University of the Philippines Manila
College of Arts and Sciences
Department of Physical Sciences and Mathematics

# Secure Computation Outsourcing of Genome-Wide Association Studies Using Homomorphic Encryption

A special problem in partial fulfillment
of the requirements for the degree of
**Bachelor of Science in Computer Science**

Submitted by:

Angelica Khryss Yvanne C. Ladisla

June 2016

Permission is given for the following people to have access to this SP:

| | |
|---|---|
| Available to the general public | Yes |
| Available only after consultation with author/SP adviser | No |
| Available only to those bound by confidentiality agreement | No |

## ACCEPTANCE SHEET

The Special Problem entitled "Secure Computation Outsourcing of Genome-Wide Association Studies Using Homomorphic Encryption" prepared and submitted by Angelica Khryss Yvanne C. Ladisla in partial fulfillment of the requirements for the degree of Bachelor of Science in Computer Science has been examined and is recommended for acceptance.

**Richard Bryann L. Chua, M.Sc.**
Adviser

**EXAMINERS:**

| | **Approved** | **Disapproved** |
|---|---|---|
| 1. Gregorio B. Baes, Ph.D. (*candidate*) | _____ | _____ |
| 2. Avegail D. Carpio, M.Sc. | _____ | _____ |
| 3. Perlita E. Gasmen, M.Sc. (*candidate*) | _____ | _____ |
| 4. Marvin John C. Ignacio, M.Sc. (*cand.*) | _____ | _____ |
| 5. Ma. Sheila A. Magboo, M.Sc. | _____ | _____ |
| 6. Vincent Peter C. Magboo, M.D., M.Sc. | _____ | _____ |

Accepted and approved as partial fulfillment of the requirements for the degree of Bachelor of Science in Computer Science.

**Ma. Sheila A. Magboo, M.Sc.**
Unit Head
Mathematical and Computing Sciences Unit
Department of Physical Sciences and Mathematics

**Marcelina B. Lirazan, Ph.D.**
Chair
Department of Physical Sciences and Mathematics

**Leonardo R. Estacio Jr., Ph.D.**
Dean
College of Arts and Sciences

**Abstract**

One of the major problems in genomic research is the delivery of necessary computational power to process extremely large genomic datasets. While cloud computing provides the computing resources needed for such a scenario, the outsourcing of data processing raises a broad range of security and privacy issues as genomic data contain highly sensitive information. This study provides a cryptographic solution to the problem using Paillier scheme. We developed a GWAS computing tool that uses a third-party server to perform genomic computations over encrypted genomic datasets without requiring the decryption key or having any interaction with the data owner; thus preserving the privacy of genomic data.

*Keywords*: genomic data privacy, genome-wide association studies, cryptography, homomorphic encryption, Paillier scheme

# Contents

# List of Figures

# List of Tables

# I. Introduction

## A. Background of the Study

The increasing demand to understand human biology in order to address pressing health issues has pushed the science community to sequence the entire human genome [1, 2]. A genome is a complete set of DNA contained in every cell of an organism that holds the complete genetic instructions used to build, develop, and maintain the organism [3]. These instructions are encoded in the form of four letters: A, C, T, G, which represent the DNA bases Adenine, Cytosine, Thymine, and Guanine, respectively [3]. Identifying the sequence of all the DNA base pairs (bp) in the human genome would provide one of the most useful tools to understand the biological concepts of human health and development. To that end, an international undertaking called the Human Genome Project (HGP) was launched in 1990 [2, 3].

It took thirteen years of efforts from the legion of dedicated scientists who participated in the HGP before the approximately 3 billion bp in the human genome was completely sequenced and released in October 2003 [4, 5]. This accomplishment of HGP to sequence the human genome represents a critically significant milestone in understanding human biology. The next challenge was to decode the instructions in the human genome by deriving meaningful information from the obtained sequence, which would pave the way for improved strategies for diagnosis, treatment, and prevention of diseases [6, 5, 2].

With a goal to translate the success of the HGP into human health benefits, several genome-based studies have surfaced subsequent to the successful sequencing of the human genome. Genome-Wide Association Study (GWAS) is the most common approach in genomic research that involves rapid scanning of genetic markers across the whole genomes in a large-scale population in order to detect any significant clinical associations (e.g. disease, adverse drug reactions) [7, 8]. These studies have led to the development of diagnostic DNA probes and person-

alized therapeutic products that are now being increasingly beneficial in clinical care [8, 9]. More medical advances are anticipated from genomic research in the following years.

Understanding and finding cure for complex diseases require extremely large datasets and powerful tools that can analyze genomic data. Fortunately, the current age of technology suggests that these requirements are viable. In fact, new-generation sequencing platforms are rapidly increasing in both its throughput and affordability [4, 10]; $1000 genome is now a reality [11]. Given the increasing volume of genomic data that has to be processed in genomic research, scientists in the field are aiming to establish a cloud-based common infrastructure that would enable highly accessible shared datasets and computational resources among authorized research institutions worldwide [12, 13, 14]. This will dramatically accelerate the growth of research. However, outsourcing the storage and processing of highly sensitive genomic data to a third-party cloud raises important privacy concerns [4, 13, 14].

## B. Statement of the Problem

There is an arising tension in genomic research between the two priorities of achieving scientific goals and protecting the privacy of the involved human subjects [15]. With the massive data overheads in genomic research, it is becoming more imperative to outsource genomic computations to high-performance computing devices [13, 14]. On the other hand, protecting the privacy of the involved research participants is a crucial responsibility [4, 15].

Handling genomic privacy is complicated considering that genomic data contain unique features. Among others, genomic data reflect information about genetic conditions and predispositions to specific diseases such as Alzheimer's, cancer, or Schizophrenia; do not change overtime (which is not the case of other medical data e.g., blood pressure, glucose level); and contain information about an individual's blood relatives [5, 4, 10]. Researches have shown that the reliance

on the common practice of de-identification as the mechanism of ensuring privacy and avoiding misuse is not viable [4, 16, 17]. The identity of genomic data owner can be recovered, even with only few markers released, by cross-referencing his/her data with publicly available information, and all it requires is an Internet connection [16, 17].

The use of cryptographic techniques can potentially address the privacy issue of genomic data. However, most of the cryptographic schemes are not applicable for cloud environment as these schemes transform genomic data into an encrypted, yet not functional format [4]. Thus, research on secure outsourcing of genomic computation is now gaining interests.

Homomorphic encryption (HE) is a cryptographic technique that allows encrypted data to be computed on directly without the need for decryption [18]. HE can potentially secure the outsourcing of genomic computations without compromising the data utility in research.

## C. Objectives of the Study

To develop a desktop computing tool that can carry out genomic computations in a remote server using the privacy-preserving scheme of the Paillier homomorphic cryptosystem [19], and that has the following user functionalities:

1. To load dataset(s).
2. To perform Minor Allele Frequency (MAF) computation.
3. To perform Hardy-Weinberg Equilibrium (HWE) computation.
4. To perform $\chi^2$ Test Statistic computation.
5. To view the computation results.

Specifically, each aforementioned genomic computation involves the following processes:

1. Encoding of dataset(s).

2. Encryption of dataset(s).

3. Uploading of dataset(s) to the server.

4. Downloading of computation results from the server.

5. Decryption of computation results.

## D. Significance of the Project

Genomic data contains highly confidential information, and the adverse effects of it being leaked is irrevocable [4]. Hence, extreme security is needed to protect the privacy of genome donors in genome-based studies. Unfortunately, the demand to keep these data private somehow hampers the growth of genomic research.

Exploring the feasibility of homomorphic encryption for securing sensitive genomic data will be an essential input to solve the privacy issues needed to address in genomic research. Through this cryptographic technique, the privacy of genomic data will be achieved without significantly degrading its data utility in research. Hence, both the privacy of the research participants and the growth of genomic research will need not be compromised.

## E. Scope and Limitations

1. This project will not cover the storage management of the datasets.

2. This project will not cover the genotyping and sequencing procedures.

3. The genetic markers to be used are Single Nucleotide Polymorphisms (SNPs).

4. The system output will be the results of the specified genomic computations.

5. The computations are intended only for case-control design GWAS.

6. Non-addition operations will be computed on the client side.

7. This project will be using The Homomorphic Encryption Project (THEP) library for implementing Paillier homomorphic encryption scheme.

## F. Assumptions

1. The SNPs are biallelic.

2. The user has case and control datasets for $\chi^2$ test statistic computation.

3. The datasets are in text files.

# II. Review of Related Literature

The participants in genome-based studies have always been informed beforehand of the inevitable risks that may violate their privacy. Even with this premise, many have still entrusted their genomic profile to research institutions given the assurance that their genomic data will be de-identified on public research databases [20]. De-identification (removal of explicit identifying attributes e.g. name) is a method that is widely used for protecting health information. This protection of privacy is regarded as a fundamental principle of research ethics, through which the support of research participants is maintained [21]. However, studies have revealed that genomic data cannot be anonymized by simply removing identifying information [4, 16, 22]. Erlich et al. [16] demonstrated how de-identified genome owners can be re-identified by profiling short tandem repeats on the Y chromosome (Y-STRs) and querying recreational genetic genealogy databases.

In addition to re-identification threats, there is always a risk for an adversary to infer sensitive phenotype information of a genetic-material owner. In 2009, it was discovered that partially available genomic data can be used to infer its unpublished regions due to linkage disequilibrium (LD), a correlation between regions of the genome [4]. As an example, Dr. Jim Watson (the co-discover of the DNA helix structure) donated his sequenced genome with the exception of his ApoE gene, which is an associated gene to Alzheimer's disease. However, it was later shown that the ApoE gene can be inferred from the published genome [23].

Kinship properties extend the risk of privacy invasion to the relatives of the genome owner. For instance, if both parents are genotyped, then most of the variants for their offspring can be inferred. A well-known example is the case of Henrietta Lacks whose cancer cells were used for medical research. Her genome was sequenced and published without the consent of the Lack family. The relatives complained that the sequence also contained much of their information, thus researchers took her genomic data down from databases. However, the previously available data had been already downloaded by many concerned people [24].

These findings have led to the shutting down of some publicly accessible genomic databases [25]. Although some geneticists questioned this step, the research community must respond to the genomic privacy issue. Rapid advances in genomic research, however, necessitate development of common infrastructures and platforms for international collaboration and public participation [13, 14]. One such initiative is the Global Alliance for Genomics and Health (Global Alliance) [26] which aims to build a common framework to enable responsible, voluntary, and secure sharing of genomic and clinical data. To address the privacy threats that hurdle research growth, several cryptographic techniques have been developed.

Baldi et al. [27] presented a privacy-preserving genetic paternity test using Private Set Intersection Cardinality (PSI-CA) protocols to perform private two-party computation of the number of set of elements shared by two parties . It is somewhat similar to Private Set Intersection (PSI) which involves two parties (i.e. the client and the server). PSI allows one party (client) to compute the intersection of its set with that of another party (server) such that the server learns nothing about the client input, and the client learns no information about the server input beyond the intersection. On the other hand, PSI-CA allows the client only to learn the cardinality rather than the contents of the intersection [28]. Such a scheme when performed to a full human genome, though optimal in accuracy, is impractical since the human genome is extremely large ( approximately three billion). Prior work is improved by computing on the chosen 1% of the genome rather than analyzing the entire genome. Nevertheless, experts claim that comparing a properly chosen 1% still yields accuracy.

In the same work of Baldi et al. [27], a cryptographic Primitive Authorized Private Set Intersection (APSI) is used for Privacy-Preserving Personalized Medicine Testing ($\mathrm{P}^3MT$), which extends PSI to enforce authorization of client input. $\mathrm{P}^3MT$ involves an authorization authority (e.g. FDA), a pharmaceutical (Client), and a patient (Server). In this setting, a pharmaceutical company obtains FDA approval on a specific DNA fingerprint $fp$ and receives a corresponding

authorization $auth$; the pharmaceutical company and the patient engage in the protocol where the former inputs ($fp$, $auth$) and the latter inputs his genome. At the end of the protocol, the pharmaceutical learns whether the patients genome matches the fingerprint $fp$, provided that $auth$ is the authorization of $fp$.

As sharing of genomic data among researchers becomes critical, Kumar et al. [29] presented cryptography architecture for secured sharing of genomic sequences. The architecture incorporated two third parties − data storage site (DS) and key holder site (KHS). DS is where encrypted genomic data is stored and processed, while KHS manages the cryptographic keys that are used for encryption and decryption of the genomic records stored in the DS. It implements double encryption using Secure Hash Algorithm (SHA-1) and Message Authentication Code (MAC) to enhance the security of the message.

Blanton et al. [30] developed a cryptographic scheme for a secure outsourcing of a particular computation, i.e. sequence comparisons. A client owns strings $\nu$ and $\mu$, and outsources the computation to two remote servers without revealing the input strings or the output sequence. It utilizes garbled circuit evaluation techniques to avoid the use of public-key cryptography, thus making the solution efficient in practice.

Indeed, there have been many attempts to protect genome privacy using cryptographic methods. However, most of the existing cryptographic solutions limit the utility of genomic data when there should be balance between the normative concerns of maximizing data utility and protecting the privacy of human subjects[4].

In the recent years, cryptography has developed a tool - Homomorphic Encryption (HE) - which can be used to encrypt data in a way that the data can be meaningfully computed in encrypted form. The concept of homomorphic encryption was first suggested by Rivest, Adleman, Dertouzos [31] in 1978. HE is an encryption scheme that permits the computation of nontrivial operations directly on the encrypted data. Suppose $E_K(\cdot)$ is an encryption function with key $K$ and

$D_K(\cdot)$ is the corresponding decryption function. Then $E_K(\cdot)$ is homomorphic with the operator $*$ if there exists an efficient algorithm $Alg$ such that:

$$Alg(E_K(x) * E_K(y)) = E_K(x * y)$$

.

In 1982, Shafi Goldwasser and Silvio Micali proposed the Goldwasser-Micali cryptosystem which is an additive homomorphic encryption. It was a provable security encryption scheme which reached a remarkable level of safety, but it can encrypt only a single bit at a time [18]. In line with this work, Pascal Paillier proposed another provable security encryption system [19] that is also an additive HE in 1999. Going beyond the additive homomorphisms, Boneh-Goh-Nissim scheme [32] presented an encryption scheme that can perform arbitrarily many additions as well as a single multiplication on ciphertexts. RSA cryptosystem [33] by Rivest et al. and the original ElGamal scheme [34] realized the property of multiplicative HE. However, RSA and ElGamal schemes are considered not secure for most real world applications [35].

Several public-key encryption schemes are naturally homomorphic with respect to a single operation such as addition or multiplication [19, 33, 34]. An interesting question, initially posed by Rivest et al. [31], is whether there exists an encryption scheme that simultaneously permits the evaluation of both addition and multiplication on ciphertexts [36].

In 2009, the seminal work of Craig Gentry [37] affirmatively answered this long-standing question by demonstrating the first plausible FHE that is based on ideal lattices. It can evaluate arbitrary number of addition and multiplication operations. Since these two operations constitute a logically complete set of operation, the encryption scheme can evaluate any arbitrary function [36]. Given ciphertexts $c_1, \ldots, c_t$ that encrypt $m_1, \ldots, m_t$ with the scheme under some key, and given any efficiently computable function $f$, anyone can efficiently compute a ciphertext that encrypts $f(m_1, ..., m_t)$ under that key [38]. However, Gentry's

approach to FHE is computationally expensive by having a complexity of $\widetilde{O}(\lambda^7)$, where $\lambda$ is the security parameter of the scheme [38].

Homomorphic encryption is already quite useful in a number of applications including securing genomic privacy. Lauter et. al [13] showed how basic genomic algorithms (i.e. Pearson Goodness-of-Test, $D$ and $r^2$ measures of the linkage disequilibrium, Estimation Max Algorithm, and Cochran-Armitage Test for Trend) can be performed on encrypted data using practical homomorphic encryption. Yasuda et al. [39] gave a practical solution for computation of multiple Hamming distance values using the LNV scheme [13] on encrypted data to find the locations where a pattern occurs in a text. Cheon et al. [40] described how to calculate edit distance on homomorphically encrypted data. In the 2015 iDASH Privacy Workshop, Kim et al. [41] addressed the challenge of performing secure computation of: i) minor allele frequencies and $\chi^2$ statistics and ii) edit distance and hamming distance between two genome sequences using two practical HE scheme - BGV and YASHE scheme. Focusing on disease susceptibility test, Ayday et. al [42] developed an architecture between a patient and a medical unit (MU) and presented a privacy-preserving susceptibility test using Paillier scheme and proxy-re-encryption.

To date, software libraries that implement HE for bioinformatics, genomics, and other research purposes are emerging. In 2013, IBM released an open-source software library written in C++ that implements a fully homomorphic encryption called HELib [43]. This is an implementation of Brakerski-Gentry-Vaikuntanathan (BGV) scheme along with further SIMD-like optimizations known as ciphertext packing or batching [44]. In this scheme, each individual ciphertext element is conceptually a vector of encrypted plaintext integrals. This construction gave rise to a single instruction multiple data (SIMD) style operations that could particularly be effective with problems that benefit from some level of parallel computation. Although this library offers a scheme that can evaluate any circuit of arbitrary size, it has a trade-off of an extremely large overhead [45]. The size of generated

keys and ciphertexts are very large which substantially increases runtimes and makes homomorphic computation of complex functions impractical.

The Microsoft Research has also released a C++ library for homomorphic encryption which is called the Simple Encrypted Arithmetic Library (SEAL) [46]. It implements a practical homomorphic encryption (PHE) which is a combination of leveled homomorphic encryption, wherein parameters are set to allow a predetermined and fixed amount of computation on the data, together with application-specific data encodings and algorithmic optimizations [47]. PHE, though more restricted than FHE, yields a significant efficiency gain in both storage and computation. Even with the efficiency gain, however, this variant of HE scheme still exacts a high price for security. HE schemes that support both addition and multiplication operations (e.g. FHE and PHE) have a drawback of more complex system than that of partially HE schemes [48]. Implementation of such a cryptosystem even for basic operations still requires significantly more complicated computations and massive ciphertext sizes.

The Homomorphic Encryption Project (THEP) [49], on the other hand, is a library that implements Paillier scheme in Java. This scheme is a partially homomorphic encryption that exhibits additive homomorphism. Partial homomorphic cryptosystems like Paillier scheme do not have much overhead in performing the computations. With smaller expansion and lower cost compared with the other schemes, Paillier scheme has a considerable potential for real world applications [50]. The main drawback of this scheme, however, is the range of circuits that it supports; the type of operation that can be evaluated homomorphically in Paillier scheme is limited only to addition. Nevertheless, this scheme can still be efficiently useful in certain applications in which addition operation is already sufficient.

To evaluate the practicality of HELib, SEAL, and THEP, each of the library's storage requirement and computational complexity is assessed by performing encryption, decryption, and arithmetic operations (addition and/or multiplication) over 4-byte integers on Intel Core i3-2470M CPU with 2.40 GHz processor running

on Windows 10. The summary of results is shown in Tables 1 and 2.

| **HE Library** | **KeyGen** | **Enc** | **Add** | **Mult** | **Dec** |
|---|---|---|---|---|---|
| HELib | 57.7182 s | 0.5561 s | 0.0007 s | 0.2799 s | 2.5753s |
| SEAL | 45.0590 s | 0.4898 s | 0.0154 s | 1.3264 s | 0.7178 s |
| THEP | 90.7 ms | 4.7 ms | 0.0025 ms | - | 1 ms |

Table 1: Computational Complexities of HELib, SEAL, and THEP

| **HE Library** | **Public Key Size** | **Secret Key Size** | **Ciphertext Size** |
|---|---|---|---|
| HELib | 677.3 MB | 679.8 MB | 3.5 MB |
| SEAL | 162 KB | 162 KB | 162 KB |
| THEP | 32 bytes | 33 bytes | 19 bytes |

Table 2: Storage Requirements of HELib, SEAL, and THEP

# III. Theoretical Framework

## A. Genome-Wide Association Study (GWAS)

Genome-Wide Association Study (GWAS) is an approach that involves rapid scanning of genetic markers across the genomes of many people to find genetic variations associated with a particular phenotype (i.e. an individual's observable trait) [7]. In GWAS, genetic markers are DNA sequences with known physical locations on chromosomes that exhibit polymorphism due to insertion, deletion, and/or substitution of nucleotides [51]. Single Nucleotide Polymorphisms, often called as SNPs (pronounced as 'snips'), are the most commonly examined markers in GWAS [51]. These are single-nucleotide substitutions (i.e. 2 or more alleles exist at a specific locus) that occur in more than one percent of the general population [51, 52].

In most cases, GWAS is based on a case-control design in which SNPs across the human genomes in a case (i.e. phenotype positive) and control (i.e. phenotype negative) group are genotyped and subjected to statistical correlation analyses [4, 51]. Prior to statistical association methods, genetic markers must undergo quality control procedures to avoid potential false-negative and false-positive associations [53]. Compromised data quality in the discovery phase may lead to false positive results that are carried forward into replication studies at great cost both in time and expense. Among others, quality control of markers can be attained using Minor Allele Frequency (MAF) [52] and Hardy-Weinberg Equilibrium (HWE) [13]. The statistical power is low for detecting associations to genetic markers with low MAF [51, 53, 54]. Statistical power is the probability to reject a null hypothesis ($H_0$) while the alternative hypothesis ($H_A$) is true [55]. It is highly affected by many factors including MAFs, Linkage Disequilibrium (LD), disease prevalence and sample size [55, 56]. To avoid false negative associations, a statistical power of 80% is widely used in large-scale association studies [55]. HWE, on the other hand, is used to determine if a population at a given genetic marker is in equilibrium.

That is, its allele frequencies stay from the same generation to generation unless perturbed by evolutionary influences [13]. SNPs that substantially deviate from HWE are often excluded [13, 51].

After the quality control procedures, the selected SNPs will be tested for association using a statistical association test. Studies on qualitative traits can either have Chi-square test, Cochran-Armitage Trend test, or logistic regression, while linear regression can be used for quantitative traits [57]. The method for association testing could have series of replications and validations before an interpretation can be made [51].

To date, GWAS have already revealed numerous disease-associated loci including those which are related to complex diseases such as diabetes, heart abnormalities, Parkinson disease, and Crohn's disease [4, 8, 10]. These identified genetic associations are invaluable information in developing better strategies on disease diagnosis, treatment, and prevention.

## B. Minor Allele Frequency (MAF)

Minor allele frequency refers to the frequency of the less abundant (or minor) allele in the population at a variable site [52, 58, 59]. In Genome-Wide Association Studies (GWAS), the power to detect genetic effects is greatly influenced by MAFs [53, 54]. Loci with a low MAF ($< 10\%$) have significantly lower power to detect weak genotypic risk ratios than loci with a high MAF ($> 40\%$) [53]. Hence, MAF is one of the population genetics metrics that is necessarily checked prior to the selection of genetic markers in GWAS. It is usual to exclude low MAF variant sites, and the common thresholds in GWAS are between 1% and 5% [54].

For MAF computation [41], suppose that $A$ and $B$ are the two alleles of a given variant site. Let $N_{AA}$, $N_{AB}$, and $N_{BB}$ denote the numbers of observed individuals for genotypes $AA$, $AB$, $BB$, respectively. The allele counts of $A$ and $B$ are given by $N_A = 2N_{AA} + N_{AB}$ and $N_B = 2N_{BB} + N_{AB}$. Then, the MAF of the given

alleles is defined by

$$\frac{min(N_A, N_B)}{N_A + N_B}$$

Kim et. al [41] presented a method to encode the genotypes at a candidate SNP. Genotypes can be encoded as integers so that one can efficiently compute for the allele counts. More precisely, for a biallelic SNP with alleles $A$ and $B$, there are 3 possible genotypes - $AA$, $AB$, $BB$, which can be encoded as follows:

$$AA \to 2, AB \to 1, BB \to 0.$$

Thus, the counts of allele $A$ ($N_A$) at a given SNP can be computed by getting the summation of all the genotype encodings at that SNP while the counts of allele $B$ ($N_B$) can be computed as $N_B = 2N - N_A$, where $N$ is the total number of people in the sample population.

## C. Hardy-Weinberg Equilibrium (HWE)

Consider a biallelic SNP with alleles $A$ and $B$, and let $N_{AA}, N_{AB}, N_{BB}$ denote the observed population counts for genotype $AA, AB, BB$, respectively. Also, let $N$ be the total number of people in the sample population; that is, $N = N_{AA} + N_{AB} + N_{BB}$. Given this notation, the corresponding frequencies of genotype $AA, AB, BB$ are given by

$$p_{AA} = \frac{N_{AA}}{N}, p_{AB} = \frac{N_{AB}}{N}, p_{AA} = \frac{N_{BB}}{N}.$$

Moreover, the frequencies of the alleles $A$ and $B$ are given by

$$p_A = \frac{2N_{AA} + N_{AB}}{2N}, p_B = \frac{2N_{BB} + N_{AB}}{2N} = 1 - p_A$$

since each count of genotype $AA$ contributes two $A$ alleles, each count of genotype $BB$ contributes two $B$ alleles, each count of genotype $AB$ contributes one $A$ allele and one $B$ allele, and the total number of alleles in a sample of $N$ people is $2N$.

The population is said to be in HWE at the SNP if these frequencies are independent. That is,

$$p_{AA} = p_A^2, p_{AB} = 2p_A p_B, p_{BB} = p_B^2.$$

Researchers test for HWE as a way to test for data quality, and probably discard SNPs that have significant deviation from the equilibrium. Departure from the equilibrium can be indicative of potential genotyping errors [51]. If no technical errors are detected then a number of biologically plausible explanations exist such as population stratification or assortative mating and inbreeding [51, 54].

Testing for deviations from HWE can be carried out using Pearson Goodness-Of-Fit Test, often known simply as $\chi^2$ test [54]. If alleles frequencies are independent, then the observed counts can be expected to be

$$E_{AA} = Np_A^2, E_{AB} = 2Np_A p_B, E_{BB} = Np_B^2.$$

Thus, the test statistic can be computed as follows:

$$\chi^2 = \sum \frac{(N_i - E_i)^2}{E_i}, \text{where } i \in \{AA, AB, BB\}.$$

The obtained test statistic value can be used to determine the $P$-value either by referring to the Chi-square distribution table or by using a Chi-square distribution calculator, and decide whether or not to discard a particular SNP from the dataset. In most GWAS, deviation tests for HWE are rejected at $P$-values less than $10^{-4}$ [51].

## D. $\chi^2$ Test Statistic

One of the classic statistical methods that are commonly used to measure the genotype-phenotype association in GWAS is the Chi-Squared ($\chi^2$) Test [60, 41]. It is highly reliable and easy to calculate for large sample sizes [60]. The $\chi^2$ test

statistic in case-control groups is computed based on the allelic contingency table, where $N_A$ and $N_B$ are the counts of allele types $A$ and $B$ in the case group, $N'_A$ and $N'_B$ are the counts of allele types $A$ and $B$ in the control group, and $N$ is the total number of people in the sample population. [41].

$$\frac{T(N_A N'_B - N'_A N_B)}{R \cdot S \cdot G \cdot K}.$$

Further, it can be written as a function of $N_A$ and $N'_A$ and be expressed as

$$\frac{4N(N_A(2N - N'_A) - N'_A(2N - N_A))^2}{2N \cdot 2N \cdot G \cdot K}$$

$$= \frac{4N(N_A - N'_A)^2}{(N_A + N'_A) \cdot (4N - (N_A + N'_A))}$$

| | Allele Type | | **Total** |
|---|---|---|---|
| | ***A*** | ***B*** | |
| Case | $N_A$ | $N_B$ | $R = 2N$ |
| Control | $N'_A$ | $N'_B$ | $S = 2N$ |
| Total | $G = N_A + N'_A$ | $K = N_B + N'_B$ | $T = 4N$ |

Table 3: Allelic Contingency Table

## E. Homomorphic Encryption

Homomorphic encryption (HE) describes a class of encryption algorithms which satisfy the homomorphic property: that is certain mathematical operations can be carried out on ciphertexts directly so that upon decryption, the same answer is obtained as operating on the original messages [45]. This is a special property not possessed by standard encryption schemes where decrypting the sum of two ciphertexts would generally render nonsense.

More precisely, an encryption scheme is said to be homomorphic for some operations acting in message space (e.g. addition) if there are corresponding operations acting in ciphertext space satistying the property [45]:

$$\text{Dec}(\text{Enc}(m_1) \circ \text{Enc}(m_2)) = \text{Dec}(\text{Enc}(m_1 \diamond m_2))$$

Note that the property does not commute when starting instead from ciphertexts due to semantic security. In general, the same message encrypts to different ciphertexts with high probability [45]:

$$\text{Enc}(m_1) \circ \text{Enc}(m_2) \neq \text{Enc}(m_1 \diamond m_2)$$

Several encryption schemes allow partial homomorphic encryption; That is, users are allowed to perform some mathematical functions on encrypted data, either addition or multiplication, but not both.

An additively homomorphic scheme is one with a ciphertext operation that results in the sum of the plaintexts [38, 18]. That is,

$$\text{Dec}(\text{Enc}(m_1) \circ \text{Enc}(m_2)) = \text{Dec}(\text{Enc}(m_1 + m_2))$$

On the other hand, a multiplicatively homomorphic scheme is one that has an operation on two ciphertexts that results in the product of the plaintexts [38, 18]. That is,

$$\text{Dec}(\text{Encrypt}(m_1) \circ \text{Encrypt}(m_2)) = \text{Dec}(\text{Encrypt}(m_1 * m_2))$$

An encryption scheme is fully homomorphic if it can handle both addition and multiplication operations [18, 38, 45].

## F. Paillier Cryptosystem

The Paillier cryptosystem [19] is a public key encryption scheme created by Pascal Paillier in 1999, with several interesting properties. Public key cryptography is the use of asymmetric key algorithms, where the key used to encrypt a message is not the same as the key used to decrypt it [38]. Public key cryptographic schemes

generate two cryptographic keys - a public key and a private key. The private key is kept secret, while the public key may be widely distributed. The keys are related mathematically, but the private key cannot be feasibly derived from the public key [38].

The Paillier cryptosystem is composed of three algorithms [18]:

**Algorithm 1** KeyGen$(p, q)$

Input: $p$ and $q$ should be prime

Compute: $n = pq$, $\lambda = \text{lcm}(p-1, q-1)$

Choose: $g | \gcd(g, n^2) = 1$ and $\gcd(L(g^\lambda \mod n^2), n) = 1$, where $L(u) = (u-1)/n$

Compute: $\mu = L(g^\lambda \mod n^2)^{-}1$

Output: $(P_k, S_k)$

Public Key: $P_k = (n, g)$

Secret Key: $S_k = (\lambda, \mu)$

**Algorithm 2** Encrypt$(m, P_k)$

Input: $m \in \mathbb{Z}_n$

Choose: $r | \gcd(r, n) = 1$

Compute: $c = g^m r^n \mod n^2$

Output: $c$

**Algorithm 3** Decrypt$(c, S_k)$

Input: $c$

Compute: $L(c^\lambda \mod n^2) \cdot \mu \mod n$

Output: $m \in \mathbb{Z}_n$

A notable feature of the Paillier cryptosystem is its additively homomorphic properties. Specifically, the following identities can be described [18]:

1. The product of two ciphertexts will decrypt to the sum of their corresponding plaintexts. That is,

$$\text{Dec}(\text{Enc}(m_1) \cdot \text{Enc}(m_2) \mod n^2) = (m_1 + m_2) \mod n$$

2. A ciphertext raised to the power of another plaintext will decrypt to the product of the two plaintexts. That is,

$$\text{Dec}(\text{Enc}(m_1)^{m_2} \mod n^2) = (m_1 m_2) \mod n$$

## G. The Homomorphic Encryption Project (THEP)

THEP [49] provides a homomorphic encryption library that currently implements Paillier cryptosystem in Java. Saving/transporting keys and encrypted integers can be accomplished through Serialization - a mechanism in Java where an object can be represented as a sequence of bytes that includes the object's data as well as information about the object's type and the types of data stored in the object.

The Pallier scheme overcomes the restriction of the message space to be $\mathbb{Z}_2$ and augments the message space to $R = (\mathbb{Z}_n, +, *)$ [45], which is apparent in THEP. To encrypt strings, however, one could consider converting each character of the string to its corresponding ASCII code value.

In performing addition, an instance $a.add(b)$ is not really adding anything, but actually computes $(a * b) \mod n^2$ [49]. This does, however, result in the addition of the plaintexts. The users of the library need not know how the underlying homomorphic cryptosystem works and continue to think as if they are working on plaintext integers.

THEP can handle negative number such as adding two encrypted negatives or multiplying an encrypted integer with a negative number [49]. Working with negative numbers is very straight forward as long as the result of the operation is positive. However, there are cases that will result to a negative value. In this case, the decrypted output will be a very large number not equal to the negative expected value but rather equal to $-(expectedValue) \mod N$. In such a case, certain threshold value (application specific) can be assigned and if the decrypted integer is greater than that, it is to be assumed the result value should have been negative. This will work with very high probability as long as a good threshold is chosen.

# IV. Design and Implementation

## A. Use Cases

The user can load dataset(s), perform genomic computations including MAF, HWE, and $\chi^2$ Test Statistic, and view the computation results. Figure 1 shows the top level use case diagram of the computing tool.

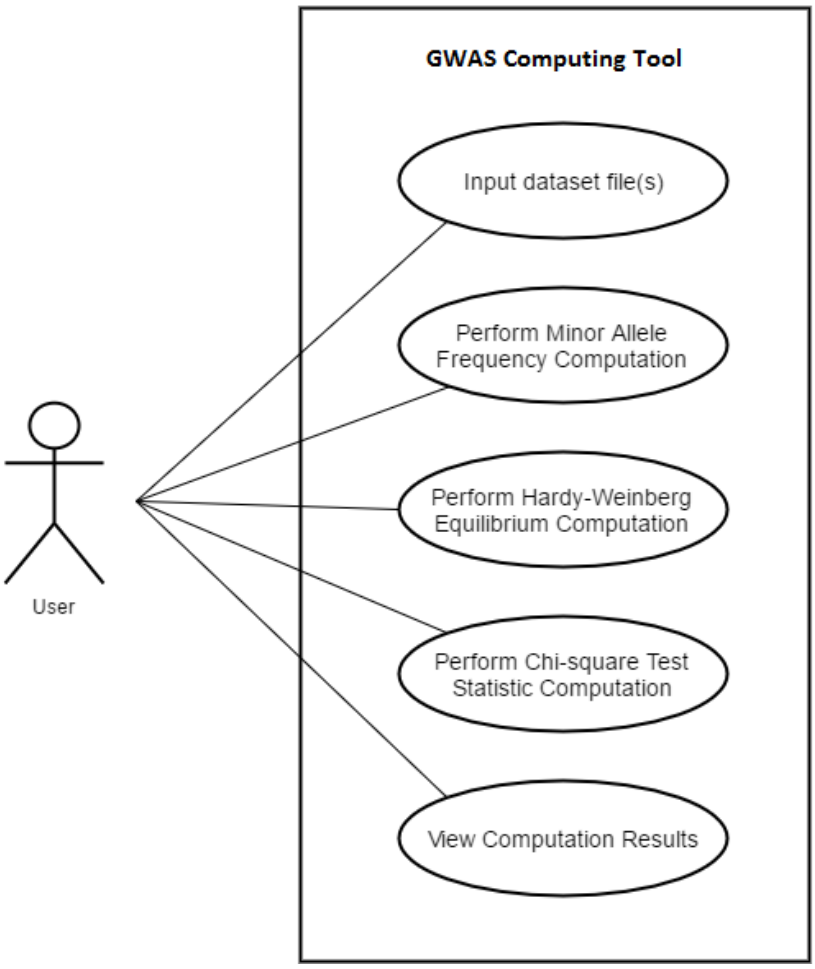


Figure 1: Top Level Use Case Diagram

The dataset(s) to be loaded must be in a text file (the specific file format will be discussed in the following subsection). In performing the $\chi^2$ test statistic, there are two required datasets, namely, the case and the control datasets. The user may compute the Minor Allele Frequency, Hardy-Weinberg Equilibrium, and $\chi^2$ test statistic of the datasets that he/she loaded into the computing tool. The genomic computations involve the following processes: encoding and encryption

of the datasets, uploading of the encrypted dataset to the server, downloading of encrypted results, and decryption of the results. When the selected computation is done, the user will be able to view the computation results in plaintext format.

## B. Security Model

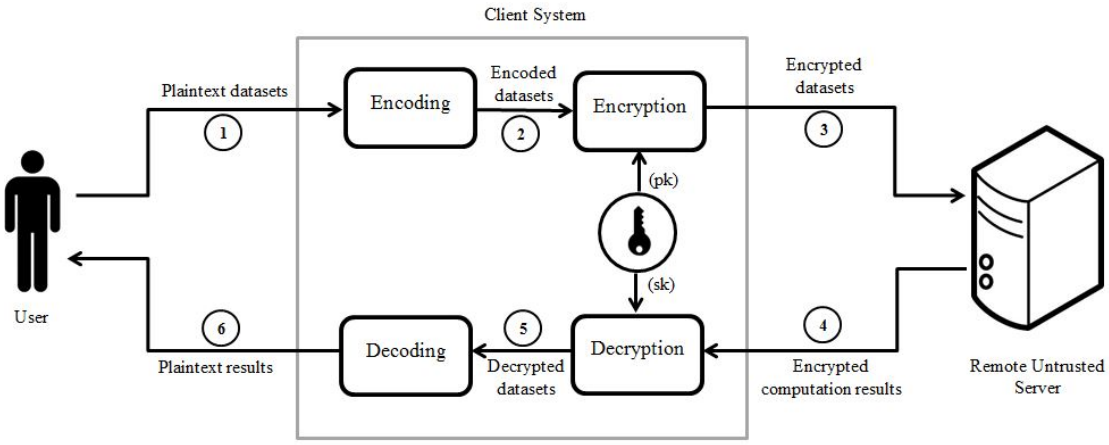


Figure 2: The security model of the computing tool

Figure 2 shows the security model of the computing tool. It consists of data pre-processing, data computation, and result processing. The pre-processing of data includes encoding each genotype sample in the dataset into its corresponding integer value. The encoding process is followed by encryption that is performed using THEP. Both the encoding and the encryption processes are done on a client machine.

The data computation phase is carried out on a remote server. The encrypted datasets are uploaded to the server that performs the pre-selected type of computation. Since the encryption scheme used to encrypt the datasets possesses additive homomorphic properties, the server can perform all the necessary additions over the encrypted datasets. The (encrypted) computation results are then sent back to the client side.

Once the results are transmitted back to the client side, the computing tool decrypts and decodes these results to plaintext format before returning them to

the user. The decryption process can only be done by the party that holds the private key, which is supposedly the intended receiver of the encrypted message. In this case, the client (research institution) is the only party that can decrypt the data (i.e. datasets, computation results) back to its plaintext format.

## C. GWAS Dataset

In our work, we use the GWAS datasets from the 2015 iDASH Healthcare Privacy Protection challenge, which consists of 311 SNPs located on human chromosome 2 from 200 PGP participants. Figure 3 shows a snapshot of the text file format of the dataset released by iDASH.

```
#case000 #case001 #case002 #case003 #case004 #case005 #case006 #case007 #case008 #case009 #case010 #case011 #case012
rs11686243
AG AG AA AG GG AA AG AA GG AG AA AA AA AA AA AA AA GG GG AG AG AA AG GG AA AA GG AG AG AG GG AG AA AA AG AG AG AG AG
rs4426491
CC CC CC CT CT CC CT CC CT CT CC CC CC CC CC CC CC CT CT CT CC CC CT CT CC CC CT CC CT CC CT CC CC CC CT CC CC CT CT
rs4305230
CC CC CC CT CT CC CT CC TT CT CC CC CC CC CC CC CC TT TT CT CC CC CT CT CC CC CT CC CT CC TT CC CC CC CT CC CC CT CT
rs4630725
GG GG GG AG AG GG AG GG AG GG GG GG GG GG GG GG GG AA AG AG GG GG GG AG GG GG AG GG AG GG AA GG GG GG AG GG GG AG AG
rs6709634
CT TT TT CT CC TT CT TT CC CT TT CT CT TT TT TT TT CC CC CT CT TT CC CC TT TT CT TT CT TT CC TT TT CT CC TT CT CT CT
rs7561914
GG GG GG AG GG GG AG GG AG AG GG GG GG GG GG GG GG AA AG AG GG GG GG AG GG GG AG GG AG GG AG GG GG GG GG GG GG AG GG
rs6547957
AA AA AA AG AA AA AG AA AA AA AA AA AA AA AA AA AA AG AA AG AA AA AA AG AA AA AG AA AA AA AA AA AA AA AA AA AG AA
rs6733973
CC CC CC CT CC CC CT CC CC CC CC CT CT CC CC CT CC CT CC CT CC CC CC CT CC CC CC CC CC CC CC CC CC CT CC CC CC CC CC
rs4666254
CC CC CT TT CC CC CT CC CT CT CC TT CT CT CC CT CC TT CT CT CT CC CC TT CC CC CC CC CC CC CC CC CC CT CT CC CT CC CC
rs4666255
GG GG GG AG GG GG AG GG GG GG GG AG AG GG GG AG GG AG GG AG GG GG GG AG GG GG GG GG GG GG GG GG AG GG GG GG GG GG
rs9973865
CC CC CT TT CC CC CT CC CT CT CC TT CT CT CC CT CC TT CT CT CT CC CC TT CC CC CC CC CC CC CC CC CC CT CT CC CT CC CC
rs7573958
TT TT CT CC TT TT CT TT CT CT TT CC CT CT TT CT TT CC CT CT CT TT TT CC TT TT TT TT TT TT TT TT TT CT CT TT CT TT TT
rs7576817
TT TT CT CC TT TT CT TT CT CT TT CT TT CT TT CC CT CT TT TT TT CT TT TT TT TT TT TT TT TT TT TT TT TT TT TT TT TT
rs10166469
CC CC CT TT CC CC CT CC CT CT CC TT CT CT CC CT CC TT CT CT CT CC CC TT CC CC CC CC CC CC CC CC CC CT CT CC CT CC CC
```

Figure 3: A snapshot of the dataset released by iDASH

The header line indicates the case/control number of the genotype samples in each SNP; this also indicates the number of genotype samples that are included in the dataset. Each entry of SNP comes in two rows: the first row indicates the RSID (a unique identifier) of the SNP, and the second row contains the genotype samples of the SNP. Each column of the SNPs genotype samples contains the genotype of a DNA donor, and this sample entry corresponds to the case/control

number indicated in the header line.

The file format released by iDASH contains less variant information than the standard file format for storing variants - Variant Call Format (VCF) [61]. One may convert a VCF file to the format issued in iDASH by removing all the meta-information lines, changing the contents of the header line to the case/control number of the genotype samples, and extracting only the SNP ID (e.g., RSID) and the genotype samples for each data line. The `GT` under the `FORMAT` column is a reserved keyword that represents genotype. The genotype samples of that particular variant are in the columns next to the `FORMAT` column which are encoded as allele values separated by / or |. An allele value of 0 means that the allele of the genotype is equal to the reference allele (REF); 1 means that the allele is equal to alternative allele (ALT); and 2 means that the allele is equal to the second allele listed in ALT (if it exists). For instance, if the reference allele is A and the alternative allele is G, a genotype sample tagged with GT=1/1 represents a genotype GG (1/0 - GA, 0/0 - AA).

## D. Homomorphic Computation

### 1. MAF Computation on encrypted genomic data

Sample genotypes of a given candidate SNP associated with a specified disease can be encoded as integers in such a way that the counts of alleles may be computed efficiently. More precisely, for a biallelic SNP with alleles $A$ and $B$, there are three possible genotypes - $AA$, $AB$, $BB$, which may be encoded as follows [41]:

$$AA \to 2, AB \to 1, BB \to 0.$$

Using the method of encoding described above, the counts of the alleles may be obtained using homomorphic additions. Let $c_i$ be the encrypted value of the $i$th sample genotype of SNP site $j$, and $N$ be the total number of people in the sample population. Additionally, suppose $c_{eval}$ is the ciphertext given by the

homomorphic operation defined by

$$\mathrm{c}_{eval} = \sum_{i=1}^{N} \mathrm{c}_i.$$

Since addition is the only allowed homomorphic operation, the server returns $\mathrm{c}_{eval}$ back to the client side where the remaining operations is performed. On the client side, if $\mathrm{m} = \mathrm{Dec}(\mathrm{c}_{eval}, \mathrm{sk})$ denotes the decryption of the ciphertext $\mathrm{c}_{eval}$, then the MAF of SNP $j$ in the dataset can be computed as

$$\frac{\min(\mathrm{m}, 2N - \mathrm{m})}{2N}.$$

**2. HWE Computation on encrypted genomic data**

There is a different encoding process for HWE computation since it requires obtaining the genotype counts instead of the allele counts. Given a biallelic SNP with alleles $A$ and $B$, the three possible genotypes $AA$, $AB$, $BB$ are assigned 0,1,2, respectively. The $i$th sample genotype of SNP site $j$ can be encoded through the encodings $e_0$,$e_1$, and $e_2$ which returns 1 if the entry value is the same as the value that the encoding represents, and 0 otherwise. More specifically, the three possible genotypes can be encoded as follows:

$$\begin{aligned}
AA \text{ (value 0):} \quad & {e_0}^{(i,j)} \leftarrow 1 \quad {e_1}^{(i,j)} \leftarrow 0 \quad {e_2}^{(i,j)} \leftarrow 0 \\
AB \text{ (value 1):} \quad & {e_0}^{(i,j)} \leftarrow 0 \quad {e_1}^{(i,j)} \leftarrow 1 \quad {e_2}^{(i,j)} \leftarrow 0 \\
BB \text{ (value 2):} \quad & {e_0}^{(i,j)} \leftarrow 0 \quad {e_1}^{(i,j)} \leftarrow 0 \quad {e_2}^{(i,j)} \leftarrow 1
\end{aligned}$$

Using this encoding method, the genotype counts can be computed through homomorphic additions. The (encrypted) counts ${N_k}^{(j)}$ of value-$k$ genotype at locus $j$ can be computed by summing all the ciphertexts ${c_k}^{(i,j)}$, that is,

$${N_0}^{(j)} = \sum_{i=1} {c_0}^{(i,j)}, {N_1}^{(j)} = \sum_{i=1} {c_1}^{(i,j)}, {N_2}^{(j)} = \sum_{i=1} {c_2}^{(i,j)}.$$

Finally, the (encrypted) total number $N^{(j)}$ can be computed by summing all the

${N_k}^{(j)}$, that is,

$$N^{(j)} = {N_0}^{(j)} + {N_1}^{(j)} + {N_2}^{(j)}$$

Since the genotype frequencies cannot be computed using homomorphic addition, the statistical algorithm for HWE must be modified to use the genotype counts instead. Recall that for a single locus, the Pearson test computes the test statistic as

$$\chi^2 = \sum_{i=0}^{2} \frac{(N_i - E_i)}{E_i},$$

where $N_i$ is the observed genotype count, and $E_i$ is the expected genotype count. Alternatively, the expected genotype count can be computed as

$$E_0 = N\left(\frac{2N_0 + N_1}{2N}\right)^2, E_1 = 2N\left(\frac{2N_0 + N_1}{2N}\right)\left(\frac{2N_2 + N_1}{2N}\right), E_2 = N\left(\frac{2N_2 + N_1}{2N}\right)^2,$$

which can be simplified to

$$E_0 = \frac{(2N_0 + N_1)^2}{4N}, E_1 = \frac{(2N_0 + N_1)(2N_2 + N_1)}{2N}, E_2 = \frac{(2N_2 + N_1)^2}{4N}.$$

The test statistic $\chi^2$ can then be computed as

$$\chi^2 = \frac{(4N_0N_2 - {N_1}^2)^2}{2N}\left(\frac{1}{2(2N_0 + N_1)^2} + \frac{1}{(2N_0 + N_1)(2N_2 + N_1)} + \frac{1}{2(2N_2 + N_1)^2}\right).$$

Since it is not allowed to perform homomorphic divisions, the server returns the encryptions $N_0$,$N_1$,$N_2$,$\beta_1$,$\beta_2$ back to the client, where are defined by

$$\beta_1 = (2N_0 + N_1),\ \beta_2 = (2N_2 + N_1).$$

The test statistic can now be computed on the client side as

$$\chi^2 = \frac{(4N_0N_2 - {N_1}^2)^2}{2N}\left(\frac{1}{2(\beta_1)^2} + \frac{1}{(\beta_1)(\beta_2)} + \frac{1}{2(\beta_2)^2}\right).$$

**3. $\chi^2$ Test Statistic Computation on encrypted genomic data**

The case and control datasets in $\chi^2$ test statistic computation can be encoded similarly to the encoding method used in MAF computation wherein given a biallelic SNP with alleles $A$ and $B$, the three possible genotypes $AA$, $AB$, $BB$ are encoded as:

$$AA \to 2, AB \to 1, BB \to 0.$$

Recall that the statistical algorithm for the computation of $\chi^2$ test statistic can be written as a function of $N_A$ and $N'_A$, where $N_A$ is the counts of allele $A$ in the case dataset and $N'_A$ is the counts of allele $A$ in the control dataset. It can be expressed as

$$\chi^2 = \frac{4N(N_A - N'_A)^2}{(N_A + N'_A) \cdot (4N - (N_A + N'_A))}.$$

Using the method of encoding described above, the counts of the allele A in case and control datasets ($N_A$,$N'_A$) can be obtained using homomorphic additions. Since it is not allowed to perform homomorphic multiplications and divisions, the server will return the encryptions $\alpha_1$ and $\alpha_2$ back to the client, where

$$\alpha_1 = N_A - N'_{A,2} = N_A + N'_A.$$

From these, the test statistic can be computed on the client side as

$$\chi^2 = \frac{(4N(\alpha_1))^2}{(\alpha_1)(4N - \alpha_2)}.$$

## E. Technical Architecture

The server machine that will perform the necessary homomorphic computations should have the following requirements:

1. Server Java SE Runtime Environment 7
2. 1GB RAM
3. 2.4 MHz single core processor

The computing tool should be running on a machine with the following requirements:

1. Java SE Runtime Environment 7
2. 8GB RAM
3. Intel(R) Core(TM) i3-2370M CPU @ 2.40GHz
4. 64-bit Windows 10

# V. Results

Figure 4 shows the user interface of the GWAS Computing Tool. At the top section, the user can specify the IP address and the port number of the server where the genomic computations are to be outsourced. In the computation setting panel, there is a dropdown box that allows the user to choose the type of computation he/she intends to perform; and there also are two buttons which allow the user to load the file(s) containing the datasets. Once the computation type and the datasets are chosen, the user may click on the Compute button to perform the calculations. The user can view the computation results at the results board, while the computation status is displayed at the console.

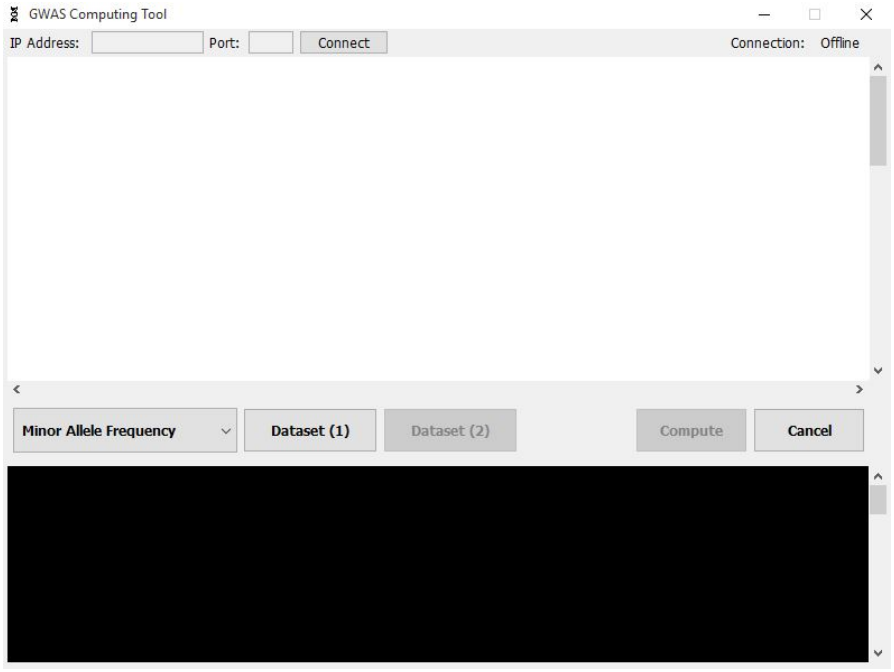


Figure 4: User Interface of the GWAS Computing Tool

The following screenshots show all the functionalities that the user is allowed to perform in the application.

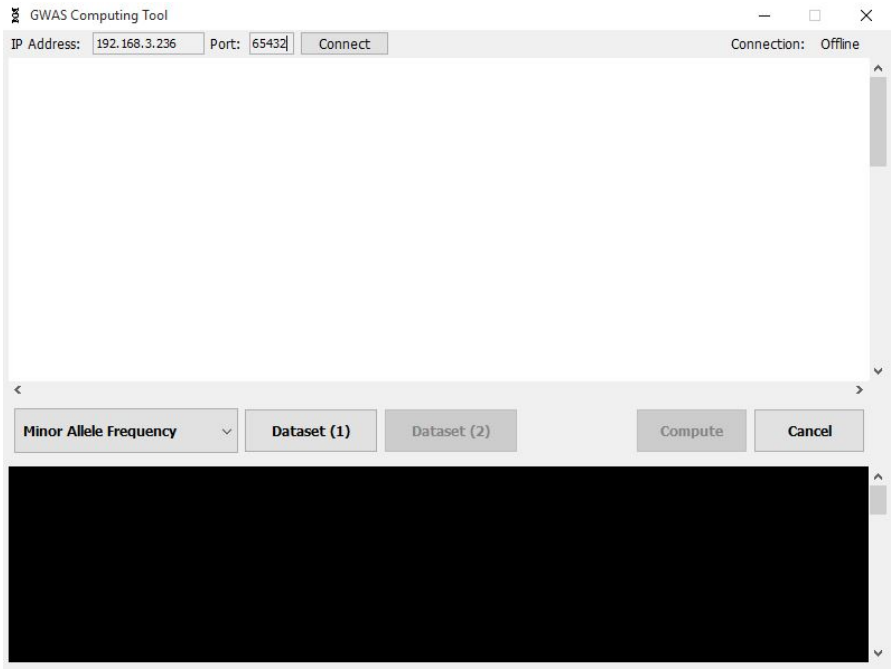


Figure 5: Connect to server

The user enters the IP address and the port number of the server that will perform the homomorphic computations through the text fields at the upper left section.

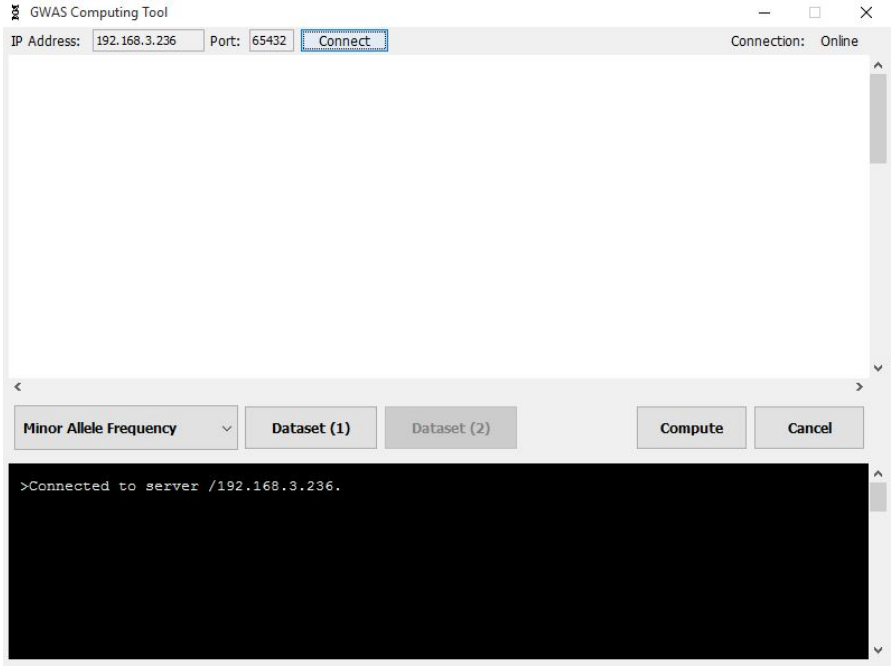


Figure 6: View connection status

The connection status is shown at the upper right section. The console will also notify the user once the connection is established.

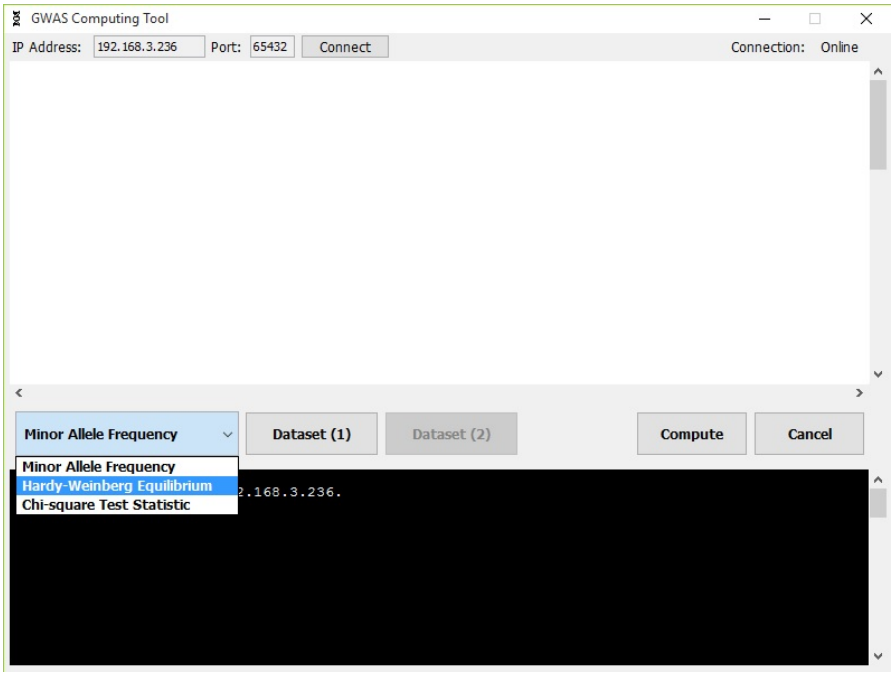


Figure 7: Select MAF computation

The user can select the genomic computation of interest through the drop-down box at the computation setting panel. For instance, the user selects MAF computation.

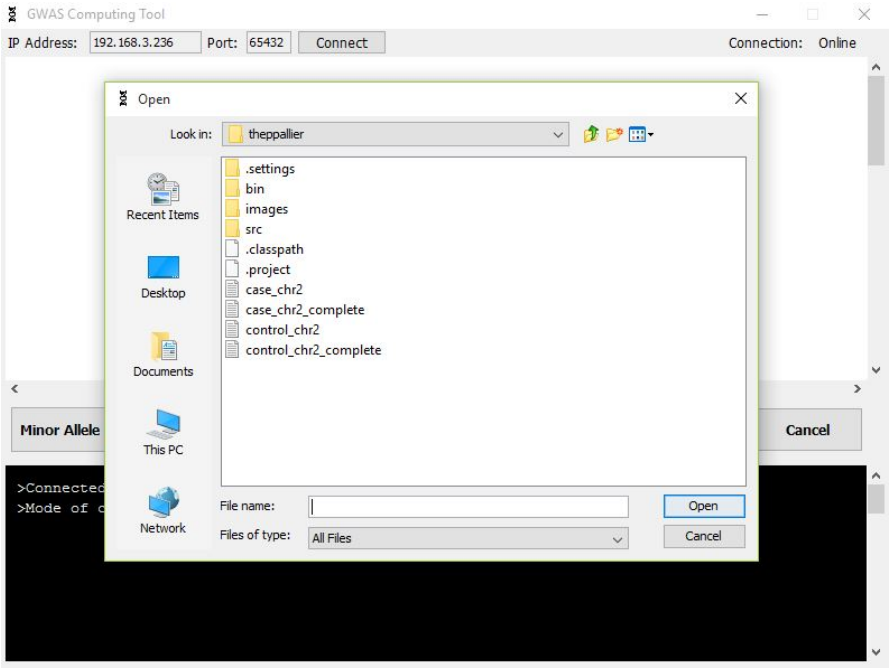


Figure 8: Select file(s) containing datasets

The user can select the file that contains the dataset for MAF computation through the button next to the dropdown box.

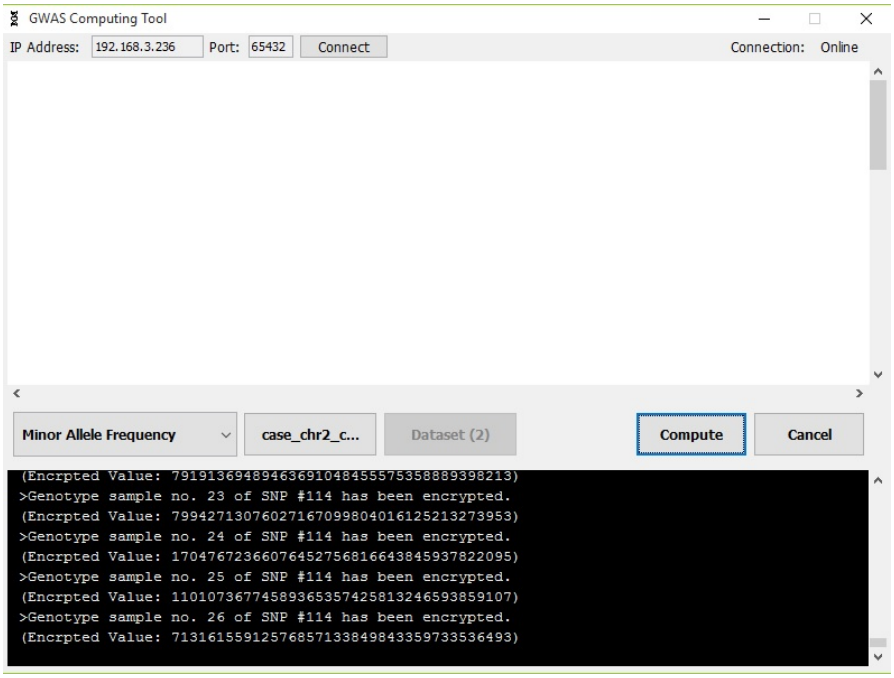


Figure 9: Start MAF computation

Once the computation details are finalized, the user can press the Compute

button to start the calculations. The computation status is shown in the console including the encoding process, encryption, uploading, and downloading of results.

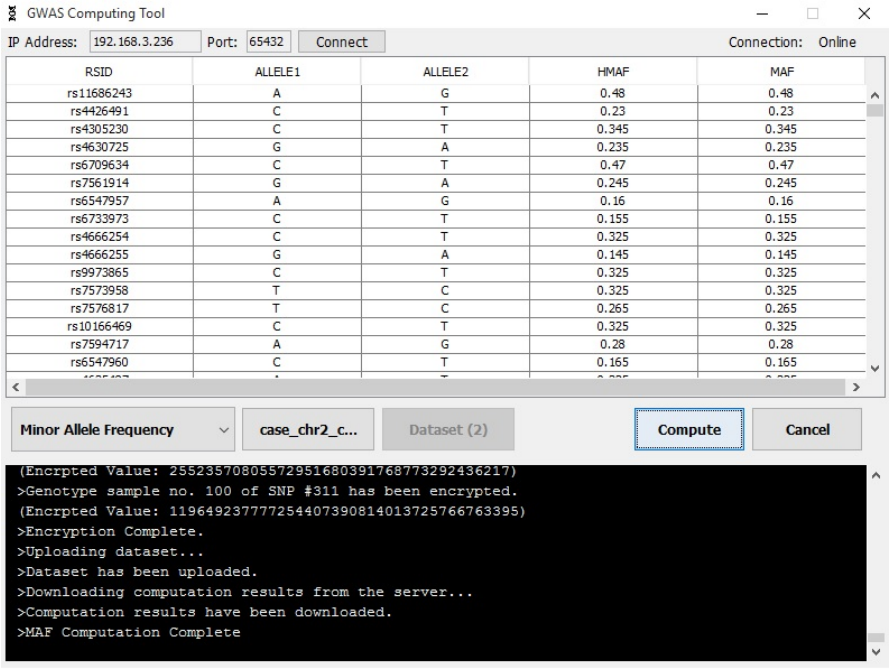


Figure 10: View the MAF computation results

The computation results are shown at the results board. The table contains the SNPs RSID, alleles, and computed MAF.

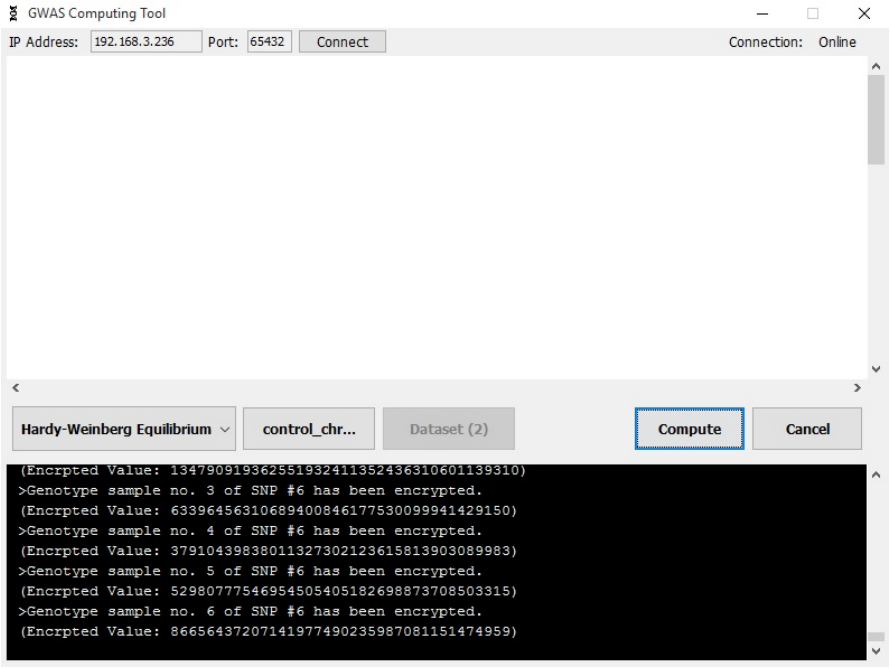


Figure 11: Select HWE computation

The user may also opt to compute for HWE. Pressing the start button will start the computation process that can be kept track through the console.

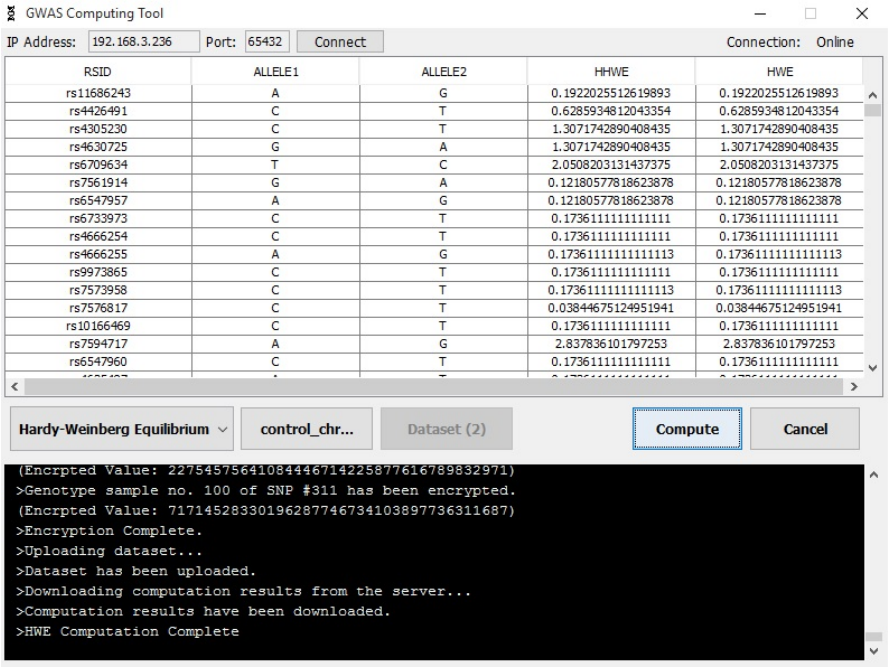


Figure 12: View the HWE computation results

The HWE computation results are shown at the results board. The table contains the SNPs RSID, alleles, and computed HWE.

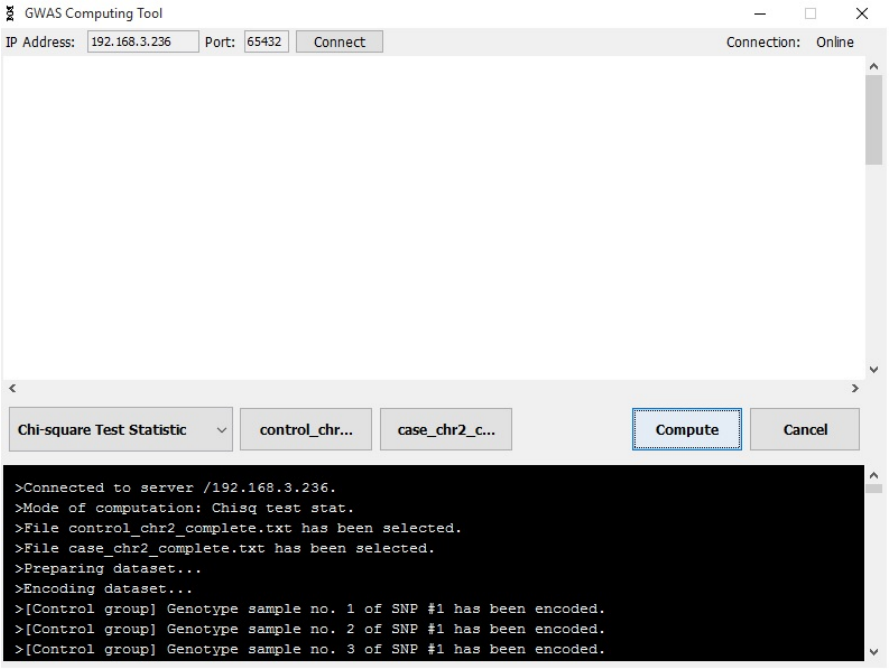


Figure 13: Select $\chi^2$ test statistic computation

Lastly, the user can compute for the $\chi^2$ test statistic. This requires the selection of two files: the file containing the control dataset, and the other file containing the case dataset. Once the Compute button is pressed, the $\chi^2$ test statistic computation will start. The whole computation process will be shown in the console.

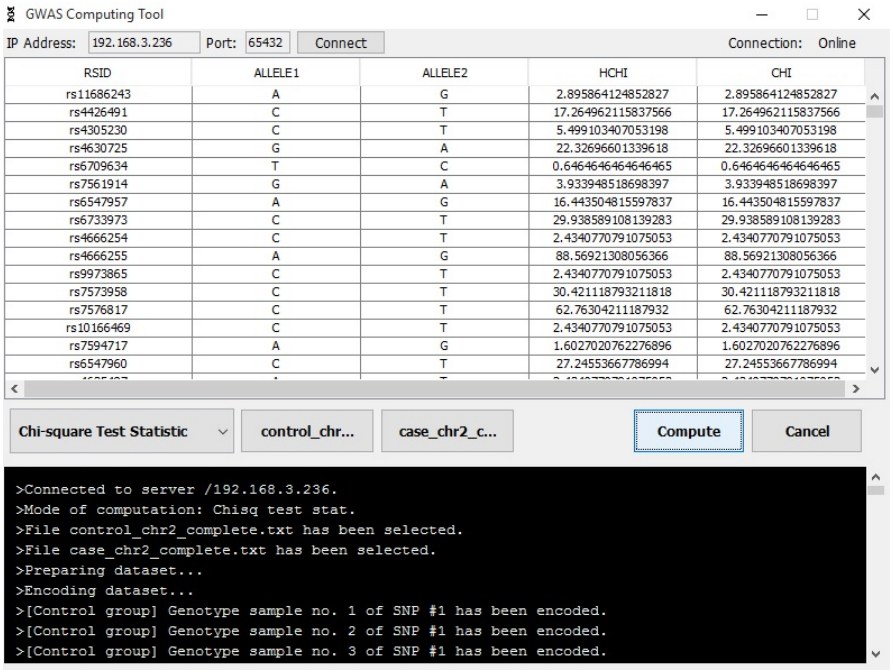


Figure 14: View the $\chi^2$ test statistic computation results

The $\chi^2$ test statistic computation results are shown at the results board. The table contains the SNPs RSID, alleles, and computed $\chi^2$ test statistic.

# VI. Discussions

The GWAS computing tool is a desktop application built on Java that provides a way for users to delegate the large computational overheads of performing genomic computations to a third-party server without compromising the confidentiality of the genomic datasets. It can perform three GWAS computations: Minor Allele Frequency, Hardy-Weinberg Equilibrium, and $\chi^2$ test statistic.

The tool used the Paillier cryptosystem as its homomorphic encryption scheme, and a Java-based library called THEP was used for its implementation. In THEP, the parameters of private key (i.e., $\lambda$ and $\mu$) and public key (i.e., $n$ and $g$) are encapsulated on its `PrivateKey` and `Publickey` classes respectively. The keys can be generated as follows:

```
int n = 64; //n is the key size
PrivateKey sk = new PrivateKey(n);
PublicKey pk = sk.getPublicKey();
```

The generated public key is used to encrypt every genotype sample of each SNP in the dataset. The encryption can be performed using the following code:

```
static void encryption(ArrayList<Snp> snps, PublicKey pk)
   ↪ throws BigIntegerClassNotValid{
     for(int i = 0; i < snps.size(); i++){
        for(int j = 0; j < snps.get(i).encodedSamples.size();
           ↪ j++){
          BigInteger code = new
             ↪ BigInteger(snps.get(i).encodedSamples.get(j).toString());
          EncryptedInteger enc = new EncryptedInteger(code, pk);
          snps.get(i).encryptedSamples.add(enc);
        }
     }
  }
```

The cipher values of the encrypted datasets (which are of class String) are transmitted to the server that will perform the homomorphic computations.To perform the homomorphic computation, the server should reconstruct the cipher values back to its equivalent `EncrypedInteger` value using the public key. Constructing an `EncrytedInteger` object using the cipher value is performed using the following code:

```
ArrayList<ArrayList<EncryptedInteger>> SNPsEncSamples = new
   ↪ ArrayList<ArrayList<EncryptedInteger>>();
for(int i = 1; i < SNPsCipherVal.size(); i++){
  ArrayList<EncryptedInteger> container = new
     ↪ ArrayList<EncryptedInteger>();
  for(int j = 0; j < SNPsCipherVal.get(i).size(); j++){
    EncryptedInteger encInt = new EncryptedInteger(pk);
    encInt.setCipherVal(new
       ↪ BigInteger(SNPsCipherVal.get(i).get(j)));
    container.add(encInt);
  }
  SNPsEncSamples.add(container);
}
```

Since the type of operation that the server can homomorphically evaluate is limited to addition, only partial computations can be done; multiplications and divisions of the computation must be done on the client side. In computing for MAF, the server can only compute for the required allele counts. Dividing the minimum allele count by the total allele count is performed on the client side. The code for MAF computation is as follows:

```
//MAF computation (server side)
static ArrayList<String>
   ↪ privateMAFComputation(ArrayList<ArrayList <String>>
   ↪ SNPsCipherVal, PublicKey pk) throws
```

```
    ↪ BigIntegerClassNotValid, PublicKeysNotEqualException{
   ArrayList<String> result = new ArrayList<String>();
   Integer totalAllele = 2*SNPsEncSamples.get(i).size();
   esAlleleCount =
      ↪ efAlleleCount.multiply(pk.getN().subtract(new
      ↪ BigInteger("1")));
   esAlleleCount = esAlleleCount.add(new
      ↪ BigInteger(totalAllele.toString()));
   result.add(efAlleleCount.getCipherVal().toString());
   result.add(esAlleleCount.getCipherVal().toString());
   }
   return result;
}
//MAF computation (client side)
for(int i = 0; i < snps.size(); i++){
   snps.get(i).dfAlleleCount = dfAlleleCount.doubleValue();
   snps.get(i).dsAlleleCount = dsAlleleCount.doubleValue();
   BigDecimal dtotalAllele = new
      ↪ BigDecimal(2*(snps.get(i).encryptedSamples.size()));

   if(dfAlleleCount.compareTo(dsAlleleCount) == -1){
      snps.get(i).emaf =
         ↪ dfAlleleCount.divide(dtotalAllele).doubleValue();
   }else{
      snps.get(i).emaf =
         ↪ dsAlleleCount.divide(dtotalAllele).doubleValue();
   }
}
```

In HWE computation, the server computes for the genotype counts (`alpha`, `beta`, `delta`) and the value of `omega` and `theta` (Refer to chapter 4). The

client side performs all the multiplications/divisions of its formula. The code for HWE computation is as follows:

```
//HWE computation (server side)
static ArrayList<String>
   ↪ privateHWEComputation(ArrayList<ArrayList <String>>
   ↪ SNPsCipherVal, PublicKey pk) throws
   ↪ BigIntegerClassNotValid, PublicKeysNotEqualException{
     for(int i = 0; i < SNPsEncSamples.size(); i++){
        EncryptedInteger alpha = new EncryptedInteger(new
           ↪ BigInteger("0"), pk);
        EncryptedInteger beta = new EncryptedInteger(new
           ↪ BigInteger("0"), pk);
        EncryptedInteger delta = new EncryptedInteger(new
           ↪ BigInteger("0"), pk);
        EncryptedInteger omega = new EncryptedInteger(new
           ↪ BigInteger("0"), pk);
        EncryptedInteger theta = new EncryptedInteger(new
           ↪ BigInteger("0"), pk);

        for(int j = 0; j < SNPsEncSamples.get(i).size()-2; j++){
          alpha = ealpha.add(SNPsEncSamples.get(i).get(j)); j++;
          beta = ebeta.add(SNPsEncSamples.get(i).get(j)); j++;
          delta = edelta.add(SNPsEncSamples.get(i).get(j));
        }

        omega = alpha.multiply(new BigInteger("2"));
        omega = omega.add(beta);
        theta = delta.multiply(new BigInteger("2"));
        theta = theta.add(beta);

        result.add(alpha.getCipherVal().toString());
```

```
        result.add(beta.getCipherVal().toString());
        result.add(delta.getCipherVal().toString());
        result.add(omega.getCipherVal().toString());
        result.add(theta.getCipherVal().toString());
      }
      return result;
    }
//HWE computation (client side)
for(int i = 0; i < snps.size(); i++){
   double frac1 = 4*dalpha.doubleValue()*ddelta.doubleValue();
   frac1 = frac1-Math.pow(dbeta.doubleValue(),2);
   frac1 = Math.pow(frac1, 2);
   frac1 = frac1/dtotalAllele.doubleValue();
   double frac2 = 1/(2*Math.pow(domega.doubleValue(), 2));
   double frac3 = 1/(domega.doubleValue()*dtheta.doubleValue());
   double frac4 = 1/(2*Math.pow(dtheta.doubleValue(), 2));

  snps.get(i).ehwe = frac1*(frac2 + frac3 + frac4);
}
```

Lastly, in computing the $\chi^2$ test statistic, the server computes for the allele counts of the SNPs in the case and control datasets. It also performs all the necessary additions between these allele counts. All the multiplications/divisions in the formula of the test statistic are performed on the client side. The code for $\chi^2$ test statistic computation is as follows:

```
//CHISQ computation (server side)
     static ArrayList<String>
        ↪ privateCHIComputation(ArrayList<ArrayList<String>>
        ↪ SNPsCipherVal, ArrayList<ArrayList<String>>
        ↪ SNPsCipherVal2, PublicKey pk) throws
        ↪ BigIntegerClassNotValid, PublicKeysNotEqualException{
```

```
ArrayList<String> result = new ArrayList();
for(int i = 0; i < SNPsEncSamples.size(); i++){
  EncryptedInteger efAlleleCount = new
     ↪ EncryptedInteger(new BigInteger("0"), pk);
  EncryptedInteger efAlleleCount2 = new
     ↪ EncryptedInteger(new BigInteger("0"), pk);

  for(int j = 0; j < SNPsEncSamples2.get(i).size(); j++){
    efAlleleCount =
       ↪ efAlleleCount.add(SNPsEncSamples2.get(i).get(j));
  }
  for(int j = 0; j < SNPsEncSamples.get(i).size(); j++){
    efAlleleCount2 =
       ↪ efAlleleCount2.add(SNPsEncSamples.get(i).get(j));
  }
  Integer totalAllele = 4*SNPsEncSamples.get(i).size();
  BigInteger fn = new BigInteger(totalAllele.toString());

  alpha = efAlleleCount2.multiply(pk.getN().subtract(new
     ↪ BigInteger("1")));
  alpha = efAlleleCount.add(alpha);
  beta = efAlleleCount.add(efAlleleCount2);
  delta = beta.multiply(pk.getN().subtract(new
     ↪ BigInteger("1")));
  delta = delta.add(fn);

  result.add(alpha.getCipherVal().toString());
  result.add(beta.getCipherVal().toString());
  result.add(delta.getCipherVal().toString());
}
return result;
```

```
    }
//CHISQ computation (client side)
for(int i = 0; i < snps.size(); i++){
    snps.get(i).dalpha2 = dalpha.doubleValue();
    snps.get(i).dbeta2 = dbeta.doubleValue();
    snps.get(i).ddelta2 = ddelta.doubleValue();
    BigDecimal dtotalAllele = new
        ↪ BigDecimal(4*snps.get(i).encryptedSamples.size());
    double num =
        ↪ dtotalAllele.doubleValue()*Math.pow(dalpha.doubleValue(),
        ↪ 2);
    double denom = dbeta.doubleValue()*ddelta.doubleValue();
   snps.get(i).echi = num/denom;
}
```

The return values from the server are decrypted on the client side using the generated private key. The code for decryption is as follows:

```
//decryption for MAF computation results
for(int i = 0; i < snps.size(); i++){
    BigDecimal dfAlleleCount = new
        ↪ BigDecimal(encResult.get(encIndex).decrypt(sk));
    BigDecimal dsAlleleCount = new
        ↪ BigDecimal(encResult.get(encIndex).decrypt(sk));
}

//decryption for HWE computation results
for(int i = 0; i < snps.size(); i++){
    BigDecimal dalpha = new
        ↪ BigDecimal(encResult.get(encIndex).decrypt(sk));
        ↪ encIndex++;
    BigDecimal dbeta = new
```

```
      ↪ BigDecimal(encResult.get(encIndex).decrypt(sk));
      ↪ encIndex++;
   BigDecimal ddelta = new
      ↪ BigDecimal(encResult.get(encIndex).decrypt(sk));
      ↪ encIndex++;
   BigDecimal domega = new
      ↪ BigDecimal(encResult.get(encIndex).decrypt(sk));
      ↪ encIndex++;
   BigDecimal dtheta = new
      ↪ BigDecimal(encResult.get(encIndex).decrypt(sk));
      ↪ encIndex++;
}

//decryption for CHISQ computation results
for(int i = 0; i < snps.size(); i++){
   BigDecimal dalpha = new
      ↪ BigDecimal(encResult.get(encIndex).decrypt(sk));
      ↪ encIndex++;
   if(dalpha.doubleValue() > 400){
     dalpha = dalpha.subtract(new
        ↪ BigDecimal(pk.getN().toString()));
   }
   BigDecimal dbeta = new
      ↪ BigDecimal(encResult.get(encIndex).decrypt(sk));
      ↪ encIndex++;
   BigDecimal ddelta = new
      ↪ BigDecimal(encResult.get(encIndex).decrypt(sk));
      ↪ encIndex++;
   BigDecimal dtotalAllele = new
      ↪ BigDecimal(4*snps.get(i).encryptedSamples.size());
}
```

To assess the performance of the HE scheme, the execution time of each of its algorithm (i.e., Key Generation, Encryption, and Decryption) were obtained. Table 4 shows the timings for the algorithms of the Paillier scheme implemented by THEP. The measurements were done in an Intel Core i3-247 CPU @ 2.40 GHz, 8GB RAM, running 64-bit Windows 10. Values are the mean of the 10 measurements of the respective algorithm.

| **SNP Counts** | **Key Size** | **KeyGen** | **Enc** | **Dec** |
|---|---|---|---|---|
| 150 | 16-bit | 0.052s | 0.528s | 0.015s |
| | 32-bit | 0.08s | 1.476s | 0.16s |
| | 64-bit | 0.96s | 1.656s | 0.16s |
| 300 | 16-bit | 0.001s | 1.079s | 0.017s |
| | 32-bit | 0.001s | 1.24s | 0.031s |
| | 64-bit | 0.081s | 2.471s | 0.047s |
| 600 | 16-bit | 0.065s | 3.693s | 0.063s |
| | 32-bit | 0.065s | 4.005s | 0.063s |
| | 64-bit | 0.081s | 4.016s | 0.063s |

Table 4: Timings for the algorithms of the homomorphic encryption scheme

The security of Paillier scheme is based on large key sizes. However it can be observed from the tabulated values that a large key size has a trade-off of large execution time. The timings for encryption and decryption processes increase along with the key size. It can also be noted that the execution time of these algorithms significantly increases with the SNP count.

For the homomorphic computations, Table 5 shows the difference between the execution times of non-homomorphic and homomorphic computations. The measurements were done in an Intel Core i3-247 CPU @ 2.40 GHz, 8GB RAM, running 64-bit Windows 10. Values are the mean of the 10 measurements of the respective algorithm.

| Computation | Non-homomorphic | Homomophic |
|---|---|---|
| MAF | 0.01s | 0.373s |
| HWE | 0.01s | 0.768s |
| $\chi^2$ | 0.013s | 0.766s |

Table 5: Timings for non-homomorphic and homomorphic computations (Key Size: 64-bit, SNP Count: 300)

Both computations have the same algorithm (i.e., the modified algorithm discussed in Chapter 4) and SNP Count (300). It can be shown that non-homomorphic computations have faster execution time than its homomorphic counterpart. Moreover, the correctness of the computations were verified by comparing the results obtained through homomorphic computations to that of the results obtained through non-homomorphic computations.

In the development of the computing tool, few problems were encountered. These include the serialization/deserialization of `EncryptedInteger` objects. When these objects were manipulated beforehand (e.g., performed homomorphic addition), the serialization/deserialization process fails. This problem was an issue in THEP. Fortunately, the developers have published this workaround to fix problem:

```
public EncryptedInteger(EncryptedInteger other) {
  this.rng = new SecureRandom();
  this.cipherval = other.getCipherVal();
  this.pub = other.getPublicKey();
  this.rngCons = other.rngCons;
  this.biCons = other.biCons;
  this.bigi = other.bigi; // adding this line to this
     ↪ constructor fixed the problem
}
```

The second one is due to the encrypted object serialization/deserialization

incompatibilities of Windows and Linux OS. The workaround to this is to pass the cipher value of the encrypted object (which is of String class) instead of passing the encrypted object itself. This requires both end (the client and the server) to recreate the encrypted object given its cipher value. The THEP provided a private `setCipherVal`method under the `EncryptedInteger` class that can be used for the reconstruction. We had to made it public in order to provide access for both client and server programs.

```
public void setCipherVal(BigInteger cipherval) {
    this.cipherval = cipherval;
}
```

Partial HE (PHE) is generally more efficient than fully HE (FHE) and somewhat HE (SWHE). In genomic computation, most studies use SWHE such as BGV and YASHE scheme. This may be due to the limitations of PHE in terms of the operations that it can homomorphically evaluate. On the other hand, FHE is more powerful than SWHE as it can perform arbitrary number of addition and multiplication, but it is still very inefficient. In this study, it can be shown that Paillier scheme (PHE) can also be used in genomic computations. However, due to its limitation, more computations had to be performed on the client machine. Even so, the performance of Paillier scheme still surpassed the reported performance of BGV and YASHE scheme in other studies.

# VII. Conclusions

The computing tool developed in this project provides secure computation outsourcing of GWAS through homomorphic encryption. With its cryptographic protocols, the computations of sensitive genomic data which commonly have large computational overheads can be delegated to a remote server (e.g., Cloud) without compromising confidentiality.

Furthermore, the tool provides the users the functionalities to load dataset(s), to perform Minor Allele Frequency computation, to perform Hardy-Weinberg Equilibrium computation, to perform $\chi^2$ test statistic computation, and to view the computation results in plaintext format.

THEP provides all the fundamental functions to perform necessary homomorphic encryption processes. Through the homomorphic encryption employed, this project can guarantee the privacy of genomic data without undermining its data utility in research.

# VIII. Recommendations

This project used Paillier scheme to perform all its HE processes. One may use other HE schemes and compare its efficiency with that of Paillier scheme in performing genomic computations. It will be good to consider HE schemes with batching techniques (packing of multiple plaintext messages into the slots of a single ciphertext) so that the sample genotypes need not be encrypted one-by-one. Essentially, the efficiency of the HE scheme should be one of the major considerations.

In addition to that, it is also recommended to include other genomic computations used in GWAS to make the computing tool more useful to the users. In doing so, it will be necessary to find an appropriate HE scheme that will provide the homomorphic operations sufficient to carry out the selected genomic computations. It is important in deciding which HE scheme to be used that the efficiency will not be compromised.

# IX. Bibliography


[1] S. B. Trinidad, S. Fullerton, J. Bares, G. Jarvik, E. Larson, and W. Burke, "Genomic research and wide data sharing: Views of prospective participants," *Genetics in Medicine: Official Journal of the American College of Medical Genetics*, vol. 12(8), 2010.

[2] D. Brutlag, *Scientific American: Introduction to Molecular Medicine*. New York, USA: Scientific American Inc., 1994.

[3] "Genome:unlocking life's code." http://naturalhistory.si.edu/exhibits/genome/. Accessed: 2016-03-25.

[4] M. Naveed, E. Ayday, E. Clayton, J. Fellay, C. Gunter, J.-P. Hubaux, B. Malin, and X. Wang, "Privacy in the genomic era," *ACM Computing Surveys*, vol. 48, 2015.

[5] A. E. Guttmacher and F. Collins, "Welcome to the genomic era," *The New England Journal of Medicine*, vol. 349, 2003.

[6] F. S. Collins, E. D. Green, A. E. Guttmacher, and M. S. Guyer, "A vision for the future of genomics research," *Nature*, vol. 422, 2003.

[7] "Genome-wide association studies." https://www.genome.gov/20019523. Accessed: 2016-03-20.

[8] "Genome-wide association studies." http://www.nature.com/nrg/series/gwas/index.html. Accessed: 2016-03-20.

[9] "What are genome-wide association studies?." https://ghr.nlm.nih.gov/handbook/genomicresearch/gwastudies. Accessed: 2016-03-20.

[10] E. Ayday, E. D. Cristofaro, J. P. Hubaux, and G. Tsudik, "The chills and thrills of whole genome sequencing." arXiv:1306.1264. Accessed: 2015-02-16.

[11] "Hiseq x ten system." http://www.illumina.com/systems/hiseq-x-sequencing-system/system.html. Accessed: 2016-03-20.

[12] G. Yoshizawa, C. W. L. Ho, W. Zhu, C. Hu, Y. Syukriani, I. Lee, H. Kim, D. F. C. Tsai, J. Minari, and K. Kato, "ELSI practices in genomic research in East Asia: implications for research collaboration and public participation," *Genome Medicine*, vol. 6:39, 2014.

[13] K. Lauter, A. Lopez-Alt, and M. Naehrig, *Private Computation on Encrypted Genomic Data*. New York, USA: Springer Interntional Publishing, 2014.

[14] Y. Zhao, X. Wang, and H. Tang, "Secure genomic computation through site-wise encryption," *AMIA Jt Summits Transl Sci Proc.*, 2015.

[15] "Genomic research and wide data sharing: Views of prospective participants." http://www.ncbi.nlm.nih.gov/pmc/articles/PMC3045967/. Accessed: 2016-03-20.

[16] M. Gymrek, A. McGuire, D. Golan, E. Halperin, and Y. Erlich, "Identifying personal genomes by surname inference," *SCIENCE*, vol. 339, 2013.

[17] R. Frederick, "Core concept: Homomorphic encryption," *Proc Natl Acad Sci USA*, 2015.

[18] M. Tebaa, S. E. Hajji, and A. E. Ghazi, "Homomorphic encryption applied to the cloude computing security," *Foundations of Secure Computation*, vol. I, 2012.

[19] P. Paillier, "Public-key cryptosystems based on composite degree residuosity classes," *EUROCRYPT*, 1999.

[20] J. Kupersmith, "The privacy conundrum and genomic research: Re-identification and other concerns;" http://healthaffairs.org/blog/2013/09/11. Accessed: 2016-03-25.

[21] C. Heeney, N. Hawkins, J. de Vries, and P. B. an d J. Kaye, "Assessing the privacy risks of data sharing in genomics," *Public Health Genomics*, 2010.

[22] "Harvard professor re-identifies anonymous volunteers in DNA study." http://www.forbes.com/sites/adamtanner/2013/04/25. Accessed: 2016-03-25.

[23] D. Nyholt, C.-E. Yu, and P. Visscher, "On Jim Watson's APOE status: Genetic infomation is hard to hide," *European Journal of Human Genetics*, 2008.

[24] R. Skloot and B. Turpin, *The Immortal Life of Henrietta Lacks*. New York, USA: Crown Publishers, 2010.

[25] E. C. Hayden, "Privacy loophole found in genetic databases," *nature*, 2013.

[26] "Global alliance for genomics and health." https://genomicsandhealth.org/. Accessed: 2015-04-01.

[27] P. Baldi, P. Gasti, and G. Tsudik, "Fast and private computation of cardinality of set intersection and union," *Proceedings of the 18th ACM conference on Computer and Communications Security*, 2011.

[28] E. D. Cristofaro, , R. Baronio, E. D. Cristofaro, P. Gasti, and G. Tsudik, "Countering gattaca: Efficient and secure testing of fully-sequenced human genomes," *Proceedings of the 18th ACM conference on Computer and Communications Security*, 2011.

[29] R. R. Kumar, V. Navya, M. Swetha, V. Sindhusha, and T. M. Kalyan, "A cryptographic approach to securely share genomic sequences," *International Journal of Computer Science and Information Technologies*, 2012.

[30] M. Blanton, M. Atallah, K. Frikken, and Q. Malluhi, "Secure and efficient outsourcing of sequence comparisons," *European Symposium on Research in Computer Security*, 2012.

[31] R. L. Rivest, L. Adleman, and M. L. Dertouzos, "On data banks and privacy homomorphisms," *Foundations of Secure Computation*, 1978.

[32] D. Boneh, E. Goh, and K. Nissim, "Evaluating 2-DNF formulas on ciphertexts," *Theory of Cryptography*, 2005.

[33] R. L. Rivest, A. Shamir, and L. Adleman, "A method for obtaining digital signatures and public-key cryptosystems," *Commun. ACM*, vol. 26, 1983.

[34] T. ElGamal, "A public-key cryptosystem and a signature scheme based on discrete logarithms," *IEEE Transactions on Information Theory*, 1985.

[35] R. Meissan, "A mathematical approach to fully homomorphic encryption," project, Worcester Polytechnic Institute, MA, USA.

[36] M. Varia, S. Yakoubov, and Y. Yang, "Hetest: A homomorphic encryption testing framework," *Commun. ACM*, vol. 53, 2010.

[37] C. Gentry, "A fully homomorphic encryption scheme," Ph.D. dissertation, Stanford University, California, USA, 2009.

[38] C. Gentry, "Computing arbitary functions of encrypted data," *Commun. ACM*, vol. 53, 2010.

[39] M. Yasuda, T. Shiyomaya, and T. K. J. Kogure, K. Yokoyama, "Secured pattern matching using somewhat homomorphic encryption," *ACM Cloud Computing Security Workshop*, 2013.

[40] J. Cheon, M. Kim, and K. Lauter, "Homomorphic computation of edit distance," *Proceedings of Financial Cryptography and Data Security - FC International Workshop WAHC*, vol. 89, 2015.

[41] J. Cheon, M. Kim, and K. Lauter, "Private genome analysis through homomorphic encryption," *BMC Medical Informatics and Decision Making*, vol. 15, 2015.

[42] E. Ayday, J. L. Raisaro, P. J. McLaren, J. Fellay, and J.-P. Hubaux, "Privacy preserving computation of disease risk by using genomic, clinical, and environmental data," *Workshop on Health Information Technologies*, 2013.

[43] "Helib 1.3." http://shaih.github.io/HElib/. Accessed: 2016-01-12.

[44] S. Halevi and V. Shoup, "Design and implementation of a homomorphic-encryption library," *IBM Research (Manuscript)*, 2013.

[45] L. J. M. Aslett, P. M. Esperanca, and C. C. Holmes, "A review of homomorphic encryption and software tools for encrypted statistical machine learning." arXiv:1508.06574v1. Accessed: 2015-08-26.

[46] "Simple encryption arithmetic library." http://sealcrypto.codeplex.com/. Accessed: 2016-01-12.

[47] "Manual for using homomorphic encryption for bioinformatics." http://research.microsoft.com/pubs/258435/ManualHE.pdf. Accessed: 2016-01-12.

[48] "Analysis of partially and fully homomorphic encryption." http://www.liammorris.com/crypto2/Homomorphic%20Encryption%20Paper.pdf. Accessed: 2016-01-30.

[49] "The homomorphic encrytion project." https://code.google.com/p/thep/. Accessed: 2016-01-12.

[50] "The paillier cryptosystem: A look into the cryptosystem and its potential application." http://www.tcnj.edu/~hagedorn/papers/CapstonePapers/OKeeffe/CapstoneOKeeffeCryptography.pdf. Accessed: 2016-01-30.

[51] K. V. Steen, "Genetics and bioinformatics." http://www.montefiore.ulg.ac.be/~kbessonov/present_data/GBIO0002-1_

GenAndBioinf2015-16/lectures/L3/GBIO0002_1516_Lecture_3.pdf. Accessed: 2016-04-08.

[52] "New SNP attribute." http://www.ncbi.nlm.nih.gov/projects/SNP/docs/rs_attributes.html. Accessed: 2016-03-20.

[53] M. E. Tabangin, J. G. Woo, and L. J. Martin, "The effect of minor allele frequency on the likelihood of obtaining false positives," *BMC Proceedingsg*, vol. 3, 2009.

[54] "Quality control for genome wide association studies." http://cgondro2.une.edu.au/snpQC/QCtutorial.pdf. Accessed: 2016-03-20.

[55] E. P. Hong and J. W. Park, "Sample size and statistical power calculation in genetic association studies," *Genomics Informatics*, vol. 10(2), 2012.

[56] J.-H. Park, M. H. Gail, C. Weinberg, R. J. Caroll, C. C. Chung, Z. Wang, S. J. Chanok, and N. C. Joseph F. Fraumeni, Jr., "Distribution of allele frequencies and effect sizes and their interrelationships for common genetic susceptibility variants," *Proceedings of the National Academy of Sciences of the United States*, vol. 108, 2011.

[57] J. G. Smith and C. Newton-Cheh, "Genome-wide association study in humans." http://www.broadinstitute.org/~chrisnc/others/GWAS_Smith_MethMolBiol09.pdf. Accessed: 2016-04-08.

[58] "Mining the hapmap phasei data using hapmart." http://hapmap.ncbi.nlm.nih.gov/hapmart.html.en. Accessed: 2016-03-20.

[59] J. Cheon, M. Kim, and K. Lauter, "Estimation of allele frequency and association mapping using next-generation sequencing data," *BMC Bioinformatics*, vol. 12:231, 2011.

[60] K. Karczewski, "How to do a genome wide association studies." http://stanford.edu/class/gene210/files/gwas-howto.pdf. Accessed: 2016-03-20.

[61] "The variant call format (vcf) version 4.2 specification." https://samtools.github.io/hts-specs/VCFv4.2.pdf. Accessed: 2016-02-23.

# X. Appendix

## A. Source Code

1. GWASComputingTool.java

```
import java.awt.Color;
import java.awt.Cursor;
import java.awt.Dimension;
import java.awt.Font;
import java.awt.Insets;
import java.util.List;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import java.awt.event.WindowAdapter;
import java.awt.event.WindowEvent;
import java.io.BufferedReader;
import java.io.File;
import java.io.FileReader;
import java.io.IOException;
import java.io.ObjectInputStream;
import java.io.ObjectOutputStream;
import java.io.Serializable;
import java.math.BigDecimal;
import java.math.BigInteger;
import java.util.ArrayList;
import java.net.*;

import javax.swing.ImageIcon;
import javax.swing.JButton;
import javax.swing.JComboBox;
import javax.swing.JFileChooser;
import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.JPanel;
import javax.swing.JScrollPane;
import javax.swing.JTable;
import javax.swing.JTextArea;
import javax.swing.JTextField;
import javax.swing.SwingWorker;
import javax.swing.UIManager;
import javax.swing.UnsupportedLookAndFeelException;
import javax.swing.table.DefaultTableCellRenderer;
import javax.swing.table.DefaultTableModel;
import javax.swing.table.TableColumnModel;


import thep.paillier.EncryptedInteger;
import thep.paillier.PrivateKey;
import thep.paillier.PublicKey;
import thep.paillier.exceptions.BigIntegerClassNotValid;
import thep.paillier.exceptions.PublicKeysNotEqualException;

public class GWASComputingTool implements Serializable{
  private static final long serialVersionUID = 1L;

  private static JFrame frame;
  private static JPanel panelA, panelB, panelC;
  private static JTextArea resultBoard, statusBoard;
  private static JScrollPane resultScroll, statusScroll;
  private static JComboBox<String> modeOfComp;
  private static JButton file1Button, file2Button, computeButton, cancelButton,
          ↪ connect;
  private static JFileChooser fileChooser;
  private static DefaultTableModel model;
  private static JTable computationResult;
  private static JLabel connection, status, connectTo, port;
  private static JTextField ipAdd, portNo;

  private static int mode = 1;
  private static String fileName1 = "";
  private static String fileName2 = "";
  private static ArrayList<String> lines = new ArrayList<String>();//stores all
          ↪ the lines in the input file except the header line
  private static ArrayList<String> lines2 = new ArrayList<String>(); // for the
          ↪ possible second file in chisq
  private static ArrayList<Snp> snps = new ArrayList<Snp>(); //stores all the
          ↪ SNPs in the input file
  private static ArrayList<Snp> snps2 = new ArrayList<Snp>();
  private static int noOfSamples = 0; //number of sample genotypes
  private static int noOfSamples2 = 0;
  private static int noOfSNPs = 0; //number of SNPs in the dataset
  private static PrivateKey sk;
  private static PublicKey pk;
  private static String n = "64";
  private static int connected = 0;
  private static long startTime = 0;
  private static long endTime = 0;
  private static double totalTime = 0;
  private static double mismatchRate = 0;
  private static int terminated = 0;

  static SwingWorker<Boolean, String> encodeWorker, encryptionWorker,
          ↪ keysWorker, computationWorker = null;

  private static ArrayList<ArrayList<String>> SNPsCipherVal = new
          ↪ ArrayList<ArrayList<String>>();
  private static ArrayList<ArrayList<String>> SNPsCipherVal2 = new
          ↪ ArrayList<ArrayList<String>>();
  private static ObjectOutputStream clientOutputStream;
  private static ObjectInputStream clientInputStream;
  private static Socket clientSocket;

  static void closeConnection() throws IOException{
    if(connected == 1){
      SNPsCipherVal = new ArrayList<ArrayList<String>>();

      ArrayList<String> container = new ArrayList<String>();
      container.add("0");
      container.add("0");
      container.add("4");

      SNPsCipherVal.add(container);
      clientOutputStream.writeObject(SNPsCipherVal);
      clientOutputStream.writeObject(SNPsCipherVal2);
      clientOutputStream.flush();

      clientOutputStream.close();
      clientInputStream.close();
      System.out.println("Closing connection...");
      clientSocket.close();
    }
    System.exit(1);
  }

  static boolean setUpConnection(String ipAdd, String port) throws
          ↪ UnknownHostException, IOException{
    try{
      clientSocket = new Socket(ipAdd, Integer.valueOf(port));
      statusBoard.append(">Connected to server " + clientSocket.getInetAddress()
              ↪ + ". \n");
      clientOutputStream = new
              ↪ ObjectOutputStream(clientSocket.getOutputStream());
      clientInputStream = new ObjectInputStream(clientSocket.getInputStream());
      computeButton.setEnabled(true);
      connected = 1;
      return true;
    }catch(Exception e){
      statusBoard.append(">Connection failed." + "\n");
      return false;
    }
  }

  static void openWindow() throws ClassNotFoundException,
          ↪ InstantiationException, IllegalAccessException,
          ↪ UnsupportedLookAndFeelException{
    UIManager.setLookAndFeel("com.sun.java.swing.plaf.windows.WindowsLookAndFeel");

    ImageIcon imgicon = new ImageIcon("images/dna_icon.jpg");
    frame = new JFrame("GWAS Computing Tool");
    frame.setSize(800, 600);
    frame.getContentPane().setBackground(new Color(240,240,240));
    frame.setIconImage(imgicon.getImage());
    frame.setLayout(null);
    frame.setResizable(false);
    frame.setVisible(true);
    frame.addWindowListener(new WindowAdapter() {
        @Override
        public void windowClosing(WindowEvent event) {
          try {
          closeConnection();
        } catch(SocketException sockEx) {

        }catch (IOException e) {
          e.printStackTrace();
        }
        }
    });

    panelA = new JPanel();
    panelA.setLayout(null);
    panelA.setBounds(5, 333, 770, 50);
    panelA.setBackground(new Color(240,240,240));

    panelB = new JPanel();
    panelB.setLayout(null);
    panelB.setBounds(5, 25, 785, 305);

    panelC = new JPanel();
    panelC.setLayout(null);
    panelC.setBounds(5, 390, 785, 175);

    connection = new JLabel("Connection:");
    connection.setBounds(650,2,70,20);
    connection.setFont(new Font("Tahoma", Font.PLAIN, 12));

    status = new JLabel("Offline");
    status.setBounds(730,2,70,20);
    status.setFont(new Font("Tahoma", Font.PLAIN, 12));

    connectTo = new JLabel("IP Address: ");
    connectTo.setBounds(7,2,70,20);
    connectTo.setFont(new Font("Tahoma", Font.PLAIN, 12));

    ipAdd = new JTextField();
    ipAdd.setBounds(80,2,100,20);
    ipAdd.setBackground(new Color(240,240,240));

    port = new JLabel("Port:");
    port.setBounds(185,2,30,20);
    port.setFont(new Font("Tahoma", Font.PLAIN, 12));

    portNo = new JTextField();
    portNo.setBounds(220,2,40,20);
    portNo.setBackground(new Color(240,240,240));

    connect = new JButton("Connect");
    connect.setBounds(265,1,80,22);
    connect.setFont(new Font("Tahoma", Font.PLAIN, 12));
    connect.addActionListener(new ActionListener() {
```

```
    @Override
    public void actionPerformed(ActionEvent arg0) {
       try {
         if(setUpConnection(ipAdd.getText(),portNo.getText())){
           status.setText("Online");
           computeButton.setEnabled(true);
         }
       } catch (UnknownHostException e) {
         e.printStackTrace();
       } catch (IOException e) {
         e.printStackTrace();
       };
     }
    });

resultBoard = new JTextArea();
resultBoard.setBounds(0,0,770,305);
resultBoard.setRows(50);
resultBoard.setBackground(Color.white);
resultBoard.setEditable(false);

resultScroll = new JScrollPane(resultBoard,
        ↪ JScrollPane.VERTICAL_SCROLLBAR_ALWAYS,
        ↪ JScrollPane.HORIZONTAL_SCROLLBAR_ALWAYS);
resultScroll.setAutoscrolls(true);
resultScroll.setBounds(0,0,785,305);
resultScroll.setPreferredSize(new Dimension(800, 110));
resultScroll.setBorder(null);

statusBoard = new JTextArea();
statusBoard.setBounds(0,0,770,150);
statusBoard.setMargin(new Insets(10,10,10,10));
statusBoard.setRows(50);
statusBoard.setBackground(Color.black);
statusBoard.setForeground(new Color(240,240,240));
statusBoard.setLineWrap(true);
statusBoard.setWrapStyleWord(true);
statusBoard.setEditable(false);

statusScroll = new JScrollPane(statusBoard,
        ↪ JScrollPane.VERTICAL_SCROLLBAR_ALWAYS,
        ↪ JScrollPane.HORIZONTAL_SCROLLBAR_NEVER);
statusScroll.setAutoscrolls(true);
statusScroll.setBounds(0,0,785,175);
statusScroll.setPreferredSize(new Dimension(800, 110));
statusScroll.setBorder(null);

 file1Button = new JButton("Dataset (1)");
 file1Button.setBounds(210,5,120,40);
 file1Button.setFont(new Font("Tahoma", Font.BOLD, 12));
 file1Button.setCursor(new Cursor(Cursor.HAND_CURSOR));
 file1Button.addActionListener(new ActionListener() {
  @Override
  public void actionPerformed(ActionEvent arg0) {
     fileChooser = new JFileChooser();
     fileChooser.setCurrentDirectory(new
             ↪ File(System.getProperty("user.dir")));
     int result = fileChooser.showOpenDialog(frame);
     if (result == JFileChooser.APPROVE_OPTION) {
        File selectedFile = fileChooser.getSelectedFile();
        fileName1 = selectedFile.getName();
        file1Button.setText(fileName1);

        statusBoard.append(">File " + fileName1 + " has been selected." +
                ↪ "\n");
     }
  }
 });

 file2Button = new JButton("Dataset (2)");
 file2Button.setBounds(335,5,120,40);
 file2Button.setFont(new Font("Tahoma", Font.BOLD, 12));
 file2Button.setCursor(new Cursor(Cursor.HAND_CURSOR));
 file2Button.setEnabled(false);
 file2Button.addActionListener(new ActionListener() {
  @Override
  public void actionPerformed(ActionEvent arg0) {
     fileChooser = new JFileChooser();
     fileChooser.setCurrentDirectory(new
             ↪ File(System.getProperty("user.dir")));
     int result = fileChooser.showOpenDialog(frame);
     if (result == JFileChooser.APPROVE_OPTION) {
        File selectedFile = fileChooser.getSelectedFile();
        fileName2 = selectedFile.getName();
        file2Button.setText(fileName2);

        statusBoard.append(">File " + fileName2 + " has been selected." +
                ↪ "\n");
     }
  }
 });

String[] comp = new String[] {" Minor Allele Frequency", " Hardy-Weinberg
        ↪ Equilibrium", " Chi-square Test Statistic"};
modeOfComp = new JComboBox<>(comp);
modeOfComp.setBounds(5,5,200,40);
modeOfComp.setFont(new Font("Tahoma", Font.BOLD, 12));
modeOfComp.setCursor(new Cursor(Cursor.HAND_CURSOR));
modeOfComp.addActionListener(new ActionListener() {
  @Override
  public void actionPerformed(ActionEvent arg0) {
    if(modeOfComp.getSelectedIndex() == 0){
      mode = 1;
      file1Button.setText("Dataset (1)");
      file2Button.setText("Dataset (2)");
      file2Button.setEnabled(false);

      statusBoard.append(">Mode of computation: MAF." + "\n");
    }else if(modeOfComp.getSelectedIndex() == 1){
      mode = 2;
      file1Button.setText("Dataset (1)");
      file2Button.setText("Dataset (2)");
      file2Button.setEnabled(false);

      statusBoard.append(">Mode of computation: HWE." + "\n");
    }else if(modeOfComp.getSelectedIndex() == 2){
      mode = 3;
      file1Button.setText("Control");
      file2Button.setText("Case");
      file2Button.setEnabled(true);

      statusBoard.append(">Mode of computation: Chisq test stat." + "\n");
    }
  }
 });

computeButton = new JButton("Compute");
computeButton.setBounds(560,5,100,40);
computeButton.setFont(new Font("Tahoma", Font.BOLD, 12));
computeButton.setCursor(new Cursor(Cursor.HAND_CURSOR));
computeButton.setEnabled(false);
computeButton.addActionListener(new ActionListener() {
 @Override
 public void actionPerformed(ActionEvent arg0) {
   snps = new ArrayList<Snp>();
   snps2 = new ArrayList<Snp>();
   lines = new ArrayList<String>();
   lines2 = new ArrayList<String>();

   if(mode == 1 || mode == 2){
     int checker = 0;
       try {
       BufferedReader inputStream = new BufferedReader(new
               ↪ FileReader(fileName1));
       String line = "";
       while((line=inputStream.readLine()) != null){
         if(!line.contains("#")){
           lines.add(line);
         }else{
           noOfSamples = countSamples(line);
         }
       }
       inputStream.close();

       statusBoard.append(">Preparing dataset..." + "\n");

       noOfSNPs = lines.size()/2;
       if(setSamples(snps, lines, noOfSamples)){
         checker = 1;
       }else{
         statusBoard.append(">Invalid input file." + "\n");
       }
       } catch (IOException e) {
       statusBoard.append(">No/Invalid input file." + "\n");
     }

       if(checker == 1){
         if(mode == 1){
           encodeWorker = new SwingWorker<Boolean, String>() {
           @Override
           protected Boolean doInBackground() throws Exception {
                System.out.println("[MAF]...\n");

             publish("Encoding dataset...");
             Thread.sleep(1000);
             for(int i = 0; i < snps.size(); i++){
               char fallele = snps.get(i).getFallele().charAt(0);
               for(int j = 0; j < snps.get(i).samples.size(); j++){
                 int code = 0;
                 if(snps.get(i).samples.get(j).charAt(0) == fallele){
                   code++;
                 }
                 if(snps.get(i).samples.get(j).charAt(1) == fallele){
                   code++;
                 }
                 snps.get(i).addEncoding(code);

                 String line = "Genotype sample no. " + (j+1) + " of SNP
                         ↪ #" + (i+1) + " has been encoded.";
                 publish(line);
                 //Thread.sleep(100);
               }
             }
             Thread.sleep(500);
             return true;
                }
                @Override
                protected void process(List<String> msgs) {
                  for(String msg: msgs){
                    statusBoard.append(">" + msg + "\n");
                  }
                }
                @Override
                protected void done() {
                  keysWorker = new SwingWorker<Boolean, String>() {
                  @Override
                  protected Boolean doInBackground() throws Exception {
                    publish("Encoding complete.");
                    publish("Generating keys...");
                    Thread.sleep(1000);

                    System.out.println("Key Generation started...");
                    startTime = System.currentTimeMillis();

                    sk = new PrivateKey(64);
                    pk = sk.getPublicKey();

                    endTime = System.currentTimeMillis();
                    totalTime = endTime-startTime;
                    System.out.println("Key Generation completed in " +
                            ↪ totalTime/1000 + "s.");

                    String line = "Private and Public keys have been
                            ↪ generated.";
                    publish(line);
                    return true;
                       }
                       @Override
                       protected void process(List<String> msgs) {
                         for(String msg: msgs){
```

```
    statusBoard.append(">" + msg + "\n");
  }
}
@Override
protected void done() {
  encryptionWorker = new SwingWorker<Boolean,
          ↪ String>() {
  @Override
  protected Boolean doInBackground() throws
          ↪ Exception {
    publish("Key generation complete.");
    publish("Encrypting dataset...");
    Thread.sleep(1000);

    startTime = System.currentTimeMillis();
    System.out.println("Encryption started...");

    for(int i = 0; i < snps.size(); i++){
      for(int j = 0; j <
              ↪ snps.get(i).encodedSamples.size();
              ↪ j++){
        BigInteger code = new
                ↪ BigInteger(snps.get(i).encodedSamples.get(j).toString());
        EncryptedInteger enc = new
                ↪ EncryptedInteger(code,
                ↪ sk.getPublicKey());
        snps.get(i).encryptedSamples.add(enc);

        String line = "Genotype sample no. " +
                ↪ (j+1) + " of SNP #" + (i+1) +
                ↪ " has been
                ↪ encrypted.\n(Encrpted Value: "
                ↪ + enc.getCipherVal() + ")";
        publish(line);
        //Thread.sleep(100);
      }
    }

    endTime = System.currentTimeMillis();
    totalTime = endTime-startTime;
    System.out.println("Encryption completed in " +
            ↪ totalTime/1000 + "s.");

    return true;
      }
      @Override
      protected void process(List<String> msgs) {
        for(String msg: msgs){
          statusBoard.append(">" + msg + "\n");
        }
      }
      @Override
      protected void done() {
        computationWorker = new
                ↪ SwingWorker<Boolean, String>()
                ↪ {
      @Override
      protected Boolean doInBackground() throws
              ↪ Exception {
        ArrayList<ArrayList<EncryptedInteger>>
                ↪ SNPsEncSamples = new
                ↪ ArrayList<ArrayList<EncryptedInteger>>();
        ArrayList<EncryptedInteger> encResult =
                ↪ new
                ↪ ArrayList<EncryptedInteger>();
        ArrayList<ArrayList<Integer>> SNPsSamples
                ↪ = new ArrayList<ArrayList
                ↪ <Integer>>();
        ArrayList<Integer> result = new
                ↪ ArrayList<Integer>();
        int encIndex = 0;
        int index = 0;

        SNPsCipherVal = new
                ↪ ArrayList<ArrayList<String>>();
        SNPsCipherVal2 = new
                ↪ ArrayList<ArrayList<String>>();
        ArrayList<String> strResult = new
                ↪ ArrayList<String>();

        publish("Encryption Complete.");
        publish("Uploading dataset...");
        Thread.sleep(1000);

        for(int i = 0; i < snps.size(); i++){
          SNPsEncSamples.add(snps.get(i).encryptedSamples);
          SNPsSamples.add(snps.get(i).encodedSamples);
        }

        ArrayList<String> container = new
                ↪ ArrayList<String>();

        //first element of the
                ↪ ArrayList<ArrayList<String>>
                ↪ SNPsCipherVal
        container.add(n);
        container.add(pk.getN().toString());
        container.add("1");
        SNPsCipherVal.add(container);

        for(int i = 0; i < SNPsEncSamples.size();
                ↪ i++){
          container = new ArrayList<String>();
          for(int j = 0; j <
                  ↪ SNPsEncSamples.get(i).size();
                  ↪ j++){
            container.add(SNPsEncSamples.get(i).get(j).getCipherVal().toString());
          }
          SNPsCipherVal.add(container);
        }

        try {
          clientOutputStream.writeObject(SNPsCipherVal);
          clientOutputStream.writeObject(SNPsCipherVal2);
          clientOutputStream.flush();
          publish("Dataset has been uploaded.");
          publish("Downloading computation results
                  ↪ from the server...");
          Thread.sleep(1000);

          strResult =
                  ↪ (ArrayList<String>)clientInputStream.readObject();

          publish("Computation results have been
                  ↪ downloaded.");
          Thread.sleep(1000);
        }catch(SocketException sockEx) {
          publish("Connection failure: MAF cannot
                  ↪ be completed.");
          connected = 0;
        }catch(IOException ioe) {
          ioe.printStackTrace();
        }


        if(connected == 1){
          for(int i = 0; i < strResult.size();
                  ↪ i++){
            EncryptedInteger encInt = new
                    ↪ EncryptedInteger(pk);
            encInt.setCipherVal(new
                    ↪ BigInteger(strResult.get(i)));
            encResult.add(encInt);
          }

          result =
                  ↪ ordinaryMAFComputation(SNPsSamples);

          startTime = System.currentTimeMillis();
          System.out.println("Decryption
                  ↪ started...");

          for(int i = 0; i < snps.size(); i++){
            BigDecimal dfAlleleCount = new
                    ↪ BigDecimal(encResult.get(encIndex).decrypt(sk));
            snps.get(i).dfAlleleCount =
                    ↪ dfAlleleCount.doubleValue();
            encIndex++;
            BigDecimal dsAlleleCount = new
                    ↪ BigDecimal(encResult.get(encIndex).decrypt(sk));
            snps.get(i).dsAlleleCount =
                    ↪ dsAlleleCount.doubleValue();
            encIndex++;
            BigDecimal dtotalAllele = new
                    ↪ BigDecimal(2*(snps.get(i).encryptedSamples.size()));

            if(dfAlleleCount.compareTo(dsAlleleCount)
                    ↪ == -1){
              snps.get(i).emaf =
                      ↪ dfAlleleCount.divide(dtotalAllele).doubleValue();
            }else{
              snps.get(i).emaf =
                      ↪ dsAlleleCount.divide(dtotalAllele).doubleValue();
            }
          }

          endTime = System.currentTimeMillis();
          totalTime = endTime-startTime;
          System.out.println("Decryption
                  ↪ completed in " +
                  ↪ totalTime/1000 + "s.");

          for(int i = 0; i < snps.size(); i++){
            double fAlleleCount =
                    ↪ result.get(index);
            index++;
            double sAlleleCount =
                    ↪ result.get(index);
            index++;
            double totalAllele =
                    ↪ 2*snps.get(i).encodedSamples.size();

            if(fAlleleCount < sAlleleCount){
              snps.get(i).maf =
                      ↪ fAlleleCount/(totalAllele);
            }else{
              snps.get(i).maf =
                      ↪ sAlleleCount/(totalAllele);
            }
          }

          for(int i = 0; i < snps.size(); i++){
            if(snps.get(i).emaf !=
                    ↪ snps.get(i).maf){
              mismatchRate++;
            }
          }
          mismatchRate =
                  ↪ mismatchRate/snps.size();

          displayResults();
        }
        return true;
      }
      @Override
      protected void process(List<String>
                      ↪ msgs) {
        for(String msg: msgs){
          statusBoard.append(">" + msg +
                  ↪ "\n");
        }
      }
      @Override
      protected void done() {
        if(connected == 1){
          statusBoard.append(">MAF
                  ↪ Computation Complete |
                  ↪ Mismatch Rate: " +
                  ↪ mismatchRate + "% | Key
```

```
                                    ↪ Size: " + n + "bit |
                                    ↪ SNP Counts: " +
                                    ↪ noOfSNPs);
                                System.out.println("\n[END]...\n");
                              }else{
                                status.setText("Offline");
                                computeButton.setEnabled(false);
                                System.out.println("\n[END]...\n");
                              }
                            }
                          };
                          computationWorker.execute();
                           }
                      };
                      encryptionWorker.execute();
                       }
                  };
                  keysWorker.execute();
                   }
              };
                encodeWorker.execute();
            }else if(mode == 2){
              encodeWorker = new SwingWorker<Boolean, String>() {
              @Override
              protected Boolean doInBackground() throws Exception {
                  System.out.println("[HWE]...\n");

                for(int i = 0; i < snps.size(); i++){
                  char fallele = snps.get(i).getFallele().charAt(0);
                  for(int j = 0; j < snps.get(i).samples.size(); j++){
                    int code = 0;
                    if(snps.get(i).samples.get(j).charAt(0) == fallele){
                      code++;
                    }
                    if(snps.get(i).samples.get(j).charAt(1) == fallele){
                      code++;
                    }

                    switch(code){
                    case 0: snps.get(i).encodedSamples2.add(0);
                         snps.get(i).encodedSamples2.add(0);
                         snps.get(i).encodedSamples2.add(1);
                         break;
                    case 1: snps.get(i).encodedSamples2.add(0);
                         snps.get(i).encodedSamples2.add(1);
                         snps.get(i).encodedSamples2.add(0);
                         break;
                    case 2: snps.get(i).encodedSamples2.add(1);
                         snps.get(i).encodedSamples2.add(0);
                         snps.get(i).encodedSamples2.add(0);
                         break;
                    }
                    String line = "Genotype sample no. " + (j+1) + " of SNP
                            ↪ #" + (i+1) + " has been encoded.";
                    publish(line);
                    //Thread.sleep(100);
                  }
                }
                return true;
                   }
                   @Override
                   protected void process(List<String> msgs) {
                     for(String msg: msgs){
                       statusBoard.append(">" + msg + "\n");
                     }
                   }
                   @Override
                   protected void done() {
                     SwingWorker<Boolean, String> keysWorker = new
                             ↪ SwingWorker<Boolean, String>() {
                     @Override
                     protected Boolean doInBackground() throws Exception {
                       startTime = System.currentTimeMillis();
                       System.out.println("Key Generation started...");

                       sk = new PrivateKey(64);
                       pk = sk.getPublicKey();

                       endTime = System.currentTimeMillis();
                       totalTime = endTime-startTime;
                       System.out.println("Key Generation completed in " +
                               ↪ totalTime/1000 + "s.");

                       String line = "Private and Public keys have been
                               ↪ generated.";
                       publish(line);
                       return true;
                          }
                          @Override
                          protected void process(List<String> msgs) {
                            for(String msg: msgs){
                              statusBoard.append(">" + msg + "\n");
                            }
                          }
                          @Override
                          protected void done() {
                            SwingWorker<Boolean, String> encryptionWorker =
                                    ↪ new SwingWorker<Boolean, String>() {
                             @Override
                             protected Boolean doInBackground() throws
                                     ↪ Exception {
                               startTime = System.currentTimeMillis();
                               System.out.println("Encryption started...");

                               for(int k = 0; k < snps.size(); k++){
                               int ctr = 1;
                                 for(int l = 0; l <
                                         ↪ snps.get(k).encodedSamples2.size();
                                         ↪ l++){
                                   BigInteger code2 = new
                                           ↪ BigInteger(snps.get(k).encodedSamples2.get(l).toString());
                                   EncryptedInteger enc2 = new
                                           ↪ EncryptedInteger(code2,
                                           ↪ sk.getPublicKey());
                                   snps.get(k).encryptedSamples2.add(enc2);
                                   if(ctr%3==0){
                                     String line = "Genotype sample no. " +
                                             ↪ ctr/3 + " of SNP #" + (k+1)
                                             ↪ + " has been
                                             ↪ encrypted.\n(Encrpted
                                             ↪ Value: " +
                                             ↪ enc2.getCipherVal() + ")";
                                     publish(line);
                                   }
                                   ctr++;
                                 }
                               //Thread.sleep(100);
                               }

                               endTime = System.currentTimeMillis();
                               totalTime = endTime-startTime;
                               System.out.println("Encryption completed in " +
                                       ↪ totalTime/1000 + "s.");

                               return true;
                                  }
                                  @Override
                                  protected void process(List<String> msgs) {
                                    for(String msg: msgs){
                                      statusBoard.append(">" + msg + "\n");
                                    }
                                  }
                                  @Override
                                  protected void done() {
                                    computationWorker = new
                                            ↪ SwingWorker<Boolean, String>()
                                            ↪ {
                                 @Override
                                 protected Boolean doInBackground() throws
                                         ↪ Exception {
                                   ArrayList<ArrayList<EncryptedInteger>>
                                            ↪ SNPsEncSamples = new
                                            ↪ ArrayList<ArrayList<EncryptedInteger>>();
                                   ArrayList<EncryptedInteger> encResult =
                                            ↪ new
                                            ↪ ArrayList<EncryptedInteger>();
                                   ArrayList<ArrayList<Integer>> SNPsSamples
                                            ↪ = new ArrayList<ArrayList
                                            ↪ <Integer>>();
                                   ArrayList<Integer> result = new
                                            ↪ ArrayList<Integer>();
                                   int encIndex = 0;
                                   int index = 0;

                                   SNPsCipherVal = new
                                            ↪ ArrayList<ArrayList<String>>();
                                   SNPsCipherVal2 = new
                                            ↪ ArrayList<ArrayList<String>>();
                                   ArrayList<String> strResult = new
                                            ↪ ArrayList<String>();

                                   publish("Encryption Complete.");
                                   publish("Uploading dataset...");
                                   Thread.sleep(1000);

                                   for(int i = 0; i < snps.size(); i++){
                                     SNPsEncSamples.add(snps.get(i).encryptedSamples2);
                                     SNPsSamples.add(snps.get(i).encodedSamples2);
                                   }

                                   ArrayList<String> container = new
                                            ↪ ArrayList<String>();

                                   //first element of the
                                            ↪ ArrayList<ArrayList<String>>
                                            ↪ SNPsCipherVal
                                   container.add(n);
                                   container.add(pk.getN().toString());
                                   container.add("2");
                                   SNPsCipherVal.add(container);

                                   for(int i = 0; i < SNPsEncSamples.size();
                                            ↪ i++){
                                     container = new ArrayList<String>();
                                     for(int j = 0; j <
                                              ↪ SNPsEncSamples.get(i).size();
                                              ↪ j++){
                                       container.add(SNPsEncSamples.get(i).get(j).getCipherVal().toString());
                                     }
                                     SNPsCipherVal.add(container);
                                   }

                                   try {
                                     clientOutputStream.writeObject(SNPsCipherVal);
                                     clientOutputStream.writeObject(SNPsCipherVal2);
                                     clientOutputStream.flush();

                                     publish("Dataset has been uploaded.");
                                     publish("Downloading computation results
                                              ↪ from the server...");
                                     Thread.sleep(1000);

                                     strResult =
                                              ↪ (ArrayList<String>)clientInputStream.readObject();

                                     publish("Computation results have been
                                              ↪ downloaded.");
                                     Thread.sleep(1000);
                                   }catch(SocketException sockEx) {
                                     publish("Connection failure: HWE cannot
                                              ↪ be completed.");
                                     connected = 0;
                                   }catch(IOException ioe) {
                                     ioe.printStackTrace();
                                   }

                                   if(connected == 1){
                                     for(int i = 0; i < strResult.size();
                                              ↪ i++){
```

```
    EncryptedInteger encInt = new
            ↪ EncryptedInteger(pk);
    encInt.setCipherVal(new
            ↪ BigInteger(strResult.get(i)));
    encResult.add(encInt);
  }

  result =
          ↪ ordinaryHWEComputation(SNPsSamples);

  startTime = System.currentTimeMillis();
  System.out.println("Decryption
          ↪ started...");

  for(int i = 0; i < snps.size(); i++){
    BigDecimal dalpha = new
            ↪ BigDecimal(encResult.get(encIndex).decrypt(sk));
            ↪ encIndex++;
    snps.get(i).dalpha =
            ↪ dalpha.doubleValue();
    BigDecimal dbeta = new
            ↪ BigDecimal(encResult.get(encIndex).decrypt(sk));
            ↪ encIndex++;
    snps.get(i).dbeta =
            ↪ dbeta.doubleValue();
    BigDecimal ddelta = new
            ↪ BigDecimal(encResult.get(encIndex).decrypt(sk));
            ↪ encIndex++;
    snps.get(i).ddelta =
            ↪ ddelta.doubleValue();
    BigDecimal domega = new
            ↪ BigDecimal(encResult.get(encIndex).decrypt(sk));
            ↪ encIndex++;
    snps.get(i).domega =
            ↪ domega.doubleValue();
    BigDecimal dtheta = new
            ↪ BigDecimal(encResult.get(encIndex).decrypt(sk));
            ↪ encIndex++;
    snps.get(i).dtheta =
            ↪ dtheta.doubleValue();
    BigDecimal dtotalAllele = new
            ↪ BigDecimal(2*(snps.get(i).encryptedSamples2.size()/3));

    double frac1 =
            ↪ 4*dalpha.doubleValue()*ddelta.doubleValue();
    frac1 =
            ↪ frac1-Math.pow(dbeta.doubleValue(),2);
    frac1 = Math.pow(frac1, 2);
    frac1 =
            ↪ frac1/dtotalAllele.doubleValue();
    double frac2 =
            ↪ 1/(2+Math.pow(domega.doubleValue(),
            ↪ 2));
    double frac3 =
            ↪ 1/(domega.doubleValue()+dtheta.doubleValue());
    double frac4 =
            ↪ 1/(2+Math.pow(dtheta.doubleValue(),
            ↪ 2));

    snps.get(i).ehwe = frac1+(frac2 +
            ↪ frac3 + frac4);
  }

  endTime = System.currentTimeMillis();
  totalTime = endTime-startTime;
  System.out.println("Decryption
          ↪ completed in " +
          ↪ totalTime/1000 + "s.");

  for(int i = 0; i < snps.size(); i++){
    double alpha = result.get(index);
            ↪ index++;
    double beta = result.get(index);
            ↪ index++;
    double delta = result.get(index);
            ↪ index++;
    double omega = result.get(index);
            ↪ index++;
    double theta = result.get(index);
            ↪ index++;
    double totalAllele =
            ↪ 2*(snps.get(i).encodedSamples2.size()/3);

    double frac1 =
            ↪ Math.pow((4+alpha*delta)-Math.pow(beta,
            ↪ 2), 2)/totalAllele;
    double frac2 = 1/(2+Math.pow(omega,
            ↪ 2));
    double frac3 = 1/(omega+theta);
    double frac4 = 1/(2+Math.pow(theta,
            ↪ 2));

    snps.get(i).hwe =
            ↪ frac1*(frac2+frac3+frac4);
  }

  for(int i = 0; i < snps.size(); i++){
    if(snps.get(i).ehwe !=
            ↪ snps.get(i).hwe){
      mismatchRate++;
    }
  }
  mismatchRate =
          ↪ mismatchRate/snps.size();

 displayResults();
}

return true;
  }
  @Override
  protected void process(List<String>
          ↪ msgs) {
    for(String msg: msgs){
      statusBoard.append(">" + msg +
              ↪ "\n");
    }
  }
  @Override
  protected void done() {
    if(connected == 1){
      statusBoard.append(">HWE
              ↪ Computation Complete |
              ↪ Mismatch Rate: " +
              ↪ mismatchRate + "% | Key
              ↪ Size: " + n + "bit |
              ↪ SNP Counts: " +
              ↪ noOfSNPs);
      System.out.println("\n[END]...\n");
    }else{
      status.setText("Offline");
      computeButton.setEnabled(false);
      System.out.println("\n[END]...\n");
    }
  }
 };
 computationWorker.execute();
  }
 };
 encryptionWorker.execute();
  }
};
 keysWorker.execute();
 }
};  encodeWorker.execute();
  }
 }
}else if(mode == 3){
 int checker = 0;
 try{
 BufferedReader inputStream = new BufferedReader(new
         ↪ FileReader(fileName1));
 String line = "";
 while((line=inputStream.readLine()) != null){
   if(!line.contains("#")){
     lines.add(line);
   }else{
     noOfSamples = countSamples(line); // count the number of
             ↪ samples declared in the header line
   }
 }
 inputStream.close();

 inputStream = new BufferedReader(new FileReader(fileName2));
 line = "";
 while((line=inputStream.readLine()) != null){
   if(!line.contains("#")){ lines2.add(line);
   }else{
     noOfSamples2 = countSamples(line); // count the number of
             ↪ samples declared in the header line
   }
 }
 inputStream.close();

 statusBoard.append(">Preparing dataset..." + "\n");

 noOfSNPs = lines.size()/2;

 if(setSamples(snps, lines, noOfSamples) && setSamples(snps2,
         ↪ lines2, noOfSamples2) && (noOfSamples == noOfSamples2)
         ↪ && (lines.size() == lines2.size()) &&
         ↪ alleleCheck(snps, snps2)){
   checker = 1;
 }else{
   checker = 0;
   statusBoard.append(">Invalid input file(s) 1." + "\n");
 }
}catch(IOException e){
 statusBoard.append(">No/Invalid input file(s)." + "\n");
}

if(checker == 1){
 encodeWorker = new SwingWorker<Boolean, String>() {
   @Override
   protected Boolean doInBackground() throws Exception {
     System.out.println("[CHISQ]\n");

     publish("Encoding dataset...");
     Thread.sleep(1000);
     for(int i = 0; i < snps.size(); i++){
       char fallele = snps.get(i).getFallele().charAt(0);
       for(int j = 0; j < snps.get(i).samples.size(); j++){
         int code = 0;
         if(snps.get(i).samples.get(j).charAt(0) == fallele){
           code++;
         }
         if(snps.get(i).samples.get(j).charAt(1) == fallele){
           code++;
         }
         snps.get(i).addEncoding(code);

         String line = "[Control group] Genotype sample no. " +
                 ↪ (j+1) + " of SNP #" + (i+1) + " has been
                 ↪ encoded.";
         publish(line);
         //Thread.sleep(100);
       }
     }

     for(int i = 0; i < snps2.size(); i++){
       char fallele = snps2.get(i).getFallele().charAt(0);
       for(int j = 0; j < snps2.get(i).samples.size(); j++){
         int code = 0;
         if(snps2.get(i).samples.get(j).charAt(0) == fallele){
           code++;
         }
         if(snps2.get(i).samples.get(j).charAt(1) == fallele){
```

```
        code++;
      }
      snps2.get(i).addEncoding(code);

      String line = "[Case group] Genotype sample no. " + (j+1)
              ↪ + " of SNP #" + (i+1) + " has been encoded.";
      publish(line);
      //Thread.sleep(100);
    }
  }
return true;
    }
    @Override
    protected void process(List<String> msgs) {
      for(String msg: msgs){
        statusBoard.append(">" + msg + "\n");
      }
    }
    @Override
    protected void done() {
      keysWorker = new SwingWorker<Boolean, String>() {
      @Override
      protected Boolean doInBackground() throws Exception {
        publish("Encoding complete.");
        publish("Generating keys...");
        Thread.sleep(1000);

        startTime = System.currentTimeMillis();
        System.out.println("Key Generation started...");

        sk = new PrivateKey(64);
        pk = sk.getPublicKey();

        endTime = System.currentTimeMillis();
        totalTime = endTime-startTime;
        System.out.println("Key Generation completed in " +
                ↪ totalTime/1000 + "s.");

        String line = "Private and Public keys have been
                ↪ generated.";
        publish(line);
        return true;
           }
           @Override
           protected void process(List<String> msgs) {
             for(String msg: msgs){
               statusBoard.append(">" + msg + "\n");
             }
           }
           @Override
           protected void done() {
             encryptionWorker = new SwingWorker<Boolean,
                     ↪ String>() {
            @Override
            protected Boolean doInBackground() throws Exception {
              publish("Key generation complete.");
              publish("Encrypting dataset...");
              Thread.sleep(1000);

              startTime = System.currentTimeMillis();
              System.out.println("Encryption started...");

              for(int i = 0; i < snps.size(); i++){
                for(int j = 0; j <
                        ↪ snps.get(i).encodedSamples.size();
                        ↪ j++){
                  BigInteger code = new
                          ↪ BigInteger(snps.get(i).encodedSamples.get(j).toString());
                  EncryptedInteger enc = new
                          ↪ EncryptedInteger(code,
                          ↪ sk.getPublicKey());
                  snps.get(i).encryptedSamples.add(enc);

                  String line = "[Control group] Genotype sample
                          ↪ no. " + (j+1) + " of SNP #" +
                          ↪ (i+1) + " has been
                          ↪ encrypted.\n(Encrpted Value: " +
                          ↪ enc.getCipherVal() + ")";
                  publish(line);
                  //Thread.sleep(100);
                }
              }
              for(int i = 0; i < snps2.size(); i++){
                for(int j = 0; j <
                        ↪ snps2.get(i).encodedSamples.size();
                        ↪ j++){
                  BigInteger code = new
                          ↪ BigInteger(snps2.get(i).encodedSamples.get(j).toString());
                  EncryptedInteger enc = new
                          ↪ EncryptedInteger(code,
                          ↪ sk.getPublicKey());
                  snps2.get(i).encryptedSamples.add(enc);

                  String line = "[Case group] Genotype sample no.
                          ↪ " + (j+1) + " of SNP #" + (i+1) +
                          ↪ " has been encrypted.\n(Encrpted
                          ↪ Value: " + enc.getCipherVal() +
                          ↪ ")";
                  publish(line);
                  //Thread.sleep(100);
                }
              }

              endTime = System.currentTimeMillis();
              totalTime = endTime - startTime;
              System.out.println("Encryption completed in " +
                      ↪ totalTime/1000 + "s.");

              return true;
                 }
                 @Override
                 protected void process(List<String> msgs) {
                   for(String msg: msgs){
                     statusBoard.append(">" + msg + "\n");
                   }
}
@Override
protected void done() {
  computationWorker = new SwingWorker<Boolean,
          ↪ String>() {
   @Override
   protected Boolean doInBackground() throws
           ↪ Exception {
    ArrayList<ArrayList<EncryptedInteger>>
            ↪ SNPsEncSamples = new
            ↪ ArrayList<ArrayList<EncryptedInteger>>();
    ArrayList<ArrayList<EncryptedInteger>>
            ↪ SNPsEncSamples2 = new
            ↪ ArrayList<ArrayList<EncryptedInteger>>();
    ArrayList<EncryptedInteger> encResult = new
            ↪ ArrayList<EncryptedInteger>();
    ArrayList<ArrayList<Integer>> SNPsSamples =
            ↪ new ArrayList<ArrayList
            ↪ <Integer>>();
    ArrayList<ArrayList<Integer>> SNPsSamples2 =
            ↪ new ArrayList<ArrayList
            ↪ <Integer>>();
    ArrayList<Integer> result = new
            ↪ ArrayList<Integer>();
    int encIndex = 0;
    int index = 0;

    SNPsCipherVal = new
            ↪ ArrayList<ArrayList<String>>();
    SNPsCipherVal2 = new
            ↪ ArrayList<ArrayList<String>>();
    ArrayList<String> strResult = new
            ↪ ArrayList<String>();

    publish("Encryption Complete.");
    publish("Uploading dataset...");
    Thread.sleep(1000);

    for(int i = 0; i < snps.size(); i++){
      SNPsEncSamples.add(snps.get(i).encryptedSamples);
      SNPsEncSamples2.add(snps2.get(i).encryptedSamples);
      SNPsSamples.add(snps.get(i).encodedSamples);
      SNPsSamples2.add(snps2.get(i).encodedSamples);
    }

    ArrayList<String> container = new
            ↪ ArrayList<String>();

    //first element of the
            ↪ ArrayList<ArrayList<String>>
            ↪ SNPsCipherVal
    container.add(n);
    container.add(pk.getN().toString());
    container.add("3");
    SNPsCipherVal.add(container);
    SNPsCipherVal2.add(container);

    for(int i = 0; i < SNPsEncSamples.size();
            ↪ i++){
      container = new ArrayList<String>();
      for(int j = 0; j <
              ↪ SNPsEncSamples.get(i).size();
              ↪ j++){
        container.add(SNPsEncSamples.get(i).get(j).getCipherVal().toString());
      }
      SNPsCipherVal.add(container);
    }

    for(int i = 0; i < SNPsEncSamples2.size();
            ↪ i++){
      container = new ArrayList<String>();
      for(int j = 0; j <
              ↪ SNPsEncSamples2.get(i).size();
              ↪ j++){
        container.add(SNPsEncSamples2.get(i).get(j).getCipherVal().toString());
      }
      SNPsCipherVal2.add(container);
    }

    try {
      clientOutputStream.writeObject(SNPsCipherVal);
      clientOutputStream.writeObject(SNPsCipherVal2);
      clientOutputStream.flush();

      publish("Dataset has been uploaded.");
      publish("Downloading computation results
              ↪ from the server...");
      Thread.sleep(1000);

      strResult =
              ↪ (ArrayList<String>)clientInputStream.readObject();

      publish("Computation results have been
              ↪ downloaded.");
      Thread.sleep(1000);
    }catch(SocketException sockEx) {
      publish("Connection failure: CHISQ cannot
              ↪ be completed.");
      connected = 0;
    }catch(IOException ioe) {
      ioe.printStackTrace();
    }

    if(connected == 1){
      for(int i = 0; i < strResult.size(); i++){
        EncryptedInteger encInt = new
                ↪ EncryptedInteger(pk);
        encInt.setCipherVal(new
                ↪ BigInteger(strResult.get(i)));
        encResult.add(encInt);
      }

      result =
              ↪ ordinaryCHIComputation(SNPsSamples,
              ↪ SNPsSamples2);
```

```
                startTime = System.currentTimeMillis();
                System.out.println("Decryption
                        ↪ started...");
                for(int i = 0; i < snps.size(); i++){
                  BigDecimal dalpha = new
                          ↪ BigDecimal(encResult.get(encIndex).decrypt(sk));
                          ↪ encIndex++;
                  if(dalpha.doubleValue() > 400){
                    dalpha = dalpha.subtract(new
                            ↪ BigDecimal(pk.getN().toString()));
                  }
                  snps.get(i).dalpha2 =
                          ↪ dalpha.doubleValue();
                  BigDecimal dbeta = new
                          ↪ BigDecimal(encResult.get(encIndex).decrypt(sk));
                          ↪ encIndex++;
                  snps.get(i).dbeta2 = dbeta.doubleValue();
                  BigDecimal ddelta = new
                          ↪ BigDecimal(encResult.get(encIndex).decrypt(sk));
                          ↪ encIndex++;
                  snps.get(i).ddelta2 =
                          ↪ ddelta.doubleValue();
                  BigDecimal dtotalAllele = new
                          ↪ BigDecimal(4*snps.get(i).encryptedSamples.size());
                  double num =
                          ↪ dtotalAllele.doubleValue()*Math.pow(alpha.doubleValue(),
                          ↪ 2);
                  double denom =
                          ↪ dbeta.doubleValue()*ddelta.doubleValue();
                  snps.get(i).echi = num/denom;
                }
                endTime = System.currentTimeMillis();
                totalTime = endTime-startTime;
                System.out.println("Decryption completed
                        ↪ in " + totalTime/1000 + "s.");
                for(int i = 0; i < snps.size(); i++){
                  double alpha = result.get(index);
                          ↪ index++;
                  double beta = result.get(index); index++;
                  double delta = result.get(index);
                          ↪ index++;
                  double totalAllele =
                          ↪ 4*(snps.get(i).encodedSamples.size());
                  double num = totalAllele*Math.pow(alpha,
                          ↪ 2);
                  double denom = beta*delta;
                  snps.get(i).chi = num/denom;
                }
                for(int i = 0; i < snps.size(); i++){
                  if(snps.get(i).echi != snps.get(i).chi){
                    mismatchRate++;
                  }
                }
                mismatchRate = mismatchRate/snps.size();
                displayResults();
              }
              return true;
                 }
                 @Override
                 protected void process(List<String> msgs)
                         ↪ {
                   for(String msg: msgs){
                     statusBoard.append(">" + msg + "\n");
                   }
                 }
                 @Override
                 protected void done() {
                   if(connected == 1){
                     statusBoard.append(">CHISQS
                             ↪ Computation Complete |
                             ↪ Mismatch Rate: " +
                             ↪ mismatchRate + "% | Key
                             ↪ Size: " + n + "bit | SNP
                             ↪ Counts: " + noOfSNPs);
                      System.out.println("\n[END]...\n");
                   }else{
                     status.setText("Offline");
                     computeButton.setEnabled(false);
                     System.out.println("\n[END]...\n");
                   }
                 }
               };
               computationWorker.execute();
                }
             };
             encryptionWorker.execute();
              }
          };
               keysWorker.execute();
           }
       };
            encodeWorker.execute();
     }
   }
 }
});


cancelButton = new JButton("Cancel");
cancelButton.setBounds(665,5,100,40);
cancelButton.setFont(new Font("Tahoma", Font.BOLD, 12));
cancelButton.setCursor(new Cursor(Cursor.HAND_CURSOR));
cancelButton.addActionListener(new ActionListener() {
 @Override
 public void actionPerformed(ActionEvent arg0) {
   terminated = 1;
   fileName1 = "";
   fileName2 = "";
   snps = new ArrayList<Snp>();
   snps2 = new ArrayList<Snp>();
   file1Button.setText("Dataset (1)");
   file2Button.setEnabled(false);
   file2Button.setText("Dataset (2)");
   if(encodeWorker.isDone()){
     if(keysWorker.isDone()){
       if(encryptionWorker.isDone()){
         computationWorker.cancel(true);
       }else{
         encryptionWorker.cancel(true);
       }
     }else{
       keysWorker.cancel(true);
     }
   }else{
     encodeWorker.cancel(true);
   }
   statusBoard.append(">Computation terminated." + "\n");
 }
});
panelA.add(modeOfComp);
panelA.add(file1Button);
panelA.add(file2Button);
panelA.add(computeButton);
panelA.add(cancelButton);
panelC.add(statusScroll);
panelB.add(resultScroll);
frame.add(panelA);
frame.add(panelB);
frame.add(panelC);
frame.add(connectTo);
frame.add(port);
frame.add(ipAdd);
frame.add(portNo);
frame.add(connection);
frame.add(status);
frame.add(connect);

frame.repaint();
frame.validate();
}

/*
 * Displays the computation results at the resultBoard
 * @return void
 * */
static void displayResults(){
  model = new DefaultTableModel();

   computationResult = new JTable(model);
  computationResult.setDefaultEditor(Object.class, null);
  computationResult.setAutoResizeMode(JTable.AUTO_RESIZE_OFF);
  computationResult.setBounds(0, 0, 775, 35);

   TableColumnModel columnModel = computationResult.getColumnModel();
   DefaultTableCellRenderer centerRenderer = new DefaultTableCellRenderer();
   centerRenderer.setHorizontalAlignment(JLabel.CENTER);

   ((DefaultTableCellRenderer)computationResult.getTableHeader().getDefaultRenderer())
   .setHorizontalAlignment(JLabel.CENTER);

  if(mode == 1){
    model.addColumn("RSID");
     model.addColumn("ALLELE1");
     model.addColumn("ALLELE2");
     model.addColumn("HMAF");
     model.addColumn("MAF");

     columnModel.getColumn(0).setPreferredWidth(167);
     columnModel.getColumn(0).setCellRenderer(centerRenderer);
     columnModel.getColumn(1).setPreferredWidth(150);
     columnModel.getColumn(1).setCellRenderer(centerRenderer);
     columnModel.getColumn(2).setPreferredWidth(150);
     columnModel.getColumn(2).setCellRenderer(centerRenderer);
     columnModel.getColumn(3).setPreferredWidth(150);
     columnModel.getColumn(3).setCellRenderer(centerRenderer);
     columnModel.getColumn(4).setPreferredWidth(150);
     columnModel.getColumn(4).setCellRenderer(centerRenderer);

     for(int i = 0; i < snps.size(); i++){
      model.addRow(new Object[]{snps.get(i).getRsid(), snps.get(i).fallele,
              ↪ snps.get(i).sallele, snps.get(i).emaf, snps.get(i).maf});
     }
  }else if(mode == 2){
    model.addColumn("RSID");
     model.addColumn("ALLELE1");
     model.addColumn("ALLELE2");
     model.addColumn("HHWE");
     model.addColumn("HWE");

     columnModel.getColumn(0).setPreferredWidth(167);
     columnModel.getColumn(0).setCellRenderer(centerRenderer);
     columnModel.getColumn(1).setPreferredWidth(150);
     columnModel.getColumn(1).setCellRenderer(centerRenderer);
     columnModel.getColumn(2).setPreferredWidth(150);
     columnModel.getColumn(2).setCellRenderer(centerRenderer);
     columnModel.getColumn(3).setPreferredWidth(150);
     columnModel.getColumn(3).setCellRenderer(centerRenderer);
     columnModel.getColumn(4).setPreferredWidth(150);
     columnModel.getColumn(4).setCellRenderer(centerRenderer);

     for(int i = 0; i < snps.size(); i++){
      model.addRow(new Object[]{snps.get(i).getRsid(), snps.get(i).fallele,
              ↪ snps.get(i).sallele, snps.get(i).ehwe, snps.get(i).hwe});
     }
  }else if(mode == 3){
```

```
    model.addColumn("RSID");
      model.addColumn("ALLELE1");
      model.addColumn("ALLELE2");
      model.addColumn("HCHI");
      model.addColumn("CHI");

      columnModel.getColumn(0).setPreferredWidth(167);
      columnModel.getColumn(0).setCellRenderer(centerRenderer);
      columnModel.getColumn(1).setPreferredWidth(150);
      columnModel.getColumn(1).setCellRenderer(centerRenderer);
      columnModel.getColumn(2).setPreferredWidth(150);
      columnModel.getColumn(2).setCellRenderer(centerRenderer);
      columnModel.getColumn(3).setPreferredWidth(150);
      columnModel.getColumn(3).setCellRenderer(centerRenderer);
      columnModel.getColumn(4).setPreferredWidth(150);
      columnModel.getColumn(4).setCellRenderer(centerRenderer);

      for(int i = 0; i < snps.size(); i++){
       model.addRow(new Object[]{snps.get(i).getRsid(), snps.get(i).fallele,
              ↪ snps.get(i).sallele, snps.get(i).echi, snps.get(i).chi});
    }
  }

   resultScroll.remove(resultBoard);
   resultScroll.getViewport ().add(computationResult);
   resultScroll.repaint();
}

/*
 * Performs ordinary HWE Computation
 * @return ArrayList of Integers containing the alpha, beta, delta, omega,
        ↪ theta
 * */
static ArrayList<Integer> ordinaryHWEComputation(ArrayList<ArrayList
        ↪ <Integer>> SNPsSamples){
  ArrayList<Integer> result = new ArrayList<Integer>();

  startTime = System.currentTimeMillis();
  System.out.println("HWE Computation started...");

  for(int i = 0; i < SNPsSamples.size(); i++){
    Integer alpha = 0;
    Integer beta = 0;
    Integer delta = 0;
    Integer omega = 0;
    Integer theta = 0;

    for(int j = 0; j < SNPsSamples.get(i).size()-2; j++){
      alpha += SNPsSamples.get(i).get(j);
      j++;
      beta += SNPsSamples.get(i).get(j);
      j++;
      delta += SNPsSamples.get(i).get(j);
    }

    omega = (2*alpha) + beta;
    theta = (2*delta) + beta;

    result.add(alpha);
    result.add(beta);
    result.add(delta);
    result.add(omega);
    result.add(theta);
  }

  endTime = System.currentTimeMillis();
  totalTime = endTime-startTime;
  System.out.println("HWE Computation completed in " + totalTime/1000 + "s.");

  return result;
}


/*
 * Performs ordinary MAF Computation
 * @return ArrayList of Integers containing the falleleCount and sAlleleCount
 * */
static ArrayList<Integer> ordinaryMAFComputation(ArrayList<ArrayList
        ↪ <Integer>> SNPsSamples){
  ArrayList<Integer> result = new ArrayList<Integer>();

  startTime = System.currentTimeMillis();
  System.out.println("MAF Computation started...");

  for(int i = 0; i < SNPsSamples.size(); i++){
    Integer fAlleleCount = 0;
    Integer sAlleleCount = 0;

    for(int j = 0; j < SNPsSamples.get(i).size(); j++){
      fAlleleCount += SNPsSamples.get(i).get(j);
    }
    Integer totalAllele = 2*SNPsSamples.get(i).size();
    sAlleleCount = totalAllele - fAlleleCount;

    result.add(fAlleleCount);
    result.add(sAlleleCount);
  }

  endTime = System.currentTimeMillis();
  totalTime = endTime-startTime;
  System.out.println("MAF Computation completed in " + totalTime/1000 + "s.");

  return result;
}

/*
 * Performs ordinary Chi Square Test Statistic Computation
 * @return ArrayList of Integers containing the alpha, beta, delta
 * */
static ArrayList<Integer> ordinaryCHIComputation(ArrayList<ArrayList
        ↪ <Integer>> SNPsSamples, ArrayList<ArrayList <Integer>>
        ↪ SNPsSamples2){
  ArrayList<Integer> result = new ArrayList<Integer>();

  startTime = System.currentTimeMillis();
  System.out.println("CHISQ Computation started...");

  for(int i = 0; i < SNPsSamples.size(); i++){
    Integer fAlleleCount = 0;
    Integer fAlleleCount2 = 0;
    Integer totalAllele = 2*SNPsSamples.get(i).size();
    Integer alpha = 0;
    Integer beta = 0;
    Integer delta = 0;

    for(int j = 0; j < SNPsSamples2.get(i).size(); j++){
      fAlleleCount += SNPsSamples2.get(i).get(j);
    }

    for(int j = 0; j < SNPsSamples.get(i).size(); j++){
      fAlleleCount2 += SNPsSamples.get(i).get(j);
    }

    alpha = fAlleleCount - fAlleleCount2;
    beta = fAlleleCount + fAlleleCount2;
    delta = 2*totalAllele-(beta);

    result.add(alpha);
    result.add(beta);
    result.add(delta);
  }

  endTime = System.currentTimeMillis();
  totalTime = endTime-startTime;
  System.out.println("CHISQ Computation completed in " + totalTime/1000 +
          ↪ "s.");

  return result;
}

/*
 * Encrypts each encoded MAF sample of a SNP
 * @return void
 *
 * */
static void encryptSamplesMAF(ArrayList<Snp> snps, PrivateKey sk) throws
        ↪ BigIntegerClassNotValid{
  for(int i = 0; i < snps.size(); i++){
    for(int j = 0; j < snps.get(i).encodedSamples.size(); j++){
      BigInteger code = new
              ↪ BigInteger(snps.get(i).encodedSamples.get(j).toString());
      EncryptedInteger enc = new EncryptedInteger(code, sk.getPublicKey());
      snps.get(i).encryptedSamples.add(enc);

      String line = "Genotype sample no. " + (j+1) + " of SNP #" + (i+1) + "
              ↪ has been encrypted.\n(Encrpted Value: " +
              ↪ enc.getCipherVal() + ")";
      System.out.println(line);
      statusBoard.append(">" + line + "\n");
    }
  }
}

/*
 * Encrypts each encoded HWE sample of a SNP
 * @return void
 *
 * */
static void encryptSamplesHWE(ArrayList<Snp> snps, PrivateKey sk) throws
        ↪ BigIntegerClassNotValid{
  for(int k = 0; k < snps.size(); k++){
    for(int l = 0; l < snps.get(k).encodedSamples2.size(); l++){
      BigInteger code2 = new
              ↪ BigInteger(snps.get(k).encodedSamples2.get(l).toString());
      EncryptedInteger enc2 = new EncryptedInteger(code2, sk.getPublicKey());
      snps.get(k).encryptedSamples2.add(enc2);

      String line = "Genotype sample no. " + (l+1) + " of SNP #" + (k+1) + "
              ↪ has been encrypted.\n(Encrpted Value: " +
              ↪ enc2.getCipherVal() + ")";
      System.out.println(line);
      statusBoard.append(">" + line + "\n");
    }
  }
}

/*
 * Encodes each sample of a SNP to integer format compatible for HWE
        ↪ computation
 * @return void
 * */
static void encodeSamplesHWE(ArrayList<Snp> snps){
  //encode for hwe
  for(int i = 0; i < snps.size(); i++){
    char fallele = snps.get(i).getFallele().charAt(0);
    for(int j = 0; j < snps.get(i).samples.size(); j++){
      int code = 0;
      if(snps.get(i).samples.get(j).charAt(0) == fallele){
        code++;
      }
      if(snps.get(i).samples.get(j).charAt(1) == fallele){
        code++;
      }

      switch(code){
      case 0: snps.get(i).encodedSamples2.add(0);
           snps.get(i).encodedSamples2.add(0);
           snps.get(i).encodedSamples2.add(1);
           break;
      case 1: snps.get(i).encodedSamples2.add(0);
           snps.get(i).encodedSamples2.add(1);
           snps.get(i).encodedSamples2.add(0);
           break;
      case 2: snps.get(i).encodedSamples2.add(1);
           snps.get(i).encodedSamples2.add(0);
           snps.get(i).encodedSamples2.add(0);
           break;
      }
```

```
        String line = "Genotype sample no. " + (j+1) + " of SNP #" + (i+1) + "
            ↪ has been encoded.";
        System.out.println(line);
        statusBoard.append(">" + line + "\n");
      }
    }
  }

  /*
   * Encodes each sample of a SNP to integer format compatible for MAF
       ↪ computation
   * @return void
   * */
  static void encodeSamplesMAF(ArrayList<Snp> snps){
    //encode for maf
    for(int i = 0; i < snps.size(); i++){
      char fallele = snps.get(i).getFallele().charAt(0);
      for(int j = 0; j < snps.get(i).samples.size(); j++){
        int code = 0;
        if(snps.get(i).samples.get(j).charAt(0) == fallele){
          code++;
        }
        if(snps.get(i).samples.get(j).charAt(1) == fallele){
          code++;
        }
        snps.get(i).addEncoding(code);

        String line = "Genotype sample no. " + (j+1) + " of SNP #" + (i+1) + "
            ↪ has been encoded.";
        System.out.println(line);
        statusBoard.append(">" + line + "\n");
      }
    }

  }

  /* Constructs Snps
   * Set rsid
   * Set sampleCount
   * Set genotype samples per Snp
   * Set fallele and sallele
   * @return an ArrayList of Snps with rsid, genotype samples, fallele, sallele
   * */
  static boolean setSamples(ArrayList<Snp> snps, ArrayList<String> lines, int
      ↪ noOfSamples){
    for(int i = 0; i < lines.size(); i++){
      Snp snp = new Snp(lines.get(i));
      snp.setSampleCount(noOfSamples);
      i++;

      int lineIndex = 0;
      int llength = lines.get(i).length();
      String fallele = "";
      String sallele = "";

      for(int j = 0; j < noOfSamples; j++){
        String genotype = "";

        try{
          if(lines.get(i).charAt(lineIndex) == 'A' ||
              ↪ lines.get(i).charAt(lineIndex) == 'C' ||
              ↪ lines.get(i).charAt(lineIndex) == 'T' ||
              ↪ lines.get(i).charAt(lineIndex) == 'G'){
            genotype += lines.get(i).charAt(lineIndex); // first allele
            lineIndex++;
          }else{
            return false;
          }

          if(lines.get(i).charAt(lineIndex) == 'A' ||
              ↪ lines.get(i).charAt(lineIndex) == 'C' ||
              ↪ lines.get(i).charAt(lineIndex) == 'T' ||
              ↪ lines.get(i).charAt(lineIndex) == 'G'){
            genotype += lines.get(i).charAt(lineIndex);
            lineIndex++;
          }else{
            return false;
          }

          if(lineIndex < llength){
            if(lines.get(i).charAt(lineIndex) == ' '){
              lineIndex++;
            }
          }
        }catch(StringIndexOutOfBoundsException e){
          return false;
        }
        //assigning the first and second allele
        if(fallele == ""){
          fallele += genotype.charAt(0);
          snp.setFallele(fallele);
          if(genotype.charAt(0) != genotype.charAt(1)){
            sallele += genotype.charAt(1);
            snp.setSallele(sallele);
          }
        }else if(sallele == ""){
          if(fallele.charAt(0) != genotype.charAt(0)){
            sallele += genotype.charAt(0);
            snp.setSallele(sallele);
          }else if(fallele.charAt(0) != genotype.charAt(1)){
            sallele += genotype.charAt(1);
            snp.setSallele(sallele);
          }
        }

        snp.addSample(genotype); // add a new genotype (first allele + second
            ↪ allele) to the snp
      }
      snps.add(snp);
    }
    return true;
  }

  /*
   * Checks if the alleles of each control and case SNP matches
   * @return true if the alleles of each control and case SNP matches, else false
   * */
  static boolean alleleCheck(ArrayList<Snp> snps, ArrayList<Snp> snps2){
    if(snps.size() != snps2.size()){
      System.out.println("ping");
      return false;
    }else{
      for(int i = 0; i < snps.size(); i++){
        if(snps.get(i).fallele.charAt(0) != snps2.get(i).fallele.charAt(0)){
          if(snps.get(i).fallele.charAt(0) != snps2.get(i).sallele.charAt(0)){
            System.out.println(snps.get(i).fallele.charAt(0) + " | " +
                ↪ snps2.get(i).fallele.charAt(0));
            System.out.println(snps.get(i).rsid);
            System.out.println("pong");
            return false;
          }
        }else if(snps.get(i).sallele.charAt(0) !=
            ↪ snps2.get(i).sallele.charAt(0)){
          if(snps.get(i).sallele.charAt(0) != snps2.get(i).fallele.charAt(0)){
            System.out.println(snps.get(i).fallele.charAt(0) + " | " +
                ↪ snps2.get(i).fallele.charAt(0));
            System.out.println(snps.get(i).rsid);
            System.out.println("pong");
            return false;
          }
        }
      }
    }
    return true;
  }

  /*
   * Counts the number of sample genotypes per SNP
   * @return void
   * */
  static int countSamples(String cases){
    int samples = 0;
    for(int i = 0; i < cases.length(); i++){
      if(cases.charAt(i) == '#'){
        samples++;
      }
    }return samples;
  }

  public static void main(String[] args) throws ClassNotFoundException,
      ↪ InstantiationException, IllegalAccessException,
      ↪ UnsupportedLookAndFeelException, InterruptedException,
      ↪ UnknownHostException, IOException {
    openWindow();
  }

}
```

2. GWASServer.java

```
import java.io.IOException;
import java.io.ObjectInputStream;
import java.io.ObjectOutputStream;
import java.io.Serializable;
import java.math.BigInteger;
import java.net.ServerSocket;
import java.net.Socket;
import java.net.SocketException;
import java.text.DecimalFormat;
import java.util.ArrayList;

import thep.paillier.EncryptedInteger;
import thep.paillier.PrivateKey;
import thep.paillier.PublicKey;
import thep.paillier.exceptions.BigIntegerClassNotValid;
import thep.paillier.exceptions.PublicKeysNotEqualException;


public class GWASServer implements Serializable, Runnable{
  private static final long serialVersionUID = 1L;
  private static Socket connectionSocket;

  GWASServer(Socket connectionSocket){
    GWASServer.connectionSocket = connectionSocket;
  }

  public static void main(String[] args) throws IOException,
      ↪ ClassNotFoundException, BigIntegerClassNotValid,
      ↪ PublicKeysNotEqualException {

    try {
      ServerSocket serverSocket = new ServerSocket(65432);
      System.out.println("Listening...");
      while(true){
        Socket csocket = serverSocket.accept();
        System.out.println(csocket.getInetAddress() + " connected.");
        new Thread(new GWASServer(csocket)).start();
      }
    }catch(SocketException sockEx) {
      System.out.println("Unable to attach to port");
      System.exit(1);
    }
    catch(IOException ioEx) {
      ioEx.printStackTrace();
    }
  }

  @Override
  public void run() {
    ArrayList<ArrayList<EncryptedInteger>> clientDatasets;
    ArrayList<ArrayList<EncryptedInteger>> clientDatasets2;
    PublicKey pk = null;
    int mode = 0;
    String ipAdd = connectionSocket.getInetAddress().toString();

    long startTime = 0;
    long endTime = 0;
    double totalTime = 0;
```

```
  ObjectInputStream serverInputStream = null;
  ObjectOutputStream serverOutputStream = null;

  ArrayList<ArrayList<String>> clientDatasetsStr;
  ArrayList<ArrayList<String>> clientDatasetsStr2;


  try {
    serverInputStream = new
            ↪ ObjectInputStream(connectionSocket.getInputStream());
    serverOutputStream = new
            ↪ ObjectOutputStream(connectionSocket.getOutputStream());

    while(mode != 4){
      clientDatasetsStr =
              ↪ (ArrayList<ArrayList<String>>)serverInputStream.readObject();
      clientDatasetsStr2 =
              ↪ (ArrayList<ArrayList<String>>)serverInputStream.readObject();

      int n = Integer.valueOf(clientDatasetsStr.get(0).get(0));
      BigInteger N = new BigInteger(clientDatasetsStr.get(0).get(1));
      mode = Integer.valueOf(clientDatasetsStr.get(0).get(2));
      pk = new PublicKey(n, N);

      ArrayList<EncryptedInteger> computationResult = new
              ↪ ArrayList<EncryptedInteger>();
      ArrayList<String> computationResultStr = new ArrayList<String>();

      if(mode == 1){
        System.out.println("\nHMAF Computation started...");
        startTime = System.currentTimeMillis();
        computationResultStr = privateMAFComputation(clientDatasetsStr, pk);
        endTime = System.currentTimeMillis();
        totalTime = endTime-startTime;
        System.out.println("HMAF computation completed in " + totalTime/1000
                ↪ + "s.");
      }else if(mode == 2){
        System.out.println("\nHHWE Computation started...");
        startTime = System.currentTimeMillis();
        computationResultStr = privateHWEComputation(clientDatasetsStr, pk);
        endTime = System.currentTimeMillis();
        totalTime = endTime-startTime;
        System.out.println("HHWE Computation completed in " + totalTime/1000
                ↪ + "s.");
      }else if(mode == 3){
        System.out.println("\nHCHISQ Computation started...");
        startTime = System.currentTimeMillis();
        computationResultStr = privateCHIComputation(clientDatasetsStr,
                ↪ clientDatasetsStr2, pk);
        endTime = System.currentTimeMillis();
        totalTime = endTime-startTime;
        System.out.println("HCHISQ Computation completed in " +
                ↪ totalTime/1000 + "s.");
      }else if(mode == 4){
        break;
      }

      serverOutputStream.writeObject(computationResultStr);
      serverOutputStream.flush();
    }

    serverInputStream.close();
    serverOutputStream.close();
    connectionSocket.close();
  }
  catch(SocketException sockEx) {
    System.out.println(ipAdd + " disconnected.");
  }
  catch(IOException ioEx) {
    ioEx.printStackTrace();
  } catch (BigIntegerClassNotValid e) {
    // TODO Auto-generated catch block
    e.printStackTrace();
  } catch (PublicKeysNotEqualException e) {
    // TODO Auto-generated catch block
    e.printStackTrace();
  } catch (ClassNotFoundException e) {
    // TODO Auto-generated catch block
    e.printStackTrace();
  }

}

/*
 * Performs Homomorphic Chi Square Test Statistic Computation
 * @return ArrayList of String containing the cipherVal of encrypted
         ↪ fAlleleCount and sAlleleCount
 * */
static ArrayList<String> privateCHIComputation(ArrayList<ArrayList<String>>
        ↪ SNPsCipherVal, ArrayList<ArrayList<String>> SNPsCipherVal2,
        ↪ PublicKey pk) throws BigIntegerClassNotValid,
        ↪ PublicKeysNotEqualException{
  ArrayList<String> result = new ArrayList();
  ArrayList<ArrayList<EncryptedInteger>> SNPsEncSamples = new
          ↪ ArrayList<ArrayList<EncryptedInteger>>();
  ArrayList<ArrayList<EncryptedInteger>> SNPsEncSamples2 = new
          ↪ ArrayList<ArrayList<EncryptedInteger>>();
  EncryptedInteger alpha = new EncryptedInteger(new BigInteger("0"), pk);
  EncryptedInteger beta = new EncryptedInteger(new BigInteger("0"), pk);
  EncryptedInteger delta = new EncryptedInteger(new BigInteger("0"), pk);


  for(int i = 1; i < SNPsCipherVal.size(); i++){
    ArrayList<EncryptedInteger> container = new ArrayList<EncryptedInteger>();
    for(int j = 0; j < SNPsCipherVal.get(i).size(); j++){
      EncryptedInteger encInt = new EncryptedInteger(pk);
      encInt.setCipherVal(new BigInteger(SNPsCipherVal.get(i).get(j)));
      container.add(encInt);
    }
    SNPsEncSamples.add(container);
  }

  for(int i = 1; i < SNPsCipherVal2.size(); i++){
    ArrayList<EncryptedInteger> container = new ArrayList<EncryptedInteger>();
    for(int j = 0; j < SNPsCipherVal2.get(i).size(); j++){
      EncryptedInteger encInt = new EncryptedInteger(pk);
      encInt.setCipherVal(new BigInteger(SNPsCipherVal2.get(i).get(j)));
      container.add(encInt);
    }
    SNPsEncSamples2.add(container);
  }

  for(int i = 0; i < SNPsEncSamples.size(); i++){
    EncryptedInteger efAlleleCount = new EncryptedInteger(new BigInteger("0"),
            ↪ pk);
    EncryptedInteger efAlleleCount2 = new EncryptedInteger(new
            ↪ BigInteger("0"), pk);

    for(int j = 0; j < SNPsEncSamples2.get(i).size(); j++){
      efAlleleCount = efAlleleCount.add(SNPsEncSamples2.get(i).get(j));
    }
    for(int j = 0; j < SNPsEncSamples.get(i).size(); j++){
      efAlleleCount2 = efAlleleCount2.add(SNPsEncSamples.get(i).get(j));
    }
    Integer totalAllele = 4*SNPsEncSamples.get(i).size();
    BigInteger fn = new BigInteger(totalAllele.toString());

    alpha = efAlleleCount2.multiply(pk.getN().subtract(new BigInteger("1")));
    alpha = efAlleleCount.add(alpha);
    beta = efAlleleCount.add(efAlleleCount2);
    delta = beta.multiply(pk.getN().subtract(new BigInteger("1")));
    delta = delta.add(fn);

    result.add(alpha.getCipherVal().toString());
    result.add(beta.getCipherVal().toString());
    result.add(delta.getCipherVal().toString());
  }
  return result;
}

/*
 * Performs Homomorphic HWE Computation
 * @return ArrayList of String containing the cipherVal of encrypted alpha,
         ↪ beta, delta, omega, theta
 * */
static ArrayList<String> privateHWEComputation(ArrayList<ArrayList <String>>
        ↪ SNPsCipherVal, PublicKey pk) throws BigIntegerClassNotValid,
        ↪ PublicKeysNotEqualException{
  ArrayList<String> result = new ArrayList<String>();
  ArrayList<ArrayList<EncryptedInteger>> SNPsEncSamples = new
          ↪ ArrayList<ArrayList<EncryptedInteger>>();

  for(int i = 1; i < SNPsCipherVal.size(); i++){
    ArrayList<EncryptedInteger> container = new ArrayList<EncryptedInteger>();
    for(int j = 0; j < SNPsCipherVal.get(i).size(); j++){
      EncryptedInteger encInt = new EncryptedInteger(pk);
      encInt.setCipherVal(new BigInteger(SNPsCipherVal.get(i).get(j)));
      container.add(encInt);
    }
    SNPsEncSamples.add(container);
  }

  for(int i = 0; i < SNPsEncSamples.size(); i++){
    EncryptedInteger ealpha = new EncryptedInteger(new BigInteger("0"), pk);
    EncryptedInteger ebeta = new EncryptedInteger(new BigInteger("0"), pk);
    EncryptedInteger edelta = new EncryptedInteger(new BigInteger("0"), pk);
    EncryptedInteger eomega = new EncryptedInteger(new BigInteger("0"), pk);
    EncryptedInteger etheta = new EncryptedInteger(new BigInteger("0"), pk);

    for(int j = 0; j < SNPsEncSamples.get(i).size()-2; j++){
      ealpha = ealpha.add(SNPsEncSamples.get(i).get(j)); j++;
      ebeta = ebeta.add(SNPsEncSamples.get(i).get(j)); j++;
      edelta = edelta.add(SNPsEncSamples.get(i).get(j));
    }

    eomega = ealpha.multiply(new BigInteger("2"));
    eomega = eomega.add(ebeta);
    etheta = edelta.multiply(new BigInteger("2"));
    etheta = etheta.add(ebeta);

    result.add(ealpha.getCipherVal().toString());
    result.add(ebeta.getCipherVal().toString());
    result.add(edelta.getCipherVal().toString());
    result.add(eomega.getCipherVal().toString());
    result.add(etheta.getCipherVal().toString());
  }
  return result;
}

/*
 * Performs Homomorphic MAF Computation
 * @return ArrayList of String containing the cipherVal of encrypted
         ↪ fAlleleCount and encrypted sAlleleCount
 * */
static ArrayList<String> privateMAFComputation(ArrayList<ArrayList <String>>
        ↪ SNPsCipherVal, PublicKey pk) throws BigIntegerClassNotValid,
        ↪ PublicKeysNotEqualException{
  ArrayList<String> result = new ArrayList<String>();
  ArrayList<ArrayList<EncryptedInteger>> SNPsEncSamples = new
          ↪ ArrayList<ArrayList<EncryptedInteger>>();

  for(int i = 1; i < SNPsCipherVal.size(); i++){
    ArrayList<EncryptedInteger> container = new ArrayList<EncryptedInteger>();
    for(int j = 0; j < SNPsCipherVal.get(i).size(); j++){
      EncryptedInteger encInt = new EncryptedInteger(pk);
      encInt.setCipherVal(new BigInteger(SNPsCipherVal.get(i).get(j)));
      container.add(encInt);
    }
    SNPsEncSamples.add(container);
  }

  for(int i = 0; i < SNPsEncSamples.size(); i++){
    EncryptedInteger efAlleleCount = new EncryptedInteger(new BigInteger("0"),
            ↪ pk);
    EncryptedInteger esAlleleCount = new EncryptedInteger(new BigInteger("0"),
            ↪ pk);

    for(int j = 0; j < SNPsEncSamples.get(i).size(); j++){
      efAlleleCount = efAlleleCount.add(SNPsEncSamples.get(i).get(j));
```

```
        }

        Integer totalAllele = 2*SNPsEncSamples.get(i).size();
        esAlleleCount = efAlleleCount.multiply(pk.getN().subtract(new
                ↪ BigInteger("1")));
        esAlleleCount = esAlleleCount.add(new BigInteger(totalAllele.toString()));

        result.add(efAlleleCount.getCipherVal().toString());
        result.add(esAlleleCount.getCipherVal().toString());
      }

      return result;
    }

}
```

### 3. Snp.java

```
import java.math.BigDecimal;
import java.math.BigInteger;
import java.util.ArrayList;

import thep.paillier.EncryptedInteger;
import thep.paillier.PrivateKey;
import thep.paillier.exceptions.BigIntegerClassNotValid;
import thep.paillier.exceptions.PublicKeysNotEqualException;


public class Snp {
  String rsid = "";
  int sampleCount = 0; // no of sample genotypes per SNP
  String fallele = "";
  String sallele = "";

  double fAlleleCount = 0; // count of 1st allele
  double sAlleleCount = 0; // count of 2nd allele
  double dfAlleleCount = 0; // count of 1st allele
  double dsAlleleCount = 0;

  EncryptedInteger efAlleleCount; // for removal
  EncryptedInteger esAlleleCount;

  double alpha = 0;
  double beta = 0;
  double delta = 0;
  double omega = 0;
  double theta = 0;
  double dalpha = 0;
  double dbeta = 0;
  double ddelta = 0;
  double domega = 0;
  double dtheta = 0;

  double alpha2 = 0;
  double beta2 = 0;
  double delta2 = 0;
  double dalpha2 = 0;
  double dbeta2 = 0;
  double ddelta2 = 0;

  ArrayList<String> samples = new ArrayList(); // sample genotypes container
  ArrayList<Integer> encodedSamples = new ArrayList(); // encoded sample
          ↪ genotypes container for MAF
  ArrayList<Integer> encodedSamples2 = new ArrayList(); // encoded sample
          ↪ genotypes container for HWE
  ArrayList<EncryptedInteger> encryptedSamples = new ArrayList(); // encrypted
          ↪ sample genotypes container for MAF
  ArrayList<EncryptedInteger> encryptedSamples2 = new ArrayList(); // encrypted
          ↪ sample genotypes container for HWE

  double maf = 0;
  double emaf = 0;
  double hwe = 0;
  double ehwe = 0;
  double chi = 0;
  double echi = 0;

  public Snp(String rsid){
    this.rsid = rsid;
  }

  public void setSallele(String sallele){
    this.sallele = sallele;
    return;
  }

  public void setFallele(String fallele){
    this.fallele = fallele;
    return;
  }

  public void setSampleCount(int count){
    this.sampleCount = count;
    return;
  }

  public void addSample(String genotype){
    this.samples.add(genotype);
    return;
  }

  public void addEncoding(Integer code){
    this.encodedSamples.add(code);
    return;
  }

  public String getRsid(){
    return this.rsid;
  }

  public String getFallele(){
    return this.fallele;
  }

  public String getSallele(){
    return this.sallele;
  }

  public static void main(String[] args) {
    // TODO Auto-generated method stub

  }

}
```

# XI. Acknowledgement

My sincerest gratitude is to You Lord for this paper could have not been possible to accomplish without You in my life.

I thank God for giving me a family who has never stopped believing in me. To my grandparents, my aunts and uncles, and of course my Daddy, who shared my dreams and aspirations by always morally and financially supporting me, I know it wasn't always easy but for loving me, I owe you my best thanks. I will never forget.

I thank God for my little turtle brother Ken, my confidant, my number one fan.

I thank God for giving me the chance to work with Sir Richard Bryann Chua. For your deep patience, guidance and understanding, I have no word but thanks.

I thank God for sending me my bestfriends - Leira and Karla - for your concern and sympathy through all the good and bad times I had in my endeavor, my thank you.

I thank God for lending me a short but sweet moment in Garden Plaza with Blanche, Roga, Jesh, and Kit. To GH and all my classmates, for making my stay in UP memorable and worth it, for taking care of me during my saddest days, for the help of making me who I am now, for the glimpse of eternal friendship, thank you!

I thank God for surrounding me with all the people who have always believed that I can surpass all the challenges in this Special Problem.

And to that one special person who may have criticized and scolded me but have always loved me, thank you!

From the humblest to the greatest things I have, for the things I know and for the things I do not know, and for all the unspeakable gifts, to God be the glory.