Blatt *et al. BMC Medical Genomics* 2020, **13**(Suppl 7):83
https://doi.org/10.1186/s12920-020-0719-9

BMC Medical Genomics

RESEARCH Open Access

Optimized homomorphic encryption solution for secure genome-wide association studies

Marcelo Blatt[1], Alexander Gusev[1,2], Yuriy Polyakov[1*], Kurt Rohloff[1] and Vinod Vaikuntanathan[1]

From 7th iDASH Privacy and Security Workshop 2018
San Diego, CA, USA. 15 October 2018

Abstract

Background: Genome-Wide Association Studies (GWAS) refer to observational studies of a genome-wide set of genetic variants across many individuals to see if any genetic variants are associated with a certain trait. A typical GWAS analysis of a disease phenotype involves iterative logistic regression of a case/control phenotype on a single-neuclotide polymorphism (SNP) with quantitative covariates. GWAS have been a highly successful approach for identifying genetic-variant associations with many poorly-understood diseases. However, a major limitation of GWAS is the dependence on individual-level genotype/phenotype data and the corresponding privacy concerns.

Methods: We present a solution for secure GWAS using homomorphic encryption (HE) that keeps all individual data encrypted throughout the association study. Our solution is based on an optimized semi-parallel GWAS compute model, a new Residue-Number-System (RNS) variant of the Cheon-Kim-Kim-Song (CKKS) HE scheme, novel techniques to switch between data encodings, and more than a dozen crypto-engineering optimizations.

Results: Our prototype can perform the full GWAS computation for 1,000 individuals, 131,071 SNPs, and 3 covariates in about 10 minutes on a modern server computing node (with 28 cores). Our solution for a smaller dataset was awarded co-first place in iDASH'18 Track 2: "Secure Parallel Genome Wide Association Studies using HE".

Conclusions: Many of the HE optimizations presented in our paper are general-purpose, and can be used in solving challenging problems with large datasets in other application domains.

Keywords: Cryptography, Homomorphic encryption, Genome-wide association studies

Background

Genome-Wide Association Studies (GWAS) refer to observational studies of a genome-wide set of genetic variants across many individuals to see if any genetic variants are associated with a certain trait. When applied to human data, GWAS typically focus on associations between single-nucleotide polymorhisms (SNPs) and a quantitative or dichotomous disease outcome, as well as a number of quantitative covariates. However, the reliance on full genotype and phenotype data across thousands of samples raises major privacy concerns for GWAS, and has limited their applicability.

Recent work has focused on secure multi-party computation algorithms to facilitate privacy-preserving GWAS, but this approach requires resource-heavy, continuous interactions between users which is impractical for GWAS studies that are aggregated over months or years. To motivate the cryptographic community, the iDASH'18 Organizing Committee ran a special competition track "Secure

*Correspondence: ypolyakov@duality.cloud
[1]Duality Technologies, Inc., Newark NJ, USA
Full list of author information is available at the end of the article

Parallel Genome Wide Association Studies using Homomorphic Encryption (HE)" to advance the state of the art in GWAS using HE, which is a non-interactive approach to secure computing.

This paper presents our HE-based solution to GWAS. Our solution is based on an optimized GWAS compute model, a new Residue-Number-System (RNS) variant of the Cheon-Kim-Kim-Song (CKKS) HE scheme, novel techniques to switch between data encodings, and more than a dozen crypto-engineering optimizations. The solution can perform the full GWAS computation for 1000 individuals, 131,071 SNPs, and 3 covariates in about 10 minutes on a modern server computing node (with 28 cores).

Related work

Several other RNS variants of the CKKS HE scheme were independently proposed in 2018. These include the work by Cheon et al. [1], the implementation in Microsoft SEAL 3.0 (released in October 2018), and the variants developed by other teams who submitted their GWAS solutions to the iDASH'18 competition, including UCSD [2] and IBM Research.

Methods

Semi-parallel approach of Sikorska et al. [3]

Logistic regression is widely used to model binary response data in GWAS. For instance, it can be used to examine the relationship between disease status (control versus real cases) with respect to phenotypes (age, weight, height, etc.) and genotypes (such as SNP variations). Let y_i denote the disease status for the i^{th} individual in a sample of size N ($y_i = 1$ if the individual is a disease case, and $y_i = 0$ otherwise), and $(\vec{x}_i', \vec{s}_i)$ be the corresponding predictor, where $\vec{x}_i' \in \mathbb{R}^K$ corresponds to the phenotypes and $\vec{s}_i \in \{0,1,2\}^M$ to the genotypes of individual i for a set of K phenotypes and M SNPs. The logistic regression model expresses the relationship between y_i and the predictor set $(\vec{x}_i', \vec{s}_i)$ in terms of the conditional probability $Pr\left(Y = y_i|\vec{x}_i', \vec{s}_i\right)$ of disease, as:

$$Pr\left(y_i|\vec{x}_i', \vec{s}_i\right) = \sigma\big((2y_i - 1)\left(\boldsymbol{\theta}_0' + \vec{x}_i' \cdot \boldsymbol{\theta}' + \vec{s}_i \cdot \boldsymbol{\beta}\right)\big),$$

where σ is the logistic function, $\sigma(x) = \frac{1}{1+\exp(-x)}$; $\theta_0' \in \mathbb{R}$, $\boldsymbol{\theta}' \in \mathbb{R}^K$ and $\boldsymbol{\beta} \in \mathbb{R}^M$ are the $K + M + 1$ parameters to be determined. For the sake of simplicity, we adopt the canonical notation, that is, $\boldsymbol{\theta} \equiv (\theta_0', \boldsymbol{\theta}') \in \mathbb{R}^{K+1}$ and $\vec{x}_i \equiv (1, \vec{x}_i') \in \mathbb{R}^{K+1}$ for $i = 1, \ldots, N$.

Assuming that the effect of each SNP is independent of each other, it is possible to formulate it as a set of M independent equations, i.e., decompose the computation into M independent logistic regression cases for $K + 1$ parameters. Sikorska et al. [3] proposed a "semi-parallel" approach to speed up the logistic regression in the above scenario. The goal is to avoid looping over each SNP by using a vectorized formulation, which includes optimized vector and matrix operations, that allows performing multiple identical actions over different data in a single operation.

The method relies on the assumption that the covariant parameters $\boldsymbol{\theta}$ are nearly the same for all SNPs. This assumption allows the reformulation of fitting N vectors in $\mathbb{R}^{K+1}$, followed by a one-step calculation for M SNPs at once. Therefore Sikorska's semi-parallel logistic regression consists of 2 stages:

1 Estimate the coefficients of the clinical covariates, $\boldsymbol{\theta} \in \mathbb{R}^{K+1}$;
2 For each of the M SNPs, estimate the corresponding coefficients $\hat{\boldsymbol{\beta}}$ and p-value $\vec{p} \in \mathbb{R}^M$.

The first stage, the estimation of $\boldsymbol{\theta}$, $\hat{\boldsymbol{\theta}}$, was widely addressed in the literature, in particular in the iDASH'17 secure genome analysis competition [4–8].

The second stage, the estimation of the SNP-coefficients $\hat{\boldsymbol{\beta}}$, approximates the optimization problem by a single Newton-Raphson iteration, leading to

$$\hat{\boldsymbol{\beta}} = \mathbf{H}^{-1}\,\mathbf{X}^\top\,\mathbf{W}\,\boldsymbol{\zeta},$$

where $\mathbf{X}$ is a matrix in $\mathbb{R}^{N\times(K+1)}$ whose rows are the vectors $\vec{x}_i$, $i = 1, \ldots, N$; $\mathbf{W} \in \mathbb{R}^{N\times N}$ is a diagonal matrix with $\omega_{ii} = \rho_i(1 - \rho_i)$ and $\rho_i = \sigma\left(\vec{x}_i \cdot \hat{\theta}^{(t)}\right)$ for $i = 1, \ldots, N$; $\mathbf{H} = \mathbf{X}^\top\,\mathbf{W}\,\mathbf{X}$ in $\mathbb{R}^{(K+1)\times(K+1)}$; $\zeta_i = \log\left(\frac{\rho_i}{1-\rho_i}\right) + \frac{y_i - \rho_i}{\omega_{ii}}$, $i = 1, \ldots, N$.

Finally, the z-value for each parameter β_j, for $j = 1, \ldots, M$, is given by $z_j = \frac{\hat{\beta}_j}{\epsilon_j}$, where $\epsilon_j = \sqrt{(\mathbf{C}^{-1})_{jj}}$ is the error associated to $\hat{\boldsymbol{\beta}}_j$ and $\mathbf{C} = \mathbf{S}^\top\mathbf{W}\left(\mathbf{S} - \mathbf{X}\mathbf{H}^{-1}\left(\mathbf{X}^\top\mathbf{W}\mathbf{S}\right)\right)$. A more compact expression of it is

$$z_j = \frac{1}{\det \mathbf{H}} \frac{\sum_i^n w_{ii}\zeta_i^* s_{ij}^*}{\sqrt{\sum_i^n w_{ii} s_{ij}^{*2}}} \quad j = 1, \ldots, m,$$

with

$$\boldsymbol{\zeta}^* = \det \mathbf{H}\,\boldsymbol{\zeta} - \mathbf{X}\mathbf{H}^\dagger\mathbf{X}^\top\mathbf{W}\,\boldsymbol{\zeta},$$
$$\mathbf{S}^* = \det \mathbf{H}\,\mathbf{S} - \mathbf{X}\mathbf{H}^\dagger\mathbf{X}^\top\mathbf{W}\,\mathbf{S}.$$

where $\mathbf{H}^\dagger$ denotes the adjoint of $\mathbf{H}$.

Our approximations

To optimize the efficiency of our HE solution, we introduced several approximations to the semi-parallel method of Sikorska et al. [3].

Logistic regression

We found that the gradient descent method is adequate for estimating $\boldsymbol{\theta}$. Starting from an initial $\boldsymbol{\theta}^{(0)}$, the gradient descent method at each iteration t updates the estimation of the regression parameters

$$\hat{\boldsymbol{\theta}}^{(t+1)} \leftarrow \hat{\boldsymbol{\theta}}^{(t)} + \alpha_t \mathbf{X}(y + \boldsymbol{\rho}),$$

where α_t is the learning rate at the t-th iteration. Our numerical experiments suggest that a single iteration of the gradient descent procedure with $\alpha_0 = 0.015$ and $\boldsymbol{\theta}^{(0)} = 0$ provides adequate accuracy. For simplicity, we denote α_0 as α in the rest of the paper.

Logistic function approximation

We used Chebyshev polynomials to approximate the logistic function [9]. From the analysis we performed, we found that a degree-1 approximation $\sigma(x) = 0.5 + 0.15625x$ provides results with sufficient accuracy. Please refer to "Analysis of our approximations" section for further details.

Approximation of $\boldsymbol{\zeta}$

In order to approximate $\boldsymbol{\zeta}$, we considered a Talyor series expansion around $p = \frac{1}{2}$:

$$\begin{aligned}\boldsymbol{\zeta}(p,y) \approx & (-2+4y) + \\ & (-8+16y)\left(p-\frac{1}{2}\right)^2 - \frac{32}{3}\left(p-\frac{1}{2}\right)^3 + \\ & (-32+64y)\left(p-\frac{1}{2}\right)^4 - \frac{256}{5}\left(p-\frac{1}{2}\right)^5 + \\ & (-128+256y)\left(p-\frac{1}{2}\right)^6 - \frac{1536}{7}\left(p-\frac{1}{2}\right)^7 + \\ & (-512+1024y)\left(p-\frac{1}{2}\right)^8 .\end{aligned}$$

Matrix inversion and division

Instead of calculating the inverse of the matrix $\mathbf{H}$, Cramer's rule was used: $\mathbf{H}^{-1} = \frac{\text{adj}(\mathbf{H})}{\det(\mathbf{H})}$, where $\text{adj}(\mathbf{H})$ is the *adjoint* of matrix $\mathbf{H}$ and $\det(\mathbf{H})$ is its *determinant*. As the division is an expensive operation, it was deferred to a later stage (after decryption).

p-value calculation

After computing the z-values on the server, the p-value computation is performed on the client as depicted in Algorithm 2.

Full procedure

The approximations described above were used to create an optimized procedure for the server computation (Algorithm 1). Note that line 2 of Algorithm 1 is the closed form for $\boldsymbol{\rho}$ that incorporates the parameter estimation of the logistic regression. Therefore $\hat{\boldsymbol{\theta}}$ does not appear explicitly in Algorithm 1.

The annotated encrypted procedure is presented in Algorithm 3. It will be referenced throughout the rest of this section.

CKKS scheme

Our solution is based an optimized variant of the Cheon-Kim-Kim-Song scheme [10]. We have developed a Double-Chinese Remainder Theorem (CRT), a.k.a, Residue Number System (RNS), variant of the original scheme. Our variant is based on the same security assumptions as the original scheme, but relies on native 64-bit integer arithmetic instead of multiprecision integer arithmetic for better performance and parallelization.

The original CKKS scheme is formulated for cyclotomic polynomial rings $\mathcal{R} = \mathbb{Z}[x]/\langle x^n + 1\rangle$, where n is a ring dimension that is a power of two (CKKS also supports general cyclotomic rings but they are typically less efficient). The current ciphertext modulus is typically defined as $Q_\ell = 2^\ell$, i.e., the scheme works with residue rings $\mathcal{R}_\ell = \mathcal{R}/Q_\ell\mathcal{R} = \mathbb{Z}_{2^\ell}[x]/\langle x^n + 1\rangle$. The algorithms are [10]:

- $\textsc{Setup}(1^\lambda)$. For an integer L that coresponds to the largest ciphertext modulus level, given the security parameter λ, output the ring dimension n. Set the small distributions χ_{key}, χ_{err}, and χ_{enc} over $\mathcal{R}$ for secret, error, and encryption, respectively.
- $\textsc{KeyGen}$. Sample a secret $s \leftarrow \chi_{key}$, a random $a \rightarrow R_L$, and error $e \leftarrow \chi_{err}$. Set the secret key $\mathbf{sk} \leftarrow (1, s)$ and public key $\mathbf{pk} \leftarrow (b, a) \in \mathcal{R}_L^2$, where $b \leftarrow -as + e\,(\bmod Q_L)$.
- $\textsc{KSGen}_{\mathbf{sk}}(s')$. For $s' \in \mathcal{R}$, sample a random $a' \leftarrow \mathcal{R}_{2\cdot L}$ and error $e' \leftarrow \chi_{err}$. Output the switching key as $\mathbf{swk} \leftarrow (b', a') \in \mathcal{R}_{2L}^2$, where $b' \leftarrow -a's' + e' + Q_L s'\,(\bmod Q_{2L})$. Set $\mathbf{evk} \leftarrow \textsc{KSGen}_{\mathbf{sk}}(s^2)$. Set $\mathbf{rk}^{(\kappa)} \leftarrow \textsc{KSGen}_{\mathbf{sk}}(s^{(\kappa)})$.

Algorithm 1 Approximated Semi-Parallel Procedure: Server Computations

```
1:  $\alpha \leftarrow 0.015$
2:  $\boldsymbol{\rho} \leftarrow 0.15625\alpha \cdot \mathbf{X}(\mathbf{X}^\top(\mathbf{y} - \mathbf{0.5})) + \mathbf{0.5}$
3:  $\vec{W} \leftarrow \boldsymbol{\rho} \star (\mathbf{1} - y)$
4:  $\boldsymbol{\zeta} \leftarrow \textsc{zExpand}(\boldsymbol{\rho}, \vec{y})$
5:  $\mathbf{H} \leftarrow (\mathbf{X}^\top\mathbf{W})\,\mathbf{X}$
6:  $\mathbf{B} \leftarrow \textsc{Adjoint}(\mathbf{H})$
7:  $d \leftarrow \textsc{Determinant}(\mathbf{H})$
8:  $\boldsymbol{\zeta}^* \leftarrow d \cdot \boldsymbol{\zeta} - (\mathbf{XH})\left((\mathbf{X}^\top\mathbf{W})\,\boldsymbol{\zeta}\right)$
9:  $\mathbf{S}^* \leftarrow d \cdot \mathbf{S} - \mathbf{X}\left((\mathbf{B}\,(\mathbf{X}^\top\mathbf{W}))\,\mathbf{S}\right)$
10: $\vec{z}^2_{\text{den}} \leftarrow (d \cdot d \cdot \mathbf{W})\,(\mathbf{S}^* \star \mathbf{S}^*)$
11: $\vec{z}_{\text{num}} \leftarrow (\mathbf{W}\boldsymbol{\zeta}^*)^\top\,\mathbf{S}^*$
```

$\star$ denotes element-wide multiplication

Algorithm 2 Approximated Semi-Parallel Procedure: Client Post-Processing

```
1: $\vec{z} \leftarrow \vec{z}_{\text{num}} \star/ \sqrt{\vec{z}^2_{\text{den}}}$
2: $\vec{p} \leftarrow 2\,\text{PNORM}\left(-\text{ABS}\left(\vec{z}\right)\right)$
                                   $\star/$ denotes element-wise division
```

- $\text{ENC}_{\mathbf{pk}}(m)$. For $m \in \mathcal{R}$, sample $v \leftarrow \chi_{enc}$ and $e_0, e_1 \leftarrow \chi_{err}$. Output $\mathbf{ct} \leftarrow v \cdot \mathbf{pk} + (m + e_0, e_1) \pmod{Q_L}$.
- $\text{DEC}_{\mathbf{sk}}(\mathbf{ct})$. For $\mathbf{ct} = (c_0, c_1) \in \mathcal{R}^2_\ell$, output $\tilde{m} = c_0 + c_1 \cdot s \pmod{Q_\ell}$.
- $\text{CADD}(\mathbf{ct}, c)$. For $\mathbf{ct} = (b, a) \in \mathcal{R}^2_\ell$ and $c \in \mathcal{R}$, output $\mathbf{ct}_{\text{cadd}} \leftarrow (b + c, a) \pmod{Q_\ell}$.
- $\text{ADD}\,(\mathbf{ct}_1, \mathbf{ct}_2)$. For $\mathbf{ct}_1, \mathbf{ct}_2 \in \mathcal{R}^2_\ell$, output $\mathbf{ct}_{\text{add}} \leftarrow \mathbf{ct}_1 + \mathbf{ct}_2 \pmod{Q_\ell}$.
- $\text{CMULT}(\mathbf{ct}, c)$. For $\mathbf{ct} \in \mathcal{R}^2_\ell$ and $c \in \mathcal{R}$, output $\mathbf{ct}_{\text{cmult}} \leftarrow c \cdot \mathbf{ct} \pmod{Q_\ell}$.
- $\text{MULT}_{\mathbf{evk}}(\mathbf{ct}_1, \mathbf{ct}_2)$. For $\mathbf{ct}_i = (b_i, a_i) \in \mathcal{R}^2_\ell$, let $(d_0, d_1, d_2) = (b_1 b_2, a_1 b_2 + a_2 b_1, a_1 a_2) \pmod{Q_\ell}$. Output $\mathbf{ct}_{\text{mult}} \leftarrow (d_0, d_1) + \lfloor Q_L^{-1} \cdot d_2 \cdot \mathbf{evk} \rceil \pmod{Q_\ell}$.
- $\text{ROTATE}_{\mathbf{rk}^{(\kappa)}}(\mathbf{ct}, \kappa)$. For $\mathbf{ct} = (b, a) \in \mathcal{R}^2_\ell$ and rotation index κ, output $\mathbf{ct}_{\text{rotate}} \leftarrow \left(b^{(\kappa)}, 0\right) + \lfloor Q_L^{-1} \cdot a^{(\kappa)} \cdot \mathbf{rk}^{(\kappa)} \rceil \pmod{Q_\ell}$.
- $\text{RESCALE}\,(\mathbf{ct}, p)$. For a ciphertext $\mathbf{ct} \in \mathcal{R}^2_\ell$ and an integer p, output $\mathbf{ct}' \leftarrow \lfloor 2^{-p} \cdot \mathbf{ct} \rceil \pmod{(Q_\ell / 2^p)}$.

The CKKS scheme supports an efficient packing of r (up to $n/2$) real numbers into a single ciphertext. The encoding and decoding operations are defined as follows:

- $\text{ENCODE}\,(\mathbf{w}, p)$. For $w \in \mathbb{R}^r$, output the polynomial $m \leftarrow \lfloor \phi(2^p \cdot \mathbf{w}) \rceil \in \mathcal{R}$.
- $\text{DECODE}\,(m, p)$. For a plaintext $m \in \mathcal{R}$, output the polynomial $\mathbf{w} \leftarrow \phi^{-1}\,(m/2^p) \in \mathbb{R}^r$.

Here, $\phi(x)$ is a certain complex canonical embedding map, which is similar conceptually to inverse Fourier transform.

Our RNS variant of the CKKS scheme

Our CKKS variant performs all operations in RNS. In other words, the power-of-two modulus $Q_\ell = 2^\ell$ is replaced with $\prod_{i=1}^{\ell} q_i$, where q_i are same-size prime moduli satisfying $q_i \equiv 1 \bmod 2n$ (for efficient number theoretic transforms (NTT) that convert native-integer polynomials w.r.t. each CRT modulus from coefficient representation to the evaluation one, and vice versa). The primes are chosen to be as close to 2^p as possible to minimize the error introduced by rescaling.

The two major changes in our variant compared to the original CKKS scheme deal with rescaling and key switching. We also made two other minor changes. First, we use the ternary random discrete distribution for χ_{key} and χ_{enc} instead of the sparse distributions as the lattice attacks for this case are better studied, and the ternary distribution is included in the HE standard [11]. Second, we do additional scaling of plaintexts and ciphertexts to support the use of RNS (only native integer arithmetic) during encoding/decoding.

Rescaling in RNS

To efficiently perform rescaling in RNS from Q_ℓ to $Q_{\ell-1}$, we replace the scaling down by 2^p with scaling down by q_ℓ. We choose all q_i, where $i \in [L]$, such that $2^p/q_i$ is in the range $(1 - 2^{-\epsilon}, 1 + 2^{-\epsilon})$, where ϵ is kept as small as possible. To minimize the cumulative approximation error growth in deeper computations, we also alternate q_i w.r.t. 2^p. For instance, if $q_1 < 2^p$, then $q_2 > 2^p$ and $q_3 < 2^p$, etc.

The new rescaling operation to scale down by one level is defined as

- $\text{RESCALERNS}\,(\mathbf{ct})$. For a ciphertext $\mathbf{ct} \in \mathcal{R}^2_\ell$, output $\mathbf{ct}' \leftarrow \lfloor q_\ell^{-1} \cdot \mathbf{ct} \rceil \pmod{Q_{\ell-1}}$.

We derive the procedure for computing $\lfloor q_\ell^{-1} \cdot \mathbf{ct} \rceil \pmod{Q_{\ell-1}}$ using the CRT scaling technique proposed in [12]. Consider the following CRT representation of a multiprecision integer $x \in \mathbb{Z}_{Q_\ell}$:

$$x = \sum_{i=1}^{\ell} x_i \cdot \tilde{q}_i \cdot q_i^* - \upsilon' \cdot\ Q_\ell \text{ for some } \upsilon' \in \mathbb{Z}, \qquad (1)$$

where

$$q_i^* = Q_\ell / q_i \in \mathbb{Z} \text{ and } \tilde{q}_i = q_i^{*\,-1} \pmod{q_i} \in \mathbb{Z}_{q_i}.$$

Then we can write

$$\frac{x}{q_\ell} = \frac{1}{q_\ell}\left(\sum_{i=1}^{\ell-1} x_i \tilde{q}_i q_i^* + x_\ell \tilde{q}_\ell q_\ell^* - \upsilon' Q_\ell\right).$$

After rounding and applying the modulo reduction, the last term is removed yielding

$$\left\lfloor \frac{x}{q_\ell} \right\rceil \equiv \sum_{i=1}^{\ell-1} x_i \cdot \frac{\tilde{q}_i q_i^*}{q_\ell} + \left\lfloor x_\ell \cdot \frac{\tilde{q}_\ell q_\ell^*}{q_\ell} \right\rceil \pmod{Q_{\ell-1}}. \qquad (2)$$

The first term can be directly computed in RNS by summing up the products of x_i and $q_\ell^{-1} \pmod{q_i}$. For the second term, we precompute the residues of $\left\lfloor \frac{\tilde{q}_\ell q_\ell^*}{q_\ell} \right\rfloor$ and multiply them by the corresponding residues of x_ℓ during rescaling. Then we add the fractional part, which has the residue of $\lfloor x_\ell / q_\ell \rceil$, i.e., 0 or 1, for each CRT modulus q_i. Note that the fractional part is negligibly small and hence can be excluded from the implementation.

The computational complexity of rescaling is determined by the computation in the second term of (2). We first need to run one native inverse NTT for residues w.r.t. q_ℓ and then $\ell - 1$ native NTTs to go back to the evaluation representation. All the computations in the first term of (2) are done directly in evaluation representation. Therefore, each rescaling operation requires ℓ native-integer NTTs.

The maximum approximation error introduced by rescaling from ℓ to $\ell - 1$ is $\left|q_\ell^{-1} \cdot m - 2^{-p} \cdot m\right| \leq 2^{-\epsilon} \cdot \left|2^{-p} \cdot m\right|$.

This procedure can be easily generalized to support scaling down by multiple CRT moduli. This case is similar to the first stage of complex scaling in CRT representation described in Section 2.4 of [12].

Key switching

For key switching, we use the CRT decomposition key switching algorithm that was originally proposed in [13] and improved in [12] for the Brakerski/Fan-Vercauteren (BFV) scheme. The advantages of this technique vs. the one used in the original CKKS scheme (initially proposed for the Brakerski-Gentry-Vaikuntanathan scheme in [14]) are that this technique has lower computational complexity for relatively small numbers of levels (up to 8 or so), and does not require an approximately two-fold increase in the ring dimension to support the appropriate lattice security level. Both of these benefits were important for our solution.

The operations of the CKKS scheme that are modified by the key switching procedure are rewritten as:

- $\textsc{KSGenRNS}_{\mathbf{sk}}(s')$. For $s' \in \mathcal{R}$, sample a random $a_i' \leftarrow \mathcal{R}_L$ and error $e_i' \leftarrow \chi_{err}$. Output the switching key as $\mathbf{swk} \leftarrow \left\{(b_i', a_i')\right\}_{i\in[L]} \in \mathcal{R}_L^{2\times L}$, where $b_i' \leftarrow -a_i's' + e_i' + \tilde{q}_i \cdot q_i^* \cdot s' \pmod{Q_L}$. Set $\mathbf{evk} \leftarrow \textsc{KSGenRNS}_{\mathbf{sk}}\left(s^2\right)$. Set $\mathbf{rk}^{(\kappa)} \leftarrow \textsc{KSGenRNS}_{\mathbf{sk}}\left(s^{(\kappa)}\right)$.
- $\textsc{MultRNS}_{\mathbf{evk}}(\mathbf{ct}_1, \mathbf{ct}_2)$. For $\mathbf{ct}_i = (b_i, a_i) \in \mathcal{R}_\ell^2$, let $(d_0, d_1, d_2) = (b_1b_2, a_1b_2 + a_2b_1, a_1a_2) \pmod{Q_\ell}$. Decompose d_2 into its CRT components $[d_2]_{q_i}$ and output
$$\mathbf{ct}_{\text{mult}} \leftarrow (d_0, d_1) + \sum_{i=1}^{\ell} [d_2]_{q_i} \cdot \mathbf{evk}_i \pmod{Q_\ell}.$$
- $\textsc{RotateRNS}_{\mathbf{rk}^{(\kappa)}}(\mathbf{ct}, \kappa)$. For $\mathbf{ct} = (b, a) \in \mathcal{R}_\ell^2$, output
$$\mathbf{ct}_{\text{rotate}} \leftarrow \left(b^{(\kappa)}, 0\right) + \sum_{i=1}^{\ell} \left[a^{(\kappa)}\right]_{q_i} \cdot \mathbf{rk}_i^{(\kappa)} \pmod{Q_\ell},$$
where $\left[a^{(\kappa)}\right]_{q_i}$ are CRT components of $a^{(\kappa)}$.

Each key-switching operation requires one inverse NTT (ℓ native-integer NTTs) to switch d_2 (or $a^{(\kappa)}$ for rotation) from evaluation to coefficient representation and then ℓ NTTs ($\ell^2 - \ell$ native-integer NTTs) to go back to evaluation representation for each CRT component. Hence, the total complexity in terms of native-integer NTTs is ℓ^2.

This key switching procedure also supports a second level of decomposition by extracting base-w digits in each residue using the procedure described in Appendix B.1 of [13].

Noise estimates

We present here heuristic noise estimates for the RNS variant of CKKS using the canonical embedding norm, which corresponds to the infinity norm for the evaluation of a polynomial $\mathcal{R}$ at $2n$ complex roots of unity. For more details on the canonical embedding mapping and norm, the reader is referred to [10]. The main differences between our expressions and those in [10] are due to the use of ternary uniform distribution and a different key switching technique.

- **Encoding and Encryption.** The bound for fresh encryption $B_{\text{clean}} = 6\sigma\left(4\sqrt{3}n + \sqrt{n}\right)$, where σ is the standard deviation for error distribution. The decoding is correct as long as $2^p > n + 2B_{\text{clean}}$.
- **Addition.** The bound for homomorphic addition $B_{\text{add}} = B_1 + B_2$, where B_i is the noise bound for i-th ciphertext.
- **Rescaling.** The noise bound for rescaling is $B_{\text{rescale}} = q_\ell^{-1} \cdot B + B_{\text{scale}}$, where B is the input noise and $B_{\text{scale}} = \sqrt{3}\left(12n + \sqrt{n}\right)$.
- **Rotation.** The noise bound for rotation (key switching) is $B_{\text{ksw}} = \frac{8}{\sqrt{3}} \cdot n\sigma w \left\lceil \log_w q_\ell \right\rceil$.
- **Multiplication.** If we have two ciphertexts $\mathbf{ct}_1$ and $\mathbf{ct}_2$ with $\|m_1\|_\infty^{\text{can}} < \nu_1$, noise bound B_1 and $\|m_2\|_\infty^{\text{can}} < \nu_2$, noise bound B_2, respectively, the noise bound $B_{\text{mult}} = \nu_1 B_2 + \nu_2 B_1 + B_1 B_2 + B_{\text{ksw}}$.

In most cases, the parameter selection is determined by the multiplicative depth and the approximation error in rescaling. The approximation error (with about ϵ bits being "erased" by rescaling) dominates the noise growth of other operations and should be done last (after a multiplication). The only practical exception is when rotations are performed before any multiplications. In this case, the key switching noise may be high if the w-base is large, e.g., comparable to 2^p as in the case of CRT decomposition without further digit decomposition of each residue.

Comparison to the RNS variant by Cheon et al. [1]

Both our RNS variant of CKKS and the variant proposed by Cheon et al. [1] work with an RNS basis consisting of native-integer primes q_i that are close to 2^p (with ϵ bits of precision). In other words, scaling down by 2^p is replaced with approximate scaling down by q_ℓ. Hence the rescaling approach in both variants is similar. The techniques for the scaling operation itself are different, but the computational complexity of both scaling techniques appears to be the same (requiring ℓ native-integer NTTs).

The key switching-procedure developed in [1] is based on the approach originally proposed for the Brakerski-Gentry-Vaikuntanathan scheme [14], which requires doubling the ciphertext modulus (and roughly doubling the ring dimension). We use the residue/digit decomposition approach originally proposed in [13] and improved in [12]. Our key-switching technique requires more NTTs but provides better overall performance for relatively "shallow" circuits (our estimates suggest this approach should be faster up to 8 levels or so).

Plaintext encoding

Our solution uses two kinds of plaintext encoding. Initially, $\mathbf{X}$ and $\mathbf{y}$ are packed in single ciphertexts similar to how it was done in [8]. We denote this as *packed-matrix encoding*. All matrix products in steps 2 through 8 of Algorithm 3 use the rotation-based SumRowVec and SumColVec procedures from [5]. Later in the algorithm (starting from step 9), the solution switches to single-integer ciphertexts for $\mathbf{X}$ and the vectors and matrices derived from $\mathbf{X}$ and $\mathbf{y}$. We call the latter encoding as *packed-integer encoding*. As a result of this, our matrix operations with the SNPs data (first appearing in step 9) involve only cheap SIMD multiplications and additions of packed-integer and packed-row-vector ciphertexts, and do not involve any expensive rotations. All operations before computing on the SNPs data are performed using packed-matrix (single) ciphertexts.

Packed-matrix encoding

The packed-matrix encoding packs a full matrix or vector into a single ciphertext, cloning as many entries as needed to support matrix-matrix and matrix-vector products. The cloning makes it possible to minimize the number of computationally expensive rotations in matrix-matrix (vector) products.

We encode/encrypt both $\mathbf{X}$ and $\mathbf{X}^\top$ to avoid calling transposition in the encrypted domain. We pack $\mathbf{X} \in \mathbb{R}^{N\times k}$ in a row-wise order, cloning each row $k-1$ times before going to the next row. Here, we introduce $k = K+1$ for brevity.

$$\mathbf{X} = \begin{bmatrix} X_{11} & X_{12} & \dots & X_{1k} \\ X_{11} & X_{12} & \dots & X_{1k} \\ \vdots & \vdots & \vdots & \vdots \\ X_{21} & X_{22} & \dots & X_{2k} \\ X_{21} & X_{22} & \dots & X_{2k} \\ \vdots & \vdots & \vdots & \vdots \\ X_{N1} & X_{N2} & \dots & X_{Nk} \\ X_{N1} & X_{N2} & \dots & X_{Nk} \end{bmatrix}$$

We pack $\mathbf{X}^\top \in \mathbb{R}^{k\times N}$ by taking each element of matrix $\mathbf{X}$ (marshalling it in the row-wise order) and cloning it to form a complete row.

$$\mathbf{X}^\top = \begin{bmatrix} X_{11} & X_{11} & \dots & X_{11} \\ X_{12} & X_{12} & \dots & X_{12} \\ \vdots & \vdots & \vdots & \vdots \\ X_{1k} & X_{1k} & \dots & X_{1k} \\ \vdots & \vdots & \vdots & \vdots \\ X_{N1} & X_{N1} & \dots & X_{N1} \\ X_{N2} & X_{N2} & \dots & X_{N2} \\ \vdots & \vdots & \vdots & \vdots \\ X_{Nk} & X_{Nk} & \dots & X_{Nk} \end{bmatrix}$$

Both matrices require $N \cdot k^2$ slots.

We pack $\mathbf{y} \in \mathbb{R}^N$ column-wise by cloning $\mathbf{y}$ $k^2 - 1$ times to the right. That is, we have

$$\mathbf{y} = \underbrace{\begin{bmatrix} y_1 & y_1 & \dots & y_1 \\ y_2 & y_2 & \dots & y_2 \\ \vdots & \vdots & \vdots & \vdots \\ y_N & y_N & \dots & y_N \end{bmatrix}}_{k^2 \text{ cloned values}}$$

The resulting vector $\boldsymbol{\rho}$ is represented the same way as $\mathbf{y}$. Both use $N \cdot k^2$ slots.

The diagonal matrix $\mathbf{W}$ is represented as a vector by extracting the diagonal, and the resulting vector is packed in the same format as $\boldsymbol{\rho}$.

The SNPs matrix $\mathbf{S}$ is encoded either as an array of ciphertexts (when $M > n/2$) or a single ciphertext (when $M \leq n/2$) without any cloning, i.e., the classical SIMD packing of vectors is used.

Matrices and vectors, such as $\mathbf{X}$ and $\mathbf{y}$, can be encoded in a single ciphertext as long as $N \cdot k^2 \leq n/2$. If this condition does not hold, the packing can be trivially extended to multiple ciphertexts per matrix/vector.

Packed-integer encoding

To support efficient matrix multiplication without rotations, we also encode $\mathbf{X}$ as $N \cdot k$ single-integer ciphertexts. In this case, each entry of $\mathbf{X}$ is cloned to all slots of a single ciphertext. We denote such packing of $\mathbf{X}$ as $\mathbf{X}_1$.

Conversion from packed-matrix to packed-integer encoding

The main bottleneck of our solution is the conversion of vectors from a packed-matrix ciphertext to multiple packed-integer ciphertexts. We have developed and implemented three different methods for performing this conversion. Based on the requirements for performance and scalability, we chose one of these methods for our prototype.

Algorithm 3 Annotated HE Computation (all scalars, vectors, and matrices are encrypted except for α and constants)

```
ENCRYPTED INPUTS: $\mathbf{X}, \mathbf{X}^\top, \mathbf{X}_1, \mathbf{y}, \mathbf{S}$
ENCRYPTED OUTPUTS: $\vec{z}^2_{\text{den}}, \vec{z}_{\text{num}}$
1: $\alpha \leftarrow 0.015$    ▷ plaintext constant
2: $\boldsymbol{\rho} \leftarrow 0.15625\alpha \cdot \mathbf{X}\left(\mathbf{X}^\top(\mathbf{y}-\mathbf{0.5})\right)+\mathbf{0.5} \in \mathbb{R}^N$    ▷ adds 3 levels (taking into account the summation depth increase); we use the packed $\mathbf{X}$ here instead of $\mathbf{X}^\top$; $D$=3.
3: $\mathbf{W} \leftarrow \boldsymbol{\rho} \star (\mathbf{1}-\boldsymbol{\rho}) \in \mathbb{R}^N$    ▷ $\star$ denotes SIMD multiplication; adds 1 level; $D$=4.
4: $\boldsymbol{\zeta} \leftarrow$ ZEXPAND $(\boldsymbol{\rho}, \mathbf{y}) \in \mathbb{R}^N$    ▷ Polynomial evaluation; 8-in-series product; depth 4 w.r.t. $\boldsymbol{\rho}$; $D$=7.
5: $\mathbf{H} \leftarrow \left(\mathbf{X}^\top\mathbf{W}\right)\mathbf{X} \in \mathbb{R}^{k\times k}$    ▷ depth 2 w.r.t. $\mathbf{W}$; first product is a SIMD multiplication. $D$=6.
6: $\mathbf{B} \leftarrow$ ADJOINT $(\mathbf{H}) \in \mathbb{R}^{k\times k}$    ▷ 2-in-series products; depth-2 HM + depth 1 for bit mask multiplication; adds 2 levels; $2k^2$ rotations; convert $\mathbf{B}$ into $k^2$ packed-integer ciphertexts, denoted as $\mathbf{B}_1$; $D$=9 for $\mathbf{B}$; $D$=10 for $\mathbf{B}_1$.
7: $d_1 \leftarrow$ DETERMINANT $(\mathbf{H}) \in \mathcal{R}$    ▷ 3-in-series products; depth-2 HMs + depth 1 for bit mask multiplication; no depth increase; $D$=9.
8: $\boldsymbol{\zeta}^* \leftarrow d_1 \cdot \boldsymbol{\zeta} - (\mathbf{XB})\left(\left(\mathbf{X}^\top\mathbf{W}\right)\boldsymbol{\zeta}\right) \in \mathbb{R}^N$    ▷ Adds 2 HMs + 2 bit mask multiplications to depth = 4 levels; $D$=13.
9: $\mathbf{S}^* \leftarrow d_1 \cdot \mathbf{S} - \mathbf{X}_1(\mathbf{B}_1(\mathbf{X}_1^\top(\mathbf{W}_1\mathbf{S}))) \in \mathbb{R}^{N\times m}$    ▷ Adds 1 to depth; most expensive matrix multiplication costing roughly $2Nk$ ciphertext multiplications; need to convert 1 ciphertext $\mathbf{W}$ into $N$ $\mathbf{W}_1$ ciphertexts; $D$=14.
10: $\vec{z}^2_{\text{den}} \leftarrow \left((d_1 \cdot d_1)\cdot \mathbf{W}_1^\top\right)(\mathbf{S}^* \star \mathbf{S}^*) \in \mathbb{R}^{1\times m}$    ▷ SIMD squaring in computing $\mathbf{S}^* \star \mathbf{S}^*$; adds 2 levels; $D$=16.
11: $\vec{z}_{\text{num}} \leftarrow (\mathbf{W}\boldsymbol{\zeta}^*)_1{}^\top \mathbf{S}^* \in \mathbb{R}^{1\times m}$    ▷ first product is SIMD multiplication; we use the index 1 here to denote the conversion of the packed-matrix ciphertext into $N$ packed-integer ciphertexts; $D$=16.
    NOTE: HM is homomorphic multiplication; $D$ is current depth; subscript 1 denotes packed-integer encoding.
```

To illustrate the problem and its solutions, we consider the task of converting the packed-matrix single-ciphertext encryption of $\mathbf{y}$ into N packed-integer ciphertexts. A similar task has to be executed twice in our algorithm for secure GWAS.

Method 1: $N\lceil\log n\rceil$ rotations

Our first solution can be summarized as follows:

1 Fill all $n/2$ slots of $\mathbf{y}$ by cloning existing $N \cdot k^2$ slots. This requires $\log\left(n/\left(2\bar{N}\cdot k^2\right)\right)$ rotations and additions. The cloning procedure is described in [8]. Here, $\bar{N} = 2^{\lceil\log N\rceil}$.
2 Run N bit mask multiplications to form N ciphertexts each containing $n/\left(2\bar{N}\right)$ cloned values for each component of $\mathbf{y}$. All other slots are zeroed out.
3 Clone existing $n/\left(2\bar{N}\right)$ non-zero values to all slots in each of the N ciphertexts. This operation requires $N\lceil\log N\rceil$ rotations and additions, and is the main bottleneck of the computation.

Method 2: $\bar{N}$ rotations and $\lceil\log N\rceil$ depth increase

The idea of our second solution is to represent the conversion as a binary tree. At each level i of the tree we perform i rotations, $4\cdot i$ bit mask multiplications, and $2\cdot i$ additions, getting two output ciphertexts from each input ciphertext. Although this recursive method requires only $\bar{N}$ rotations, $4\bar{N}$ bit mask multiplications, and $2\bar{N}$ additions, there is a $\lceil\log N\rceil$ depth increase due to bit mask multiplications at each level of the binary tree.

To illustrate this approach, consider a simpler case (the logic would stay the same when we clone y_i any number of times):

$$\left[y_1y_2y_3\cdots y_{N-2}y_{N-1}y_N\right].$$

First rotate by -1 and get

$$Rot_1(y) = \left[y_Ny_1y_2\cdots y_{N-3}y_{N-2}y_{N-1}\right].$$

Then multiply both y and $Rot_1(y)$ by $M_1 = [\,101010\cdots 10]$ and $M_2 = [\,010101\cdots 01]$, and sum up two possible combinations, yielding

$$y_{1,1} = y{\star}M_1{+}Rot_1(y){\star}M_2 = \left[y_1y_1y_3y_3\cdots y_{N-1}y_{N-1}\right],$$

$$y_{1,2} = y{\star}M_2{+}Rot_1(y){\star}M_1 = \left[y_Ny_2y_2\cdots y_{N-2}y_{N-2}y_N\right].$$

Next compute $Rot_2(y_{1,1})$ and $Rot_2(y_{1,2})$, multiply $y_{1,1}$ and $y_{1,2}$ and their rotations by $[\,110011\cdots 1100]$ and $[\,001100\cdots 0011]$ for each pair, and sum up four possible combinations. Now there are 4 $y_{2,i}$ items.

We recursively execute this procedure until the end.

Method 3: $\bar{N}^2$ bit mask multiplications and $\bar{N}$ rotations

Another approach achieving N rotations can be summarized as follows:

1 Fill all $n/2$ slots of $\mathbf{y}$ by cloning existing $N \cdot k^2$ slots.
2 Compute $\bar{N}-1$ cheap rotations of the original ciphertext using the hoisting procedure from [15].
3 For each component of $\mathbf{y}$, do $\bar{N}$ bit mask multiplications (one per rotation) that would extract the component and zero out all other slots.

4 For each component of $\mathbf{y}$, do $\bar{N} - 1$ additions of masked ciphertexts.

Although this procedure requires only roughly $\bar{N}$ cheap rotations, it involves $\bar{N}^2$ bit mask multiplications and additions, which now become the main bottleneck for relatively large values of N.

Comparison of the methods

We implemented all three methods, and carried out both complexity and practical performance comparison.

As N is relatively large (at least 245), $\bar{N}^2$ bit mask multiplications in Method 3 resulted in computation runtimes that are at least 2x-3x larger than Method 1 with $N\lceil\log N\rceil$ rotations. However, Method 3 would be faster for smaller N, e.g., less than 100.

Method 2 is a good option only when the depth increase can be incorporated in the existing circuit without increasing the overall circuit depth. But the scalability of this approach is questionable. The depth increase of $\lceil\log N\rceil = 8$ could not be integrated in the circuit of our solution, and thus we chose Method 1 for our implementation.

Note that in our implementation the depth cost of bit mask multiplication is the same as for homomorphic multiplication, which implies there is room for improvement. Therefore, a more depth-efficient bit mask multiplication procedure may result in a significantly better performance for Method 2, possibly superior to that of Method 1.

Minimizing the number of key switching operations

One of the optimization goals for our solution is to reduce the number of key switching operations, which are used both for rotation and relinearization (after homomorphic multiplication). Each such operation has a high computational complexity, i.e., requires ℓ^2 native-integer NTTs. We have optimized our algorithm to minimize the number of key switching operations. For instance, all computations involving encrypted SNPs data require only 16 (k^2) key switching operations in total. A great majority of the computations involving encrypted SNPs data use only "cheap" SIMD multiplications and additions, and sparingly rescaling operations.

Multiplications with lazy or no relinearization

In steps 9 through 11 of Algorithm 3, our procedure calls only 16 (k^2) relinearizations. In other words, all large-dimension SIMD products are performed without relinearization (the ciphertext size is allowed to grow). The procedure calls the relinearization procedure only when multiplying by $\mathbf{B}_1$ in step 9, which works with the smallest dimension (k) in the chained matrix product. We refer to this deferred relinearization as "lazy" relinearization. Any homomorphic multiplications after this product are performed without a single relinearization, which significantly reduces the runtime of computation.

Use of additions instead of rotations

The packed-integer encoding is introduced in steps 9 through 11 of Algorithm 3 to replace any rotation-based summations over rows/columns with SIMD homomorphic additions. The only places where the rotations are used are to homomorphically convert $\mathbf{B}$, $\mathbf{W}$, and $(\mathbf{W}\boldsymbol{\zeta}^*)$ from packed-matrix encoding to the packed-integer one. The use of rotation-based summation in the chained product of step 9 would require a substantially larger number of rotations as compared to the conversion of two vectors of size N and one matrix of size $k \times k$.

Minimizing the number of NTTs

Besides key switching, NTTs are used for rescaling. In some cases, expensive rotations can be replaced with hoisted automorphisms from [15], reducing the number of NTTs for multiple rotations of the same ciphertext to the NTT cost of a single rotation. Our solution minimizes the number of rescaling operations and uses hoisted automorphisms where applicable.

Use rescaling sparingly

We use the following techniques to minimize the number of rescaling operations:

- When there are homomorphic multiplications followed by aggregation of ciphertexts, such as addition of multiple ciphertexts, we apply rescaling after the aggregation, i.e., we call it once rather than for every homomorphic multiplication.
- If there is a benefit in lazy rescaling, e.g., when the number of ciphertexts at the following level is much smaller, we defer rescaling until later. In this case, we have to make sure the depth requirement is not increased, which is true when one of the multiplicands is scaled w.r.t. 2^p rather a power of it.
- The rescaling operations are not called at the end of computation if skipping them does not increase the multiplicative depth of the circuit.

Hoisted automorphisms

Hoisted automorphisms are useful when multiple rotations of the same ciphertext need to be computed [15]. Our solution encounters this scenario when computing the matrix inversion of $\mathbf{H}$ in steps 6 and 7 of Algorithm 3, and hence the hoisted automorphisms are used there in favor of regular rotations.

Minimizing the noise growth and ciphertext modulus

We minimized the noise growth/ciphertext modulus of the computation circuit using the following techniques:

- Binary tree multiplication was employed for any chained products of ciphertexts.
- Closed-form expressions (such as in step 2 of Algorithm 3) were derived to get the maximum benefit from binary tree multiplication.

- Binary tree addition for any summation of a large number of ciphertexts was employed to achieve a $O(\log N)$ noise growth.
- To guarantee that the end result of the computation requires only one native-integer polynomials, we multiplied both numerator and denominator by estimated scaling factors (different from 2^p). These factors were introduced during bit mask multiplications to avoid any extra depth increase due to this additional scaling.
- The maintenance operations of HE, such as key switching and rescaling, were properly ordered to minimize the noise growth. For instance, rescaling was done after the rotations following a multiplication (not before).

Harnessing the CRT ladder

As the circuit evaluation progresses, the number of CRT limbs, i.e., native polynomials in the Double-CRT structure, gets reduced due to rescaling. For instance, at level ℓ the number of CRT limbs is reduced by $L-\ell$ as compared to fresh ciphertexts. This provides a speedup in CKKS compared to scale-invariant schemes, such as BFV. We can further take advantage of the decreasing CRT "ladder" by encrypting plaintexts at the level they are first used and by compressing evaluation keys as the computation progresses. This reduces storage requirements. We also minimize the number of CRT limbs by finding the minimum number of limbs needed for correct result (starting from the end of the computation circuit). Below we provide some examples of how these techniques are applied in our solution.

Encrypt ciphertexts at the level first used

As the SNPs matrix $\mathbf{S}$ is first used in step 9 of Algorithm 3 (after 10 levels of computation), we encrypt it using 7 CRT limbs rather than 17 corresponding to the initial ciphertext modulus. This reduces the storage requirements for the SNPs matrix by a factor of 2.4x.

Compress evaluation keys as needed

Same rotation keys are used multiple times throughout the computation. Whenever they are no longer required below a certain level, we compress them to the current level, thus reducing the number of CRT limbs. Note that the rotation keys consume most of the space utilized by public keys in our solution.

Use the lowest number of CRT limbs for ciphertexts

Once the lowest multiplicative depth for the circuit is determined, we choose the actual level for ciphertexts by counting from the end of the circuit (not from the beginning) up to the specific computation. This minimizes the number of CRT limbs used, thus reducing both runtime and storage requirements.

Consider the example of $\mathbf{S}$. If we were to count the level from the beginning of the circuit, we would choose level 8 (to match the level of $\mathbf{B}_1$). But we choose 10 instead because the maximum depth of computations from $\mathbf{S}$ in step 9 to the end of the circuit is 6. This gives more than 1.5x runtime improvement for the rotations in the conversion from $\mathbf{W}$ to $\mathbf{W}_1$, which is done immediately before computing $\mathbf{W}_1\mathbf{S}$. The storage requirement for $\mathbf{S}$ is also reduced by roughly a factor of 1.3x.

Matrix inversion

As pointed out earlier, we use Cramer's rule to compute the matrix inverse of $\mathbf{H}$. The numerator is the adjoint of $\mathbf{H}$ while the denominator is the determinant of $\mathbf{H}$. To extract specific components of $\mathbf{H}$, we use cheap rotations (hoisted automorphisms) followed by bit mask multiplications to clear out the values that are not used. As both numerator and denominator contain a lot of common products of the rotations for $\mathbf{H}$, we wrote both of them down in the closed form and compute common products only once. The closed form for the determinant also allows the direct application of binary tree multiplication (3-in-series products require a binary depth of 2). The depth cost of these steps is 3 (2 for homomorphic multiplications and 1 for bit mask multiplication).

When computing the determinant and k^2 components in the adjoint, all homomorphic multiplications are performed without relinearization, and the relinearization is applied at the very end (for each component) after all additions and subtractions are done. This significantly reduces the number of expensive key switching operations when computing the matrix adjoint and determinant.

The procedure for computing the adjoint and determinat also prepares the packed-matrix variant of $\mathbf{B}$ for computing $\boldsymbol{\zeta}^*$ in step 8 and the packed-integer variant $\mathbf{B}$, i.e., $\mathbf{B}_1$, for computing $\mathbf{S}^*$ in step 9 by performing appropriate rotations and additions. The final rescaling for the components in the adjoint and determinant is done after all rotations are computed. Otherwise the noise growth in rotations would lead to incorrect results after decryption.

Order of products in matrix chain multiplication

The order of matrix products in matrix chain multiplications has a major effect on the performance of our solution. The two most complex and costly chained matrix products in Algorithm 3 are step 8 (computation of $\boldsymbol{\zeta}^*$) and step 9 (computation of $\mathbf{S}^*$). Typically the matrix chain multiplication problem is an optimization problem that can be solved using dynamic programming. In the case of regular plaintext computations, the goal is usually to minimize the number of element multiplications. In the encrypted solution, additional constraints are introduced, and these constraints can be different depending on the plaintext encoding used, as illustrated below.

In step 8, we work with a chain of single ciphertexts (packed matrix encoding). The constraints for this case can be summarized as follows:

- Make sure the outcome of each intermediate product is a single ciphertext. For instance, we cannot have a product where outer dimensions are both N.
- The costs of SUMROWVEC and SUMCOLVEC are different. The latter requires a bit mask multiplication, and the number of rotations corresponds either to row or column size. The possible constraints are to minimize the number of rotations and/or minimize the depth of bit mask multiplications.
- Minimize the depth of the overall circuit. In other words, the term at highest level should be given special attention. The binary tree multiplication technique should also be properly applied.

In step 9, we work with products of many packed-integer ciphertexts and N SIMD-packed ciphertexts (for each row of matrix $\mathbf{S}$). The guidelines for optimization in this case can be summarized as follows:

- Minimize the total number of SIMD multiplications.
- Minimize the depth of the overall circuit. In other words, the term at highest level should be given special attention. The binary tree multiplication technique should also be properly applied.

In our solution, the decisions regarding the order of matrix chain multiplication were done by hand. But in a more general case, where the computation circuit is built automatically, one would have to include algorithms for finding the optimal order by solving the appropriate dynamic optimization problem.

Loop parallelization

To benefit from multi-core CPU environments, our solution applies loop parallelization at various levels.

At the encryption stage, the parallelization is done for the loop iterating over all individuals (size N, which is at least 245). This implies the encryption runtime should decrease almost linearly with the number of physical cores.

In the computation stage, the following loop parallelizations are applied:

- All matrix products in $\mathbf{X}_1\left(\mathbf{B}_1\left(\mathbf{X}_1^\top\left(\mathbf{W}_1\mathbf{S}\right)\right)\right)$ at step 9 of Algorithm 3 are parallelized over inner dimensions (N or k, depending on the product).
- All SIMD products in steps 10 and 11 of Algorithm 3 are parallelized over N.
- In matrix inversion, the extraction of k^2 components of $\mathbf{H}$ is parallelized over k^2.
- In the homomorphic encoding conversion routine of Method 1, the parallelization is applied to the main loop over N.
- Loop parallelization is also applied in many places at the level of CKKS and lower-lever ring operations. In the case of NTTs for polynomials in Double-CRT representation, the parallelization is done over ℓ. In the case of RNS subroutines, the parallelization is applied at the level of polynomial coefficients (dimension n).

Results

Dataset

Our experiments were performed using the training dataset provided by the iDASH 2018 organizers. The training data were extracted from the Personal Genome Project (https://www.personalgenomes.org/us). The dataset includes 245 individuals, 10,643 SNPs, and 3 covariates. We also generated larger datasets for scalability analysis by re-sampling the original dataset.

Software implementation

We implemented our solution in PALISADE v1.2 [16]. We added our own implementation for the RNS variant of the CKKS scheme to PALISADE. For loop parallelization, we used OpenMP.

Parameter selection

The parameters used are summarized below. According to [11], our parameters correspond to at least 128 bits of security for classical computers.

- The size of ciphertext modulus Q_L for fresh ciphertexts is 850 bits.
- The ring dimension n is $2^{15} = 32,768$.
- The number of CRT limbs in the fresh ciphertext modulus is 17 ($L = 17$), which corresponds to 16 levels in the computation circuit. Each CRT modulus is 50 bits long.
- Number of bits p in the plaintext scaling factor of CKKS scheme is 50. For this value of p, the approximation error introduced by each rescaling typically affected up to 25 least significant bits of the encrypted data.
- The key switching window matches the size of CRT moduli, i.e., 50 bits.
- We use the ternary secret key distribution, i.e., random integers between -1 and 1, as commonly done for BFV.
- The error distribution parameter σ is 3.19.

Table 1 Maximum storage requirements for $N = 245$; $M = 10,643$; $K = 3$

Ciphertexts [GB]					Evaluation Keys [GB]	
$\mathbf{X}$	$\mathbf{X}^\top$	$\mathbf{y}$	$\mathbf{S}$	$\mathbf{X}_1$	Rotation	Relinearization
0.0085	0.0085	0.0085	0.84	2.87	3.65	0.42

Performance results

Storage requirements

The maximum (initial) storage requirements for the case of $N = 245$; $M = 10,643$; $K = 3$ are summarized in Table 1. The storage requirements take into account that $\mathbf{S}$ and $\mathbf{X}_1$ are first used at $\ell = 7$ and $\ell = 6$, respectively. The rotation key size is computed as a sum of space requirements for 16 keys at $\ell = 17$, 13 at $\ell = 12$, and 12 at $\ell = 9$. The relinearization keys are used from the start of the computation ($\ell = 17$). The sizes of public and secret keys are relatively small: 4.7 and 8.5 MB, respectively.

The encryption storage requirements in practical settings can be reduced by converting homomorphically the encrypted packed-matrix ciphertext $\mathbf{X}$ to N packed-integer ciphertexts, i.e., $\mathbf{X}_1$, on demand. This can be done as an offline operation, resulting in an approximately 4x reduction in fresh ciphertext size.

Execution time and peak memory utilization

Table 2 reports the runtimes and peak RAM utilization observed for the official iDASH evaluation environment and a 28-core server node. The results suggest that it takes about 3.5 minutes and about 10 GB of RAM (all ciphertexts and keys are stored in memory) to evaluate homomorphically the GWAS procedure for 245 individuals, 14,841 SNPs, and 3 covariates on a 4-core Amazon instance. The runtime and storage requirements for the case of 1,000 individuals, 131,071 SNPs, and 3 covariates for a modern server computing node (2 x 14 cores) are about 10 minutes and 116 GB, respectively.

Accuracy analysis

We compared the accuracy of the p-values computed using our HE prototype with a plaintext reference implementation of the semi-parallel method proposed by Sikorska et al. [3]. The results for the case of $N = 245$ and $M = 10,643$ are summarized in Fig. 1. The graphs visualize the confusion table when choosing 0.01 as a threshold to classify SNPs as significant or not (depicted as the red lines). It is a log-log plot of the p-values obtained by the two different approaches. The vertical axes correspond to the semi-parallel logistic regression and horizontal axes to the p-values obtained by the HE computation. The diagonal blue line depicts the case when the two classifiers provide exactly the same p-value for each input data.

Each quadrant corresponds to one of possible outcomes: true positive (both classify a SNP as significant), false positive (the semi-parallel model as not significant and the HE computation as significant), true negative (both classify a SNP as significant) and false negative (the semi-parallel model as significant and the HE computation as not significant). The graph shows the true positive rate (TPR), false positive rate (FPR), true negative rate (TPR) and false negative rate (FNR). We use F1 score as a single index to summarize the performance. The graph suggests that the error introduced by our approximation is negligibly small (F1 score of 0.991).

Analysis of our approximations

As described in "Our approximations" section, there are 3 compute-model parameters that affect the approximation error: the highest degree of the Chebyshev polynomials used to approximate the logit-function, d_l; the degree the Taylor expansion of ζ, d_z; and the number of iterations, t, for the gradient descent procedure. Clearly, there is a trade-off between the accuracy of the approximation and the depth of the computation circuit, which determines the computational complexity.

In order to avoid over-fitting, we also used other data sets from the Harvard Personal Genome Project [17]. We ran experiments for different conditions reported in the PGP Participant Survey and found that the approximation of ζ has a significant impact on the quality of the results, and is highly sensitive to the choice of cases and disease populations. Therefore, we selected a relatively high degree for the Taylor expansion, $d_z = 8$, that provides adequate accuracy for unbalanced populations of up to 10%/90%. Note that the data used for the iDASH competition was relatively balanced.

Table 2 Runtimes and peak RAM utilization on a UTHealth ITS VM (4 cores, 16 GB RAM, 200 GB hard drive, AWS T2 Xlarge equivalent, official iDASH'18 evaluation environment) and a server node with 2 x 14 cores of Intel(R) Xeon(R) CPU E5-2680 v4 at 2.40GHz (500 GB RAM and 2 TB hard drive)

System	N	M	KeyGen [min]	Enc [min]	Eval [min]	Dec [s]	Peak RAM [GB]
UTHealth ITS VM (iDASH)	245	14,841	0.35	0.34	3.46	0.06	9.99
28-core server node	245	10,643	0.12	0.059	1.45	0.06	12.2
28-core server node	300	20,000	0.12	0.088	1.88	0.11	16.2
28-core server node	1,000	131,071	0.12	0.72	10.44	0.4	116

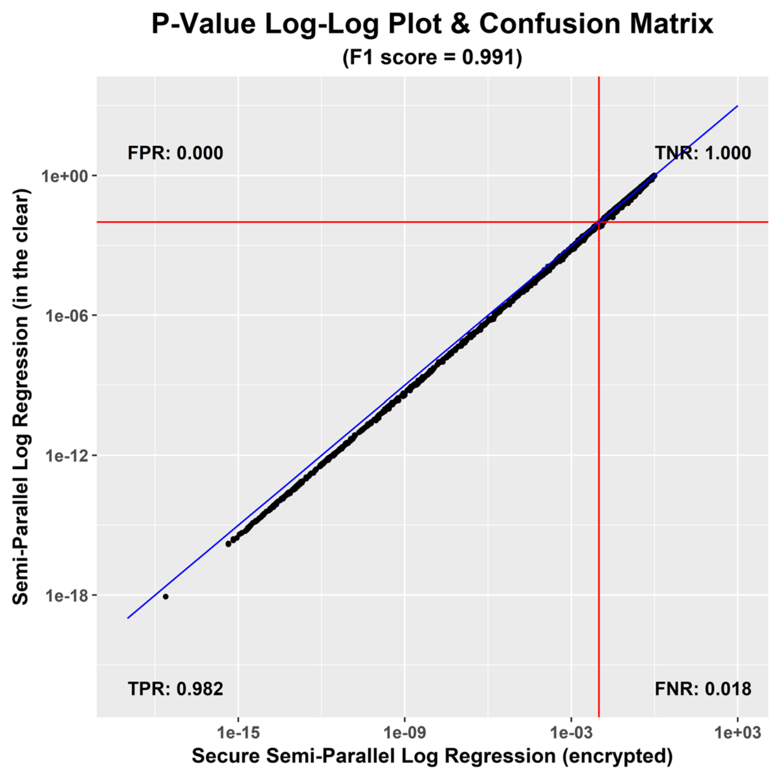

Fig. 1 Accuracy of our encrypted computing prototype w.r.t the plaintext reference implementation [3]

We found that increasing the number of iterations, t, and the degree d_l of the Chebyshev polynomials used to approximate the logit-function has a relatively minor effect on the accuracy of our solution. As an example, Table 3 shows that the F_1 score for the p-value with threshold 0.01 does not significantly change with increase in d_l, while the expected computational cost of using a higher depth would be substantial.

Profiling

Table 4 reports the breakdown of runtimes for three different cases. The results for $N = 245$, $M = 10,643$ suggest that the conversion of vectors from the packed-matrix to packed-integer encoding is the bottleneck for the single-threaded case. However, the conversion procedure parallelizes better (improving by a factor of 15.3x on a 28-core machine) than most of the other operations, effectively reducing its contribution from 77% in the single-threaded experiment to 38% for the 28-threaded experiment. The experiments for larger numbers of SNPs imply that the contribution of the conversion procedure further declines as its computational complexity does not depend on M.

As the maximum size of individuals did not exceed 1,024 in our experiments, all operations in Steps 1–8 of Algorithm 3 worked with single ciphertexts, and the runtime of these steps stayed approximately the same for all experiments. At the same time, the contribution of the matrix products involving $\mathbf{S}$ (steps 9 through 11) significantly increased (from 15% for $N = 245, M = 10,643$ to 68% for $N = 1,000, M = 131{,}071$).

Table 3 F_1 score as a function of the degree d_l of the Chebyshev polynomials used to approximate the logit-function at $d_z = 8$ and $t = 1$

d_l	1	3	5	7	9	11	13	15
F_1	0.9914	0.9924	0.9927	0.9931	0.9932	0.9933	0.9933	0.9933

Discussion

The solution presented in this work was awarded first place (along with another solution from UCSD) in the iDASH'18 competition (Track 2: Secure Parallel Genome Wide Association Studies using Homomorphic Encryption). Hence it represents the state of the art in secure GWAS using homomorphic encryption.

The main limitations of our solution are (1) the need to know the computation and parameters of the semi-parallel procedure in advance and (2) the hand-tuned nature of many optimizations applied to our solution. The first problem can be solved once the bootstrapping for the CKKS scheme becomes more practical. The second

Table 4 Runtime profiling on the 28-core node; time in seconds; numbers in header row denote step #'s in Algorithm 3; numbers in parentheses are for the single-threaded experiment; → denotes the conversion from packed-matrix to packed-integer encoding

N	M	1–5	6–7 + $\mathbf{B} \to \mathbf{B}_1$	8	$\mathbf{W} \to \mathbf{W}_1$	9	10	$\mathbf{W}\zeta^* \to (\mathbf{W}\zeta^*)_1$	11
245	10,643	13.3	23.4	4.6	27.4	10.4	1.8	5.5	0.62
		(27.4)	(40.2)	(6.5)	(419)	(59.3)	(12.0)	(84.1)	(1.64)
300	20,000	13.1	23.5	4.6	33.2	25.7	3.8	7.3	1.5
1,000	131,071	12.7	22.9	4.2	132.8	360.6	47.2	25.0	21.0

challenge can be tackled once automated compilers for homomorphic encryption are developed. Both are open research problems.

Conclusions

The results demonstrate that our solution is able to perform the full GWAS computation homomorphically for 1000 individuals, 131,071 SNPs, and 3 covariates in about 10 minutes on a modern server computing node. Many of the optimizations presented in our paper are general-purpose and can be applied to solving challenging problems dealing with large datasets in other application domains. The major general-purpose optimizations include a new RNS variant of the CKKS scheme and multiple methods of homomorphic switching between data encodings.

Acknowledgements
We gratefully acknowledge the technical assistance with AWS and parallelization by Liron Liptz and Ofer Itzhaki. We also thank the iDASH'18 Organizing Committee for motivating this research study.

About this supplement
This article has been published as part of *BMC Medical Genomics Volume 13 Supplement 7, 2020: Proceedings of the 7th iDASH Privacy and Security Workshop 2018*. The full contents of the supplement are available online at https://bmcmedgenomics.biomedcentral.com/articles/supplements/volume-13-supplement-7.

Authors' contributions
All authors contributed equally to this work. All author(s) have read and approved the manuscript.

Funding
Research reported in this publication was supported by National Human Genome Research Institute of the National Institutes of Health under award number 1R43HG010123. Publication costs were funded by Duality Technologies, Inc.

Availability of data and materials
The iDASH 2018 competition data was only available to registered competition participants.

Ethics approval and consent to participate
Not applicable.

Consent for publication
Not applicable.

Competing interests
AG performed his work for the paper as a consultant for Duality Technogies, Inc. All other authors declare they have no competing interests.

Author details
[1]Duality Technologies, Inc., Newark NJ, USA. [2]Dana-Farber Cancer Institute, Boston MA, USA.

Published: 21 July 2020

References
1. Cheon JH, Han K, Kim A, Kim M, Song Y. A full RNS variant of approximate homomorphic encryption. In: Cid C, Jacobson Jr. MJ, editors. Selected Areas in Cryptography – SAC 2018. Cham: Springer; 2019. p. 347–68.
2. Kim M, Song Y, Li B, Micciancio D. Semi-parallel Logistic Regression for GWAS on Encrypted Data. Cryptology ePrint Archive, Report 2019/294. 2019. https://eprint.iacr.org/2019/294.
3. Sikorska K, Lesaffre E, Groenen PJF, Eilers PHC. Gwas on your notebook: fast semi-parallel linear and logistic regression for genome-wide association studies. BMC Bioinformatics. 2013;14(1):166.
4. Wang X, Tang H, Wang S, Jiang X, Wang W, Bu D, Wang L, Jiang Y, Wang C. iDASH secure genome analysis competition 2017. BMC Med Genet. 2018;11(4):85.
5. Han K, Hong S, Cheon JH, Park D. Efficient logistic regression on large encrypted data. IACR Cryptol ePrint Arch. 2018;2018:662.
6. Chen H, Gilad-Bachrach R, Han K, Huang Z, Jalali A, Laine K, Lauter K. Logistic regression over encrypted data from fully homomorphic encryption. BMC Med Genet. 2018;11(4):81.
7. Kim M, Song Y, Wang S, Xia Y, Jiang X. Secure logistic regression based on homomorphic encryption: Design and evaluation. JMIR Med Inform. 2018;6(2):19. https://doi.org/10.2196/medinform.8805.
8. Kim A, Song Y, Kim M, Lee K, Cheon JH. Logistic regression model training based on the approximate homomorphic encryption. BMC Med Genet. 2018;11:254.
9. Press WH, Teukolsky SA, Vetterling WT, Flannery BP. Numerical Recipes in FORTRAN (2Nd Ed.): The Art of Scientific Computing. Book chapters: "Chebyshev Approximation" (§5.8),"Derivatives or Integrals of a Chebyshev-Approximated Function" (§5.9), and "Polynomial Approximation from Chebyshev Coefficients" (5.10). New York, NY, USA: Cambridge University Press; 1992.
10. Cheon JH, Kim A, Kim M, Song Y. Homomorphic encryption for arithmetic of approximate numbers. In: Advances in Cryptology – ASIACRYPT 2017. Cham: Springer; 2017. p. 409–37.
11. Chase M, Chen H, Ding J, Goldwasser S, et al. Security of homomorphic encryption. Technical report, HomomorphicEncryption.org, Redmond WA. 2017.
12. Halevi S, Polyakov Y, Shoup V. An improved RNS variant of the BFV homomorphic encryption scheme. In: Matsui M, editor. Topics in Cryptology – CT-RSA 2019. Cham: Springer; 2019. p. 83–105.
13. Bajard J-C, Eynard J, Hasan MA, Zucca V. A full RNS variant of FV like somewhat homomorphic encryption schemes. In: SAC 2016. Cham: Springer; 2017. p. 423–42.
14. Gentry C, Halevi S, Smart N. Homomorphic evaluation of the AES circuit. In: "CRYPTO 2012" LNCS vol.7417. Berlin Heidelberg: Springer Berlin Heidelberg; 2012. p. 850–67. http://eprint.iacr.org/2012/099.
15. Halevi S, Shoup V. Faster homomorphic linear transformations in helib. In: CRYPTO 2018. Cham: Springer; 2018. p. 93–120.
16. Polyakov Y, Rohloff K, Ryan GW. PALISADE Lattice Cryptography Library. https://git.njit.edu/palisade/PALISADE. Accessed Aug 2018.
17. The Harvard Personal Genome Project. https://pgp.med.harvard.edu/. Accessed Mar 2019.