err}$ and set the evaluation key $evk \leftarrow (b', a')$, where $b' = -a' \cdot s + e' + L \cdot s^2 \pmod{2^{2L}}$.
- **Encrypt(pk, m):** For $m \in \mathcal{R}$, sample $v \leftarrow \chi_{enc}$ and $e_0, e_1 \leftarrow \chi_{err}$. Let $v \cdot pk + (m + e_0, e_1) \pmod{2^L}$ and output (v, L).
- **Decrypt(sk, ct):** For $ct = ((c_0, c_1), \ell) \in \mathcal{R}_\ell^2$, output $c_0 + c_1 \cdot s \pmod{2^\ell}$
- **Add(ct_1, ct_2):** For $ct_1 = ((c_{0,1}, c_{1,1}), \ell), ct_2 = ((c_{0,2}, c_{1,2}), \ell)$, compute $(c_0', c_1') \leftarrow= (c_{0,1}, c_{1,1}) + (c_{0,2}, c_{1,2}) \pmod{2^\ell}$ and output $(c_0', c_1'), \ell)$.
- **Mult(ct_1, ct_2):** For ciphertexts $ct_1 = ((c_{0,1}, c_{1,1}), \ell)$ and $ct_2 = ((c_{0,2}, c_{1,2}), \ell)$, let $(d_0, d_1, d_2) = (c_{0,1}c_{0,2},\ c_{1,1}c_{0,2} + c_{0,1}c_{1,2}, c_{1,1}c_{1,2}) \pmod{2^\ell}$. Compute $(c_0', c_1') \leftarrow (d_0, d_1) + \lfloor 2^{-L} \cdot d_2 \cdot evk \pmod{2^\ell} \rceil$ and output $(c_0', c_1'), \ell)$.
- **Rescale(ct, p):** For a ciphertext $ct = ((c_0, c_1), \ell)$ and an integer $p \leq \ell$, output $((c_0', c_1'), \ell - p)$, where $(c_0', c_1') \leftarrow \lfloor 2^{-p} \cdot (c_0, c_1) \rceil \pmod{2^{\ell - p}} \rceil$.

With the CKKS scheme, we are able to encode $N/2$ complex numbers into a single element in its message spaces, $\mathcal{R}$. This allows us to view a ciphertext as an encrypted array of fixed point numbers. Let $\phi : \mathbb{C}^{N/2} \to \mathcal{R}$,

- **Encode($z_1, z_2, \ldots, z_{N/2}$):**
  Output $m = \phi(z_1, z_2, \ldots, z_{N/2})$.
- **Decode(m):**
  Output $(z_1, z_2, \ldots, z_{N/2}) = \phi^{-1}(m)$.

Informally, $\phi(\cdot)$ maps $(z_1, \ldots, z_{N/2})$ to the vector $(\zeta_j)_{j \in \mathbb{Z}_N^*}$, where $\zeta_j = \lfloor z_j \rceil$ and $\zeta_{N-j} = \lfloor \overline{z_j} \rceil$ for $1 \leq j \leq N/2$. This (ζ_j) is then mapped to an element of $\mathcal{R}$ with the inverse of the canonical embedding map. $\phi^{-1}(\cdot)$ is straightforward, an element in $\mathcal{R}$ is mapped to a N-dimensional complex vector with the complex canonical embedding map and then the relevant entries of the vector is taken to be the vector of messages.

The ability to encode multiple numbers into one ciphertext allows us to reduce the number of ciphertexts used and compute more efficiently. We refer to each number encoded as a slot of the ciphertext. This offers a SIMD-like structure where the same computation on all numbers within a ciphertext can be done simultaneously. This means that adding or multiplying two ciphertexts together would be equivalent to adding or multiplying each slot simultaneously.

The ciphertext of the CKKS scheme can also be transformed into another ciphertext whose slots are a permutation of the original ciphertext.

- `Rotate`(ct, r): Outputs ct' whose slots are rotated to the right by r positions.

Homomorphic Matrix Operations

In this section, we describe our method of encoding matrices with HE. The batching property of the CKKS scheme allows us to treat ciphertexts as encrypted arrays. With this, we propose 4 methods of encoding a matrix with ciphertexts.

Column-Packed (CP) Matrices.

This is our primary method of encoding a matrix. We encrypt each column of a matrix in one ciphertext and therefore a matrix will be represented by a vector of ciphertexts. This method of encoding a matrix was suggested by Halevi and Shoup in [10].

We require a function, `Replicate` that takes a vector ν of size n and returns vectors ν_1, ν_2, ..., ν_n where ν_i for $i = 1, \ldots, n$, is $\nu[i]$ in all positions. We describe in Algorithm 9, a naive version of `Replicate`. The reader is advised to refer to [10] for details on implementing a faster and recursive variant.

Algorithm 1: `Replicate`

Input: Ciphertext ct
Output: vector⟨Ciphertext⟩ result

```
1 vector⟨int⟩ one = [0, 0, ..., 0]
2 for i = 0 to ct.size do
3 |   one [i] = 1
4 |   Ciphertext temp = ct * one
5 |   for j = 0 to log2(ct.size) − 1 do
6 |   |   temp += Rotate(temp, 2^j)
7 |   end
8 |   result[i] = temp
9 end
```

We first define matrix-vector multiplication between a CP matrix and a vector in Algorithm 4. First, we invoke `Replicate` on the vector. Next, we multiply each column in the left-hand side (LHS) matrix with its corresponding ν_i. Finally, sum up all ciphertexts and this will give the matrix-vector product.

Algorithm 2: `CP-MatVecMult`

Input: vector⟨Ciphertext⟩ A, Ciphertext b
Output: Ciphertext result

```
1 vector⟨Ciphertext⟩ colrep ← Replicate(b)
2 for i = 1 to colrep.size do
3 |   result += A(i) ∗ colrep(i)
4 end
```

Matrix multiplication between CP matrices is defined as an iterative process over `CP-MatVecMult` between the LHS matrix and the columns of the right-hand side (RHS) matrix. This is described in Algorithm 3

Algorithm 3: CP-MatMult

Input: vector⟨Ciphertext⟩ A, vector⟨Ciphertext⟩ B
Output: vector⟨Ciphertext⟩ result

for $i = 1$ **to** B.`size` **do**
  result$[i]$ = `CP-MatVecMult`$((A, B(i))$
end

Column-Compact-Packed (CCP) Matrices.

In the case where the entries of a matrix can fit within a single vector, we concatenate its columns and encrypt that in one ciphertext. For this type of matrix, we are mainly concerned with the function `colSum` which returns a vector whose entries are the sum of each column. We present the pseudocode in Algorithm 3. This is achieved by a series of rotations and additions. However, we do not rotate for all slots of the vector, but rather $\log_2(colSize)$, where $colSize$ is the number of rows in the CCP matrix. We note here that the final sums are stored in every $colSize$ slots, starting from the first slot.

Algorithm 4: `colSum`

Input: Ciphertext ν
Output: Ciphertext result

for $i = 0$ **to** $\log_2($`colsize`$) - 1$ **do**
  result += `Rotate`$(C, 2^i)$
end

Row-Packed (RP) Matrices.

For this encoding, we encrypt rows of a matrix into a ciphertext, representing them with a vector of ciphertexts just like CP matrices. In this work, we only consider matrix-vector multiplication between an RP matrix and a vector. Multiplication of an RP matrix by a CP matrix is a lot like naive matrix multiplication.

To compute the multiplication of an RP matrix with a vector, we define the dot product between two vectors encoded in two ciphertexts in Algorithm 4. For that, we first multiply the ciphertexts together, which yields their component-wise products. Then, we apply rotations to obtain the dot product in every slot of the vector.

Algorithm 5: `DotProd`

Input: Ciphertext A, Ciphertext b
Output: Ciphertext result

Ciphertext $C \leftarrow A * B$
for $i = 0$ **to** $\log_2(C.\texttt{size}) - 1$ **do**
  $C \mathrel{+}= \texttt{Rotate}(C, 2^i)$
end

With `DotProd`, we apply it over the rows of the RP matrix with the vector, producing several ciphertexts that each contain the dot product between a row and said vector. Though a series of masks and additions, these separate ciphertexts are combined into the matrix-vector product between an RP matrix and a vector as shown in Algorithm 5.

Algorithm 6: `RP-MatVecMult`

Input: Ciphertext A, Ciphertext b
Output: Ciphertext result

vector$\langle$int$\rangle$ zero $= [0, 0, \ldots, 0]$
for $i = 0$ **to** $A.\texttt{size}$ **do**
  zero$[i] = 1$
  result $+= \texttt{DotProd}(A(i), b) *$ zero
end

Row-Expanded-Packed (REP) Matrices.

This method of encoding a matrix is similar to RP matrices, except that each entry is repeated q times for some integer q that is a power of two. As with RP matrices, REP matrices are represented by vectors of ciphertexts. For this encoding, we only consider matrix products between CP and REP matrices.

First, we define a function, `Duplicate` in Algorithm 3. Suppose that a ciphertext has k filled slots out of n, `Duplicate` fills the remaining slots with repetitions of the k slots. This can be realized using simple rotations and additions.

Algorithm 7: `Duplicate`

Input: Ciphertext ν
Output: Ciphertext result

for $i = 0$ **to** $\log_2(\texttt{colsize}/\lceil k \rceil_{PO2}) - 1$ **do**
  result $+= \texttt{Rotate}(C, \lceil k \rceil_{\mathrm{PO2}})$
end

To compute matrix products between CP and REP matrices, we first apply `Duplicate` the columns of the CP matrix. Then, we multiply each column in the CP matrix with its corresponding row in the REP matrix. Finally, we sum all the ciphertexts and obtain the product of the matrices in a CCP matrix. This is shown in Algorithm 3.

Algorithm 8: `CP-REP-MatMult`

Input: Ciphertext A, Ciphertext B
Output: Ciphertext result

```
for i = 0 to A.size do
    result += Duplicate(A(i)) * B(i)
end
```

Logistic Regression with Homomorphic Encryption

The first step in the GWAS algorithm is to solve a logistic model for its weights $\boldsymbol{\beta}$. There are several solutions [11, 12, 13, 14, 15] that solve a logistic model with HE, given that it was one of the challenges in the iDASH 2017 competition.

Logistic Regression.

Logistic regression estimates the parameters of a binary logistic model. Such models are used to predict the probability of an event occurring given some input features. These models assume that the logarithm of the odds ratio (log-odds) is a linear combination of the input features.

Let p denote the probability of an event occurring. The assumption above can be written as

$$\log\left(\frac{p}{1-p}\right) = \beta_0 + \beta_1 x_1 + \cdots + \beta_d x_d. \tag{1}$$

Rearranging Equation (1), we get

$$p(\mathbf{x}, \boldsymbol{\beta}) = \frac{1}{1 + e^{-\boldsymbol{\beta}^\intercal \mathbf{x}}} \tag{2}$$

where $\boldsymbol{\beta} = (\beta_0, \beta_1, \ldots, \beta_d)$ and $\mathbf{x} = (1, x_1, \ldots, x_d)$. This is known as the sigmoid function.

Logistic regression estimates the regression coefficients $\boldsymbol{\beta}$ using maximum likelihood estimation (MLE). This likelihood is given as

$$\begin{aligned} L(\mathbf{X}, \boldsymbol{\beta}) &= \prod_{i=1}^{n} P(y_i | \mathbf{x}_i) \\ &= \prod_{i=1}^{n} p(\mathbf{x}_i, \boldsymbol{\beta})^{y_i} (1 - p(\mathbf{x}_i, \boldsymbol{\beta}))^{1-y_i}. \end{aligned} \tag{3}$$

where $\mathbf{x}_i$ denotes the rows of the covariates matrix $\mathbf{X}$. Often, MLE is performed with the log-likelihood

$$\begin{aligned}\ell(\mathbf{X}, \boldsymbol{\beta}) = &\sum_{i=1}^{n} y_i \log(p(\mathbf{x}_i, \boldsymbol{\beta})) \\ &+ \sum_{i=1}^{n} (1 - y_i) \log(1 - p(\mathbf{x}_i, \boldsymbol{\beta}))\end{aligned} \tag{4}$$

$$\begin{aligned}= &\sum_{i=1}^{n} y_i \log\left(\frac{p(\mathbf{x}_i, \boldsymbol{\beta})}{(1 - p(\mathbf{x}_i, \boldsymbol{\beta})}\right) \\ &+ \sum_{i=1}^{n} \log\left(\frac{1}{e^{\boldsymbol{\beta}^\intercal \mathbf{x}} + 1}\right)\end{aligned} \tag{5}$$

$$= \sum_{i=1}^{n} y_i(\boldsymbol{\beta}^\intercal \mathbf{x}_i) - \sum_{i=1}^{n} \log(e^{\boldsymbol{\beta}^\intercal \mathbf{x}} + 1) \tag{6}$$

Maximizing Equation (6) requires an iterative process. Our implementation in solving the logistic model applies the Newton-Raphson method. This is because the Newton-Raphson method is known to converge quadratically and we wish to solve the model with as little iterations as possible.

The Newton-Raphson method iterates over the following equation

$$\beta^{(t+1)} = \boldsymbol{\beta}^{(t)} - \mathbf{H}^{-1}(\boldsymbol{\beta}^{(t)})\mathbf{g}(\boldsymbol{\beta}^{(t)}) \tag{7}$$

where $\mathbf{g}$ and $\mathbf{H}$ are given as

$$\mathbf{g}(\boldsymbol{\beta}) = \mathbf{X}^\intercal(\mathbf{y} - \mathbf{p}(\boldsymbol{\beta})), \tag{8}$$

$$\mathbf{H}(\boldsymbol{\beta}) = \mathbf{X}^\intercal(\mathbf{W}(\boldsymbol{\beta}))\mathbf{X}. \tag{9}$$

However, there are two non-HE friendly aspects in this algorithm. Firstly, for each iteration, $\mathbf{H}$ is re-computed with the iteration's $\boldsymbol{\beta}$. This is computationally expensive with homomorphic encryption. Secondly, the sigmoid function Equation (2) contains the exponential function, e^x which is not natively supported by HE schemes. Hence, we approximate the Hessian matrix and the sigmoid function in our implementation.

Hessian Matrix Approximation.

We use an approximation for all Hessian matrices as suggested by Böhning and Lindsay [16]. They proposed using

$$\tilde{\mathbf{H}} = \frac{1}{4}\mathbf{X}^\intercal\mathbf{X} \tag{10}$$

as a lower bound approximation for all Hessian matrices in solving a logistic model with the Newton-Raphson method. This approximation is also used by Xie *et. al.* [17] in their distributed privacy preserving logistic regression. We chose to precompute $(\mathbf{X}^\intercal\mathbf{X})^{-1}$ and encrypt it as an input to the GWAS algorithm.

Sigmoid Function Approximation.

We use the approximation from Kim *et. al.* [12] who proposed polynomials of degree $3, 5, 7$ as approximations of the sigmoid function. We chose the polynomial of degree 7:

$$\sigma_7(x) = 0.5 - 1.73496\left(\frac{x}{8}\right) + 4.19407\left(\frac{x}{8}\right)^3 - 5.43402\left(\frac{x}{8}\right)^5 + 2.50739\left(\frac{x}{8}\right)^7. \quad (11)$$

Our Algorithm.

We described our algorithm in Algorithm 7. We encrypt $\mathbf{X}$ and $(\mathbf{X}^\intercal\mathbf{X})^{-1}$ as CP matrices and $\mathbf{y}$ as a ciphertext. We initialize $\boldsymbol{\beta}$ in a ciphertext by encrypting a vector of zeros. We first compute $\mathbf{X}\boldsymbol{\beta}$ with Algorithm 4 and apply Equation (11) on to each slot in the ciphertext. Note here that $\mathbf{X}\boldsymbol{\beta}$ is now a vector and is represented by one ciphertext. Instead of encrypting $\mathbf{X}^\intercal$, we treat $\mathbf{X}$ as $\mathbf{X}^\intercal$ encrypted as a RP matrix. We thus invoke RP matrix vector multiplication, Algorithm 5 with $\mathbf{X}^\intercal$ and $(\mathbf{y} - \mathbf{p})$. Finally, $\boldsymbol{\beta}$ is updated with Equation (7).

Algorithm 9: `Homomorphic LogReg`

Input: $\mathbf{X}$, $\mathbf{y}$, $(\mathbf{X}^\intercal\mathbf{X})^{-1}$
Output: $\boldsymbol{\beta}$

for $i = 1$ **to** κ **do**
  $\mathbf{p} \leftarrow \sigma_7(\mathbf{X}\boldsymbol{\beta})$.
  $g \leftarrow \mathbf{X}^\intercal(\mathbf{y} - \mathbf{p})$
  $\tilde{\mathbf{H}}^{-1} \leftarrow 4\,(\mathbf{X}^\intercal\mathbf{X})^{-1}$
  $\boldsymbol{\beta}_{new} \leftarrow \boldsymbol{\beta} - \tilde{\mathbf{H}}^{-1}g$
  $\boldsymbol{\beta} = \boldsymbol{\beta}_{new}$
end

Semi-Parallel GWAS with Homomorphic Encryption

The semi-parallel GWAS algorithm proposed by Sikorska *et. al.* [6] rearranges linear model computations and leverages fast matrix operations to achieve some parallelization and thus better performance. A logistic model is first solved with the covariates matrix. Let $\mathbf{z}$ be a temporary variable

$$\mathbf{z} = \mathbf{X}\boldsymbol{\beta} + \mathbf{W}^{-1}(\mathbf{y} - \mathbf{p}) \quad (12)$$

where $\boldsymbol{\beta}$ is the weights of the logistic model, $\mathbf{y}$ is the response vector and $\mathbf{p}$ is the vector of probabilities from evaluating the sigmoid function Equation (2) with $\boldsymbol{\beta}$.

The SNP matrix $\mathbf{S}$ is then orthogonalized with

$$\mathbf{S}^* = \mathbf{S} - \mathbf{X}\left(\mathbf{X}^\intercal\mathbf{W}\mathbf{X}\right)^{-1}\mathbf{X}^\intercal\mathbf{W}\mathbf{S} \tag{13}$$

and $\mathbf{z}$ is orthogonalized with

$$\mathbf{z}^* = \mathbf{z} - \mathbf{X}\left(\mathbf{X}^\intercal\mathbf{W}\mathbf{X}\right)^{-1}\mathbf{X}^\intercal\mathbf{W}\mathbf{z}. \tag{14}$$

The estimated SNP effect $\mathbf{s}$ can then be computed with

$$\mathbf{b} = \frac{(\mathbf{W}\mathbf{z}^*)^\intercal \cdot \mathbf{S}^*}{\texttt{colsum}(\mathbf{W}(\mathbf{S}^*)^2)} \tag{15}$$

and the standard error can be computed with

$$\mathbf{err} = \frac{1}{\texttt{colsum}(\mathbf{W}(\mathbf{S}^*)^2)}. \tag{16}$$

Division here denotes element-wise division between the vectors $(\mathbf{W}\mathbf{z})^\intercal \cdot \mathbf{S}^*$ and $colsum(\mathbf{W}(\mathbf{S}^*)^2)$.

The main obstacle for HE with the semi-parallel GWAS algorithm is matrix inversion. General matrix inversion is computationally expensive and inefficient in HE. This is mainly because integer division, which is used frequently in matrix inversion, cannot be efficiently implemented in HE. There are two instances where matrix inversion has to be computed. The first occurs in Equation (12) and the second occurs in the orthogonal transformations Equations (13) and (14). In the following paragraphs, we will describe our method for implementing the semi-parallel GWAS algorithm with HE. We will also describe some optimizations that reduce memory consumption and accelerate computations to qualify within the competition requirements.

Inverse of W.

We exploit the nature of $\mathbf{W}$ to compute its inverse with the Newton-Raphson method in HE. Recall that $\mathbf{W}$ is a n by n diagonal matrix whose entries are $p_i(1-p_i)$ for $i = 1, \ldots, n$. Firstly, we represent the diagonal matrix $\mathbf{W}$ by a vector $\mathbf{w}$ containing the diagonal entries to reduce storage and computational complexity. Secondly, the inverse of a diagonal matrix is can be obtained by inverting the entries along the main diagonal. This means that $\mathbf{W}^{-1}$ can be computed by inverting the slots of $\mathbf{W}$. The entries of $\mathbf{w}$ are given as $p_i(1-p_i)$, where $p \in (0, 1)$. We claim an upper bound of 0.25 on the slots of $\mathbf{w}$. We used this information to set a good initial guess of 3 in the Newton-Raphson method. This would reduce the number of iterations needed to obtain an accurate inverse. We describe this algorithm in Algorithm 5

Modification of Orthogonal Transformations.

We propose modifications to Equations (13) and (14) as $\left(\mathbf{X}^\intercal\mathbf{W}\mathbf{X}\right)^{-1}$ is too expensive to be computed in the encrypted domain.

Algorithm 10: `inverseSlots`

Input: $\mathbf{w}$
Output: $\mathbf{w}^{-1}$

guess $= [3, 3, \cdots, 3]$
for $i = 1$ **to** 3 **do**
  $\mathbf{w}^{-1}$ = guess (2 - $\mathbf{w}$ * guess)
  guess $= \mathbf{w}^{-1}$
end

Let $\mathbf{M}$ be defined as

$$\mathbf{M} = I - \mathbf{X}\left(\mathbf{X}^\intercal\mathbf{X}\right)^{-1}\mathbf{X}^\intercal. \tag{17}$$

We proposed a modification, inspired by the Hessian approximation in Equation (10), to the orthogonal transformation of $\mathbf{S}$ with

$$\begin{aligned}\mathbf{S}' &= \mathbf{M}\mathbf{S} \\ &= \mathbf{S} - \mathbf{X}\left(\mathbf{X}^\intercal\mathbf{X}\right)^{-1}\mathbf{X}^\intercal\mathbf{S}\end{aligned} \tag{18}$$

and $\mathbf{z}$ with

$$\begin{aligned}\mathbf{z}' &= \mathbf{M}\mathbf{z} \\ &= \mathbf{z} - \mathbf{X}\left(\mathbf{X}^\intercal\mathbf{X}\right)^{-1}\mathbf{X}^\intercal\mathbf{z}.\end{aligned} \tag{19}$$

The estimated SNP effect is now computed with

$$\mathbf{b}' = \frac{(\mathbf{W}\mathbf{z}')^\intercal \cdot \mathbf{S}'}{\texttt{colSum}(\mathbf{W}(\mathbf{S}')^2)}. \tag{20}$$

and the standard error is

$$\mathbf{err}' = \frac{1}{\texttt{colsum}(\mathbf{W}(\mathbf{S}')^2)}. \tag{21}$$

Complex Space of CKKS ciphertext

For our first optimization, we exploit the scheme's native support for complex numbers to pack two SNPs into a single complex number, putting one SNP in the real part and another in the complex part. This allows us to fit twice as many SNPs in a single ciphertext and cut the runtime by half.

However, $(\mathbf{S}'_i)^2$ in Equation (20) is more difficult to compute with this packing method. Simply squaring the ciphertext does not yield the correct output as slots now contain complex numbers; for some complex number $z = x + yi$,

$$(x + yi)^2 = (x^2 - y^2) + 2xyi. \tag{22}$$

Instead, we consider multiplying z by its complex conjugate $\overline{z} = x - yi$. We have

$$(x + yi)(x - yi) = x^2 + y^2 \tag{23}$$

Extracting the real parts of Equation (22) and Equation (23), we get

$$x^2 = \frac{Re(z\overline{z} + z^2)}{2} \qquad y^2 = \frac{Re(z\overline{z} - z^2)}{2} \tag{24}$$

Recall that $\mathbf{S}'_i$ is a CCP matrix which is represented by one ciphertext with each slot holding one complex numbers encoding two SNPs. Thus, we compute $\mathbf{S}'_i\mathbf{S}'_i$ and $\mathbf{S}'_i\overline{\mathbf{S}'_i}$. We assign

$$\mathbf{S}_{even} = \frac{\mathbf{S}'_i\overline{\mathbf{S}'_i} + \mathbf{S}'_i\mathbf{S}'_i}{2} \tag{25}$$

and

$$\mathbf{S}_{odd} = \frac{\mathbf{S}'_i\overline{\mathbf{S}'_i} - \mathbf{S}'_i\mathbf{S}'_i}{2} \tag{26}$$

Optimizations with HEAAN

There are two optimizations that we used with the HEAAN library to reduce the parameters needed and to improve runtime.

For the first optimization, we rescale the ciphertext by a value that is smaller than p after every plaintext multiplication. This means each plaintext multiplication is now "cheaper" than a ciphertext multiplication and hence the value of L when initialized can be lowered.

The second optimization would be to perform only power-of-two rotations. A rotation by τ slots is a composition of power-of-two rotations in the HEAAN library. The required power-of-two rotations are the 1s of the binary decomposition of τ. Thus, it would be more efficient if we only perform rotations by a power-of-two. We illustrate this with an example. A rotation by 245 slots would require 6 power-of-two rotations as the binary decomposition of 245 is 11110101. A rotation by 256 slots would require 1 power-of-two rotations as the binary decomposition of 256 is 100000000. This reduces the number of rotations in our implementation.

Batching SNPs

As $\mathbf{S}$ is too large to be stored in memory when encrypted, we propose to divide $\mathbf{S}$ column-wise and process batches of SNPs. We show how to compute the maximum number of SNPs that can fit within a batch. Let τ be the number of SNPs in a batch. Consider $\mathbf{MS}$, a matrix product between a n by n and a n by τ matrix. By Algorithm 3, the result is a CCP matrix, whose ciphertext has to have enough slots for $n \times \tau$ elements. For efficiency as described in the previous section, we round the size of each column to the nearest power-of-two and pad the columns with zeroes. Together with the complex space of the

HEAAN ciphertext, the maximum number of SNPs that can be processed as a batch is given as

$$\tau = \frac{2^{N-1}}{2 \times \lceil n \rceil_{\text{PO2}}} \tag{27}$$

Smart Cache Module

We consider the largest matrix in our implementation, $\mathbf{M}$ which is a n by n matrix. This means that the ciphertext would need to have at least $\lceil n \rceil_{\text{PO2}}{}^2$ slots. Consequentially, $\log N$ is at least $2 \times \lceil n \rceil_{\text{PO2}}{}^2$. This results in a large set of parameters for the HE scheme which translate to a large amount of memory usage. Furthermore, the virtual machine (VM) that the iDASH organizers provide only has 16GB RAM. As a result, we choose to move ciphertexts to the hard disk when they are not used for computations.

We designed a cache module that exploits the vectorized ciphertext structure of encrypted matrices. Multiple threads are used to write ciphertexts into files on the hard disk and read ciphertexts from the hard disk to memory. A ciphertext will be pre-fetched into memory before it is needed for computation, replacing a ciphertext that is no longer needed.

Our Algorithm

We give a detailed walkthrough of our modified semi-parallel GWAS algorithm in Algorithm 23

First, we perform logistic regression with $\mathbf{X}$, $\mathbf{y}$ and $(\mathbf{X}^\intercal\mathbf{X})^{-1}$ as described in Algorithm 7. We use $\boldsymbol{\beta}$ from logistic regression, together with $\mathbf{p}$ from the previous iteration to compute $\mathbf{w}$ and $\mathbf{z}$.

Next, compute the inverse of the slots elements in $\mathbf{w}$ with `inverseSlots`. Note that $\mathbf{W}^{-1}(\mathbf{y} - \mathbf{p})$ is equivalent to multiplying the ciphertexts $\mathbf{w}^{-1}$ and $(\mathbf{y} - \mathbf{p})$. We then compute $\mathbf{z}'$ as given in Equation (19). At this point, we have $\mathbf{z}'$ and $\mathbf{w}$ which are both vectors, stored in a ciphertext each.

We construct a temporary variable $\mathbf{M} = \mathbf{Id} - \mathbf{X}\,(\mathbf{X}^\intercal\mathbf{X})^{-1}$ which is a CP matrix to facilitate computations. Here, we choose to encrypt $\mathbf{X}^\intercal$ as a REP matrix. The reason for encrypting differently is because multiplying a CP matrix by a RP matrix requires the RP matrix to be first converted into REP form. This process is very inefficient homomorphically and hence we decided to encrypt it directly as a REP matrix. Thus, the product of $\mathbf{X}\,(\mathbf{X}^\intercal\mathbf{X})^{-1}$ with $\mathbf{X}^\intercal$ is a `CP-REP-MatMult` as shown in Algorithm 3. We then convert $\mathbf{M}$ into a CP matrix to compute $\mathbf{MS}$.

As described earlier, we iterate over partial blocks of the SNP matrix, $\mathbf{S}_i$, divided column-wise. Next, compute its orthogonal transformation $\mathbf{S}'_i$. We remind the reader here again that $\mathbf{MS}_i$ is computed with `CP-REP-MatMult` which produces a CCP matrix, $\mathbf{S}'$. We compute, separately, the numerator, **numerator**, and denominator, **denominator**, of Equation (20) for each $\mathbf{S}'_i$.

For **numerator**, we multiply $\mathbf{w}^{-1}$ and $(\mathbf{z}')$ slots-wise and duplicate the slots for as many columns in the CCP matrix $\mathbf{S}'$. The vector-matrix product is now

Algorithm 11: Semi Parallel GWAS

Input: $\mathbf{X}$, $\boldsymbol{\beta}$, $\mathbf{y}$, $\mathbf{p}$, $(\mathbf{X}^\intercal\mathbf{X})^{-1}$, $\mathbf{X}^\intercal$, $\mathbf{S}$
Output: $\mathbf{b}'$

$\boldsymbol{\beta}$, $\mathbf{p} \leftarrow$ `HomLogisticRegression` $\left(\mathbf{X}, \mathbf{y}, (\mathbf{X}^\intercal\mathbf{X})^{-1}\right)$
$\mathbf{w} \leftarrow \mathbf{p}(\mathbf{1} - \mathbf{p})$
$\mathbf{w}^{-1} \leftarrow$ `inverseSlots`$(\mathbf{w})$
$\mathbf{z} \leftarrow \mathbf{X}\boldsymbol{\beta}^{(k)} + \mathbf{w}^{-1} * (\mathbf{y} - \mathbf{p}^{(k-1)})$.
`/* superscripts for` $\boldsymbol{\beta}$ `and` $\mathbf{p}$ `indicate the iteration number in Algorithm 7 */`
$\mathbf{M} \leftarrow \mathbf{Id} - \mathbf{X}\left(\mathbf{X}^\intercal\mathbf{X}\right)^{-1}\mathbf{X}^\intercal$
$\mathbf{z}' \leftarrow \mathbf{Mz}$
vector⟨double⟩ numerator, denominator
for $batch = 1$ **to** κ **do**
  $\mathbf{S}'_i \leftarrow \mathbf{MS}_i$
  encNumerator $\leftarrow (\mathbf{w} * \mathbf{z}')^\intercal \cdot \mathbf{S}'_i$
  numerator.`insert` (`Decrypt` (encNumerator))
  WSS $\leftarrow (\mathbf{WS}'_i) \cdot \mathbf{S}'_i$
  WSC $\leftarrow (\mathbf{WS}'_i) \cdot \overline{\mathbf{S}'_i}$
  denEven $\leftarrow 0.5 * (\text{WSC} + \text{WSS})$
  denOdd $\leftarrow 0.5 * (\text{WSC} - \text{WSS})$
  for $i = 0$ **to** denEven.`size` **do**
    denominator.`insert` (`Decrypt` (denEven(i)))
    denominator.`insert` (`Decrypt` (denOdd(i)))
  end
end
for $i = 1$ **to** k **do**
  $\mathbf{b}'_i =$ numerator(i)/denominator(i)
end

Table 1: Depth of Homomorphic Operations

Homomorphic Operation	No. Successive Operations
Plaintext Multiplication*	73
Ciphertext Multiplication†	82
Ciphertext Rotation	544

* `Rescale` with $\log p = 10$.
† `Rescale` with $\log p = 45$.

redefined as a ciphertext multiplication, followed by calling `colSum` over n slots.

For **denominator**, the computation is similar. After squaring the slots of the CCP matrix $\mathbf{S}'$, we duplicate $\mathbf{w}$ and perform a slot-wise multiplication. `colSum` of the resulting CCP matrix is exactly the second part of the vector-matrix product for **numerator** - the accumulation sum over every n slots.

We wish to highlight here that as stated in Q15 FAQ for the competition, it is acceptable to return **numerator** and **denominator** separately [18]. As such, we decrypt and concatenate all **numerator**s and **denominator**s respectively instead of performing a costly inversion of **denominator**. Finally, we divide the two vectors element-wise to obtain the estimated SNP effect, $\mathbf{b}'$.

Results

We used the provided dataset of 245 users with 4 covariates and 10643 SNPs.

The HE library used is the HEAAN library [19], commit id *da3b98*. The HE parameters used are $\log N = 17$, $\log L = 2440$ and $\log p = 45$. We observed that the HEAAN context based on these parameters utilizes about 3.5GB on both machines. The context can be thought of as the base memory needed for HE computations. Furthermore, we run **Rescale** on the output with $p = 45$ for ciphertext multiplications and $p = 10$ for plaintext multipications to control noise growth. This gives us a security level of about 93 bits based on the LWE estimator provided by Albrecht *et. al.* [20].

As described earlier, we require at least $\lceil n \rceil_{\text{PO2}}{}^2$ slots, where $n = 245$. We chose the minimum number of slots needed, 2^{16} slots and set $\log N$ to be 17. We are able to process a total of $\tau = 512$ SNPs in each batch, based on Equation (27). This gives us a total of $\lceil 10643/512 \rceil = 21$ batches. We set $\kappa = 3$ for the number of iterations in `HomLogisticRegression`.

We have tabulated the number of sequential homomorphic computations of our modified GWAS algorithm in Table 1. These numbers represent the circuit depth of the GWAS algorithm. We find that a comparison of the number of these computations is a better measure of evaluating a HE program, independent of HE library used.

We report the time taken and memory consumed on two servers: the VM provided by the iDASH organizers and our server.

Table 2: Time Taken and Memory Consumption with iDASH server using HEAAN

Process	Time Taken (min)	Memory (GB)
Encryption	0.79	0.802628
Computations	717.20	3.98849
Decryption	0.32	0.063
Total	718.31	4.854118

Number of cores: 4
Context generated: 3.5GB

Table 3: Time Taken and Memory Consumption with our server using HEAAN

Process	Time Taken (min)	Memory (GB)
Encryption	0.404	0.886795
Computations	203.42	24.1119
Decryption	0.30	0.063
Total	204.124	25.061695

Number of cores: 22
Context generated: 3.5GB

The machine provided by the iDASH organizers is an Amazon T2 Xlarge or equivalent VM, which has 4 vCPU, 16GB memory, disk size around 200GB [18]. The results are shown in Table 2.

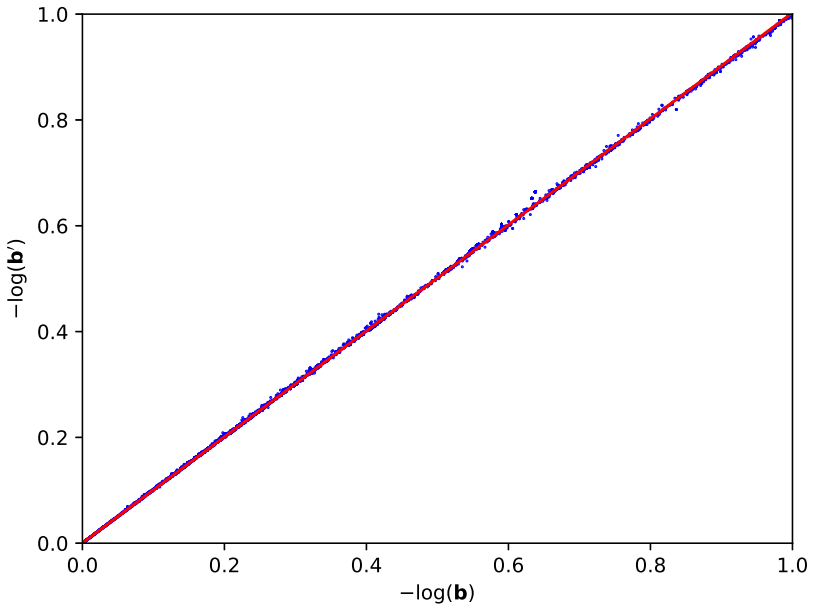
For our server, the CPU model used is Intel Xeon Platinum 8170 CPU at 2.10GHz with 26 cores and the OS used is Arch Linux. The results are shown in Table 3.

We evaluated the accuracy of our results with two methods. The first method compares the vectors $\mathbf{b}$ and $\mathbf{b}'$, counting the number of entries that are not equal. However, since the CKKS scheme introduces some error upon decrypting, we are unable to get any identical entries. Instead, we opt to count the number of p-values for which our solution differs from the original algorithm by more than some error, e. This is shown in Table 4

Table 4: HEAAN Accuracy

Error e	No. of Different Entries	HEAAN Accuracy (%)
0.1	0	100
0.01	168	98.42
0.005	645	93.94

Figure 1: HEAAN Implementation Scatter Plot



The second method would be to plot a scatter diagram whose x-axis represent $\mathbf{b}$ and y-axis represent $\mathbf{b}'$. Ideally, if $\mathbf{b} = \mathbf{b}'$, the best fit line of the scatter plot should be $y = x$. We compute the line of best fit with the `numpy.polyfit` function from python and compared against the line $y = x$. Our HEAAN based solution gives the line $y = 1.002x + 0.0005317$. The scatter plot is given in Figure 1.

We port our implementation to the SEAL library [21] which recently released a version of the CKKS scheme that does not require the 2^{2L} modulus. We implemented this with 22 cores on our machine. The parameters used are $\log N = 17$, $\log L = 1680$ and $\log p = 50$. The context generated in this instance is approximately 73.4GB. The results of this implementation is given in Table 5.

The accuracy of the SEAL implementation based on the first method is tabulated in Table 6.

Our SEAL based solution gives the line $y = 1.017x + 0.007565$. The scatter plot for the results is given in Figure 2.

Table 5: Time Taken and Memory Consumption with our server using SEAL

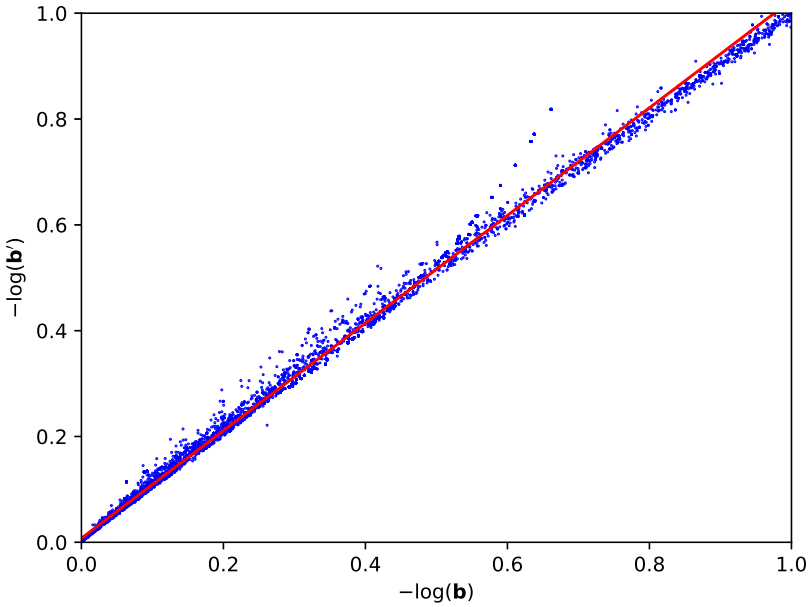
Process	Time Taken (min)	Memory (GB)
Encryption	0.20	1.60404
Computations	24.70	38.4843
Decryption	0.31	0.666103
Total	25.21	40.76

Number of cores: 22
Context generated: 73.4GB

Table 6: SEAL Accuracy

Error e	No. of Different Entries	SEAL Accuracy (%)
0.1	127	98.81
0.01	4061	61.84
0.005	5940	44.19

Figure 2: SEAL Implementation Scatter Plot



Discussion

In our submission, we miscalculated the security level, assuming that it fit the 128-bit requirements while it was actually about 93 bits. This is due to the use of the modulus 2^{2L} for the evaluation key, which is a quirk of the HEAAN library [19].

There is also a limit of 256 subjects with our implementation, due to our desire to pack the entire test dataset into a single ciphertext. For a larger number of subjects (up to 512), the matrix $\mathbf{X}^\intercal$ will need at least 512 by 512 slots, which means that $\log N$ has to be at least 19.

We are aware of the limitations in HEAAN, namely the 2^{2L} modulus and slower homomorphic operations. However, it was the only publicly available HE library based on the CKKS scheme.

We can see that SEAL's implementation of the CKKS scheme is superior in terms of runtime. This is because SEAL implemented an RNS-variant of the CKKS, which improves the speed of the algorithm by almost 8 times. The security level of this implementation based on the LWE estimator is about 230 bits.

However, we are unable to execute our GWAS algorithm with SEAL using $\kappa = 3$. The set of parameters that supports the depth of the algorithm with $\kappa = 3$ appears to be too large and caused our server to run out of memory. Hence, for the implementation with SEAL, we reduced κ to 1. This reduces the depth of the algorithm and hence the parameters that were used. Consequentially, the accuracy of the results has decreased from 98.42% to 61.84%.

Conclusions

In this paper, we demonstrated an implementation of a semi-parallel GWAS algorithm for encrypted data. We employed suitable approximations in adapting the semi-parallel GWAS algorithm to be HE-friendly. Our solution shows that the model trained over encrypted data is comparable to one trained over unencrypted data. Memory constraints are shown to be of little concern with our implementation of a smart cache, which reduced memory consumption to fit within the limits imposed. This signifies another milestone for HE, showing that HE is mature enough to tackle more complex algorithms.

Acknowledgements

The authors would like to thank Ahmad Al Badawi for his constructive comments in writing the paper.

References

[1] Estrada, K., Abuseiris, A., Grosveld, F.G., Uitterlinden, A.G., Knoch, T.A., Rivadeneira, F.: Grimp: a web- and grid-based tool for high-speed analysis of large-scale genome-wide association using imputed data. Bioinformatics (2009)

[2] Lin, Z., Owen, A.B., Altman, R.B.: Genomic research and human subject privacy. Science **305**(5681), 183–183 (2004). doi:10.1126/science.1095019. http://science.sciencemag.org/content/305/5681/183.full.pdf

[3] Homer, N., Szelinger, S., Redman, M., Duggan, D., Tembe, W., Muehling, J., Pearson, J.V., Stephan, D.A., Nelson, S.F., Craig, D.W.: Resolving Individuals Contributing Trace Amounts of DNA to Highly Complex Mixtures Using High-Density SNP Genotyping Microarrays. `https://doi.org/10.1371/journal.pgen.1000167`

[4] Rousseau, D.: Biomedical research: Changing the common rule by david rousseau. PLOS Genetics (2017)

[5] iDASH Privacy & Security Workshop. `http://www.humangenomeprivacy.org`

[6] Sikorska, K., Lesaffre, E., Groenen, P.F., Eilers, P.H.: Gwas on your notebook: fast semi-parallel linear and logistic regression for genome-wide association studies. BMC Bioinformatics (2013)

[7] Rivest, R.L., Adleman, L., Dertouzos, M.L.: On data banks and privacy homomorphisms. Foundations of Secure Computation, Academia Press (1978)

[8] Gentry, C.: Fully homomorphic encryption using ideal lattices. In: 41st ACM Symposium on Theory of Computing, pp. 169–178. ACM Press, ??? (2009)

[9] Cheon, J.H., Kim, A., Kim, M., Song, Y.: Homomorphic Encryption for Arithmetic of Approximate Numbers. Cryptology ePrint Archive, Report 2016/421. `http://eprint.iacr.org/2016/421` (2016)

[10] Halevi, S., Shoup, V.: Algorithms in helib. In: Advances in Cryptology – CRYPTO 2014 (2014)

[11] Kim, M., Song, Y., Wang, S., Xia, Y., Jiang, X.: Secure Logistic Regression Based on Homomorphic Encryption: Design and Evaluation. Cryptology ePrint Archive, Report 2018/074. `https://eprint.iacr.org/2018/074` (2018)

[12] Kim, A., Song, Y., Kim, M., Lee, K., Cheon, J.H.: Logistic Regression Model Training based on the Approximate Homomorphic Encryption. Cryptology ePrint Archive, Report 2018/254. `https://eprint.iacr.org/2018/254` (2018)

[13] Chen, H., Gilad-Bachrach, R., Han, K., Huang, Z., Jalali, A., Laine, K., Lauter, K.: Logistic regression over encrypted data from fully homomorphic encryption. Cryptology ePrint Archive, Report 2018/462. https://eprint.iacr.org/2018/462 (2018)

[14] Crawford, J.L.H., Gentry, C., Halevi, S., Platt, D., Shoup, V.: Doing Real Work with FHE: The Case of Logistic Regression. Cryptology ePrint Archive, Report 2018/202. https://eprint.iacr.org/2018/202 (2018)

[15] Han, K., Hong, S., Cheon, J.H., Park, D.: Efficient Logistic Regression on Large Encrypted Data. Cryptology ePrint Archive, Report 2018/662. https://eprint.iacr.org/2018/662 (2018)

[16] Böhning, D., Lindsay, B.G.: Monotonicity of quadratic-approximation algorithms. Annals of the Institute of Statistical Mathematics (1988)

[17] Xie, W., Wang, Y., Boker, S.M., Brown, D.E.: Privlogit: Efficient privacy-preserving logistic regression by tailoring numerical optimizers. CoRR (2016)

[18] FAQ for iDASH Privacy Protection Competition. https://docs.google.com/document/d/1sVq413MvMrtJhb61sjSqxchBZyt7bS4khBKXN0y0xxc/edit

[19] Cheon, J.H., Kim, A., Kim, M., Song, Y.: HEAAN. GitHub (2018)

[20] Albrecht, M.R., Player, R., Scott, S.: On the concrete hardness of Learning with Errors. Cryptology ePrint Archive, Report 2015/046. https://eprint.iacr.org/2015/046 (2015)

[21] Simple Encrypted Arithmetic Library (release 3.1.0). https://github.com/Microsoft/SEAL. Microsoft Research, Redmond, WA. (2018)