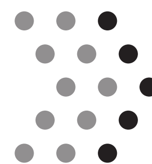
Master's Thesis

Large-Scale Privacy-Preserving Statistical Computations for Distributed Genome-Wide Association Studies

Oleksandr Tkachenko

September 13, 2017



Technische Universität Darmstadt
Center for Research in Security and Privacy
Engineering Cryptographic Protocols

Supervisors: M.Sc. Christian Weinert
Dr. Thomas Schneider
Prof. Dr. Kay Hamacher

Thesis Statement
pursuant to §22 paragraph 7 of APB TU Darmstadt

I herewith formally declare that I have written the submitted thesis independently. I did not use any outside support except for the quoted literature and other sources mentioned in the paper. I clearly marked and separately listed all of the literature and all of the other sources which I employed when producing this academic work, either literally or in content. This thesis has not been handed in or published before in the same or similar form.

In the submitted thesis the written copies and the electronic version are identical in content.

Darmstadt, September 13, 2017

____

Oleksandr Tkachenko

Abstract

This thesis focuses on privacy-preserving algorithms for Genome-Wide Association Studies (GWAS) by applying Secure Multi-Party Computation (SMPC) techniques, which are used to protect the privacy of participants when medical institutions collaborate to compute statistics in a distributed fashion. The existing privacy-preserving solutions for this issue lack efficiency and/or use inadequate algorithms that are of limited practical value.

We implemented multiple algorithms for χ^2-, G- and P-test in the ABY framework for Secure Two-Party Computation and evaluated them in a distributed GWAS scenario. Statistical tests generally require advanced mathematical operations. For the operations that cannot be calculated in integer arithmetic, we made use of the existing IEEE 754 floating point arithmetic implementation in ABY. We improved the efficiency of the algorithms by using protocol conversion in ABY to make use of advantages of multiple SMPC protocols. In order to combine the advantages of the aforementioned techniques, we implemented an integer to floating point conversion functionality in ABY. Furthermore, we extended the contingency table for the χ^2- and G-test to use codeword counts instead of counts for only two alleles. Additionally, we considered an outsourcing scenario where multiple institutions secret-share their data, send it to two non-colluding semi-trusted third parties and receive the secret-shared result. The implemented algorithms are benchmarked and evaluated.

Compared to the prior related work of Constable et al. (BMC Medical Informatics and Decision Making 2015), we improved the run-time efficiency of the χ^2-test by up to factor 35x.

Contents

1 Introduction **1**
1.1 Motivation 2
1.2 Contributions 2
1.3 Outline 3

2 Preliminaries **4**
2.1 Notation 4
2.2 Genomic Primer 6
2.3 Genome-Wide Association Studies 7
2.4 Statistical Tests 7
2.4.1 Chi-squared-test 8
2.4.2 G-test 10
2.4.3 P-test 11
2.5 Secure Multi-Party Computation 11
2.5.1 Oblivious Transfer 12
2.6 ABY Framework 13
2.6.1 Arithmetic Sharing 14
2.6.2 Boolean Sharing 15
2.6.3 SIMD Gates 16
2.6.4 Conversion Gates 17
2.7 Outsourcing Computation 18

3 Related Work **20**
3.1 GWAS using Noise-based Approaches 20
3.2 GWAS using Homomorphic Encryption 21
3.3 GWAS using Yao's Garbled Circuits 21
3.4 GWAS using Other Techniques 22

4 Implementation **23**
4.1 Integer to Floating Point Number Conversion 23
4.2 Chi-squared-test 29
4.2.1 Non-optimized 29
4.2.2 Fully Optimized 30
4.2.3 Partially Optimized 32
4.3 G-test 34

4.3.1 Non-optimized 34
4.3.2 Fully Optimized 36
4.3.3 Partially Optimized 37
4.4 Extended Chi-squared-test and G-test 39
4.5 P-test 39
4.6 Outsourcing Computation 40

5 Evaluation **41**
5.1 Benchmarking Environment 41
5.2 Benchmarking in the LAN Setting 41
5.2.1 Chi-squared-test 41
5.2.2 G-test 42
5.2.3 Comparison with [CTW+15] 43
5.2.4 P-test 45
5.3 Benchmarking in the WAN Setting 46
5.4 Outsourcing Computation 46
5.4.1 Local Share Creation 47
5.4.2 Benchmarking 48
5.5 Communication 51

6 Conclusion **53**
6.1 Summary and Discussion 53
6.2 Future Work 54

List of Abbreviations **57**

Bibliography **59**

1 Introduction

In 2000, the first human genome analysis took 9 months and cost 100 million *US-Dollars (USD)*. Nowadays, genome sequencing is much more affordable, it costs just about 2000 USD and requires 15 minutes of computation [WW16]. Since genome sequencing is becoming more efficient and genomic data is getting collected widely, the research community wants to conduct analyses on this data for many reasons. One of the possible applications of genomic data is investigating the associations between diseases and specific parts of the genome. Nevertheless, genomic data is highly sensible and identifies its biological owner with a very high probability. This is why it must not be disclosed to the public. Disclosing genomic data could lead to disadvantaging people with predispositions to some diseases, also known as genetic discrimination. For example, a health insurance fund could increase the fee or even decline the client's application because of "bad" genes. Genetic discrimination can be an obstacle not only for a fair health insurance, but also for one's career [NAC+14].

Genomic data holders sign contracts with people that provide their genomic information to them. One of the goals of these contracts is to ensure privacy of the provided data: the contracts allow an institution to analyze the data, but not to share it with other institutions. However, since many diseases are very rare, the need of collaboration between institutions arises to perform distributed analyses on aggregated genomic data in order to collect a sufficient amount of information for conducting more expressive statistical tests [SSDM09]. Because of the contractual obligation, the institutions cannot perform distributed analyses without a permission of the data owner.

In the past years, there were attempts by the research community to apply *Secure Multi-Party Computation (SMPC)* for guaranteeing privacy of data owners involved in distributed *Genome-Wide Association Studies (GWAS)*, e.g. [CTW+15]. This approach is very promising since it makes possible to share data on rare diseases and to potentially improve the curative treatment of those diseases. However, this approach is slow and of limited use because of the limitation in the number of participants involved in the protocol, which is 32 768 participants in [CTW+15], and low precision of the used 16-bit floating point arithmetic.

Other approaches use k-anonymity [Swe02], l-diversity [DMNS06] or differential privacy [MKGV07; RK17] to perform GWAS on distributed datasets. Nevertheless, several attacks were proposed immediately after that [GWA+12; SAW13; VSJO13]. These approaches seem also not to be suitable for GWAS because of their noise-based security, which reduces the utility of GWAS.

1.1 Motivation

Current approaches for conducting privacy-preserving GWAS have room for improvement in terms of performance. The ABY framework [DSZ15] appears to be well-suited for this task. It implements state-of-the-art and highly efficient protocols for *Secure Two-Party Computations (STPC)*. In particular, it uses efficient Boolean and Arithmetic sharing protocols as well as protocol conversion. These can be used to design algorithms for the statistical tests used in GWAS. ABY also supports STPC conversion during the execution, which can be used to improve the overall efficiency of algorithms by using certain protocols where they perform best.

Existing SMPC-based GWAS implementations also lack high precision and a possibility to include a large number of participants. These aspects can be improved by applying floating point arithmetic on data with increased bit-length, which has both high precision and a wide range of possible values for adequate bit-lengths, e.g. 2^{-126} to 2^{127} for 32-bit arithmetic.

To improve the efficiency of GWAS, one can use protocol conversion gates in ABY. However, since we use floating point arithmetic, which can be applied only in Boolean and Yao sharing, and our most efficient optimizations consider usage of Arithmetic sharing and then floating point arithmetic, ABY requires the conversion functionality from integer to floating point numbers inside of the SMPC protocol, which ABY unfortunately does not provide.

So far, only a very limited number of statistics was implemented for SMPC-based GWAS. This can be improved by introducing constructions for more advanced statistics that require more demanding operations that are not feasible in integer arithmetic, e.g. the G-test, which requires the calculation of the logarithm functionality.

The latest work for SMPC-based GWAS [CTW+15] uses counts for two alleles to construct the contingency table for the Chi-squared (χ^2)-test. This is inadequate because of the information loss caused by the dimension reduction and can be improved by extending the contingency table to use codewords instead of alleles.

1.2 Contributions

In this work, we design a large set of algorithms for the χ^2-, G- and P-test for the use in GWAS analysis and present optimizations for their implementation in the ABY framework [DSZ15]. In prior related work, Constable *et al.* [CTW+15] used only the χ^2-test and the Minor Allele Frequency (MAF) statistic in GWAS, a two-party LAN scenario, 2 columns in the contingency table, and performed only a very limited benchmarking. Their circuits operated only in unsigned integer 32-bit arithmetic and they simulated 16-bit floating point number arithmetic for the division operation in the χ^2-test.

Our contribution begins with the use of *Institute of Electrical and Electronics Engineers (IEEE)* 754 Floating Point numbers in the implemented algorithms, inter-protocol conversion for secure computations and the conversion of unsigned integer numbers to floating point numbers inside of the algorithm. Furthermore, we designed three protocols for each χ^2- and G-test, which are: (i) a straightforward algorithm implementation in Boolean sharing using only 32-bit floating point numbers, (ii) an optimization that performs input, addition and multiplication operations in Arithmetic sharing, and (iii) an optimization that uses Arithmetic sharing as well, but not for multiplications.

Performing the tests only on two allele counts, as it is done in [CTW+15], seems to be inappropriate for modern GWAS algorithms. That is why we construct additional algorithms for the χ^2- and G-test considering n codeword counts instead of two allele counts.

We consider an outsourcing scenario for GWAS, where n institutions holding their own genomic data collaborate in an SMPC protocol. Here, the institution I locally creates shares s_0 and s_1 of their genomic data and securely sends them to the corresponding non-colluding semi-trusted third parties T_0 and T_1. They collect the information from all of the institutions and perform GWAS statistics on the aggregated data. After that, they send the resulting shares to each institution which can (having both shares) locally compute the result from the received shares. Aggregating the data received from the institutions in Arithmetic sharing does not add any noticeable run-time costs to the SMPC protocol.

The P-test is implemented as a one-tailed threshold test using a precomputed value from the χ^2-distribution with respect to some p-value. In the one-tailed test, a result becomes significant by exceeding the given threshold. Here, we first compute the χ^2- or G-test and based on its results we compute the P-test.

Compared to [CTW+15], our implementation of the χ^2-test reduces the run-time by: (i) a factor 35.3x allowing $\sim$ 10x fewer participants in GWAS, and (ii) a factor 29.8x allowing millions of participants, which is by a few orders of magnitude larger.

1.3 Outline

This thesis has the following structure: in Chapter 2 we explain the used notation, give an overview over the basic concepts for SMPC and GWAS and detail the most important concepts. In Chapter 3 we discuss the latest scientific works in the areas of GWAS and SMPC. In Chapter 4 we give a description of the implementation details of the statistical test routines and the routines for outsourcing the computation to the semi-trusted third parties. In Chapter 5, we depict and describe the results of the benchmarks of the implemented algorithms. In Chapter 6 we conclude the thesis and discuss open research questions.

2 Preliminaries

In this chapter we introduce basic concepts and the notation for this thesis. We detail genome basics and the concept of how statistical tests can be used on the human genome to discover genetic-caused diseases. In addition, we describe the statistical tests used in this thesis. Next, we give information about SMPC and its building blocks. Afterwards, we describe the ABY framework for SMPC, the SMPC protocols that it supports, and the strategies how the efficiency of SMPC can be improved in ABY. Finally, we depict an outsourcing scenario where multiple parties outsource their secret shared inputs to semi-trusted third parties.

2.1 Notation

Most of the notation is taken from [DSZ15], where more details are given regarding the notation.

We write P_0 and P_1 for the parties among which secure computation is taking place and T_0, T_1 for the semi-trusted third parties that are used to outsource the computation.

The term $l[i]$ denotes a list operator that references an element i in list l. Alternatively, we write $l.e$ for the access operator that refers to element e in structure l, e.g. share $\langle s \rangle$ in a list l is $l.\langle s \rangle$.

We denote $\langle s \rangle_i^t$ as share s of party P_i in the sharing type t with $t \in \{A, B, Y\}$ corresponding to **A**rithmetic, **B**oolean and **Y**ao sharings (see Sections (2.6.1) and (2.6.2))). $\mathsf{Rec}_i^t(\langle x \rangle^t)$ denotes the reconstruction of share $\langle x \rangle$ in sharing t by party P_i or $\mathsf{Rec}_*^t(\langle x_0 \rangle^t, \langle x_1 \rangle^t)$ if the party already holds both shares. We write $\mathsf{Shr}_i^t(x)$ for secret-sharing value x by party P_i in sharing t.

We denote $x \oplus y$ and $x \wedge y$ as bit-wise *Exclusive OR (XOR)* and *AND* operations, respectively. We write $\langle z \rangle^t = \langle x \rangle^t \odot \langle y \rangle^t$ for operations on sharings, where $\odot \colon \langle x \rangle^t \times \langle y \rangle^t \mapsto \langle z \rangle^t$ with $t \in \{A, B, Y\}$.

The expression $\langle x \rangle^s = t2s(\langle x \rangle^t)$ denotes the conversion from t to s where $t \neq s$ and $t, s \in \{A, B, Y\}$, e.g. Y2B converts a Yao share to a Boolean share. This conversion implies that no additional information about the initial values is leaked in the protocol.

We write $\langle 0 \rangle$, $\langle 1 \rangle$ or $\langle \mathbf{F}(\cdot) \rangle$ for the secret-shared constant values which can also be a result of some function **F** that is computed locally and does not belong to the secret sharing protocol itself.

We denote (C-/R-)OT_l^n as n parallel (C-/R-)OTs on l-bit strings.

The *Multiplexer (MUX)*-gate is defined as $\mathbf{MUX}(a, b, s) = a$ if $(s = 0)$; b else, with the selection bit $s \in \{0, 1\}$.

2.2 Genomic Primer

Genetics is the study of inheritance and genetic variability. Genetic variability is the diversity of phenotypes in the population and the ability of the successors to develop different genotypes [RMG12].

Genetic information is stored in a *DNA* molecule. All known living organisms and some viruses contain DNA. It consists of a phosphate backbone and *nucleotides*. A nucleotide consists of one of the four nucleobases: *Adenine (A), Thymine (T), Cytosine (C)* or *Guanine (G)*. Nucleobases form base pairs according to the base pairing rules. The structure of DNA is shown in Figure 2.1[1]. DNA builds structures called *chromosomes*, where each chromosome is a DNA molecule containing all or a part of the genome of an organism. Human DNA consists of 23 pairs of chromosomes. These are 22 pairs of autosomes and a pair of sex chromosomes, which results in 46 chromosomes. The human genome consists of about 3 billion base pairs, however, genetic information of the human genome is more than 99% identical among individuals. That is why it is reasonable to perform genetic analyses only on *Single-Nucleotide Polymorphisms (SNPs)*. SNPs are variations of base pairs in the DNA sequence. They are inheritable or inherited variants.

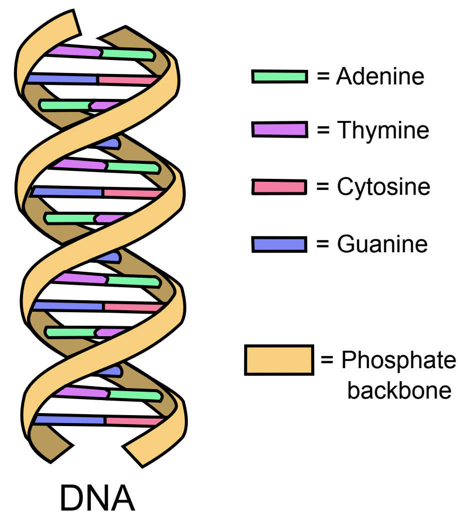


Figure 2.1: DNA sequence illustration.

The location of a gene in a chromosome is called *locus*. A genetic variant in a locus is called *allele*, which stands for different forms of a gene located in the certain locus of the homological chromosome, e.g. one person has the A allele and the other the T allele in the same locus. Alleles determine alternative developing variants of a trait. They can result in observable differences in traits, i.e. phenotypes, but usually have no observable effects.

Genetic techniques are an important milestone in finding associations between genetic variants and diseases for preventing genetic-caused diseases or when genetic properties have a boosting effect on a disease.

[1] Source: `https://upload.wikimedia.org/wikipedia/commons/f/fe/DNA_simple2.svg`

2.3 Genome-Wide Association Studies

GWAS are an approach of analyzing SNPs in order to find associations between diseases and genetic variants. For this purpose, the extent of linkage disequilibrium and the density of genetic markers should be adequate to make capturing the common variations in data possible.

The most commonly used strategy in GWAS is called *case-control group testing*. It is based on analyzing genetic data of the case and control groups, where the case group denotes a group of people affected by a disease and the control group denotes a group of healthy people. GWAS are performed by applying statistical tests on the data of case and control groups in order to find discriminative patterns that allow to distinguish one group from another.

The representation of SNP data for GWAS is shown in Table 2.1. The number of observations in each cell of the table is denoted as follows: a and c represent observations of the corresponding alleles 1 and 2 in the case group. Analogously, b and d correspond to the same alleles, but in the control group. In addition, we describe the total number of observations in alleles as $n_{A_1} = a + b$ and $n_{A_2} = c + d$ for alleles 1 and 2, as well as the number of observations in case and control groups as $n_{G_1} = a + c$ and $n_{G_2} = b + d$. The total number of observations for a SNP is denoted as $n = n_{A_1} + n_{A_2} = n_{G_1} + n_{G_2}$.

Table 2.1: SNP distribution table.

	Allele 1	Allele 2	Total
Case Group	a	c	n_{G_1}
Control Group	b	d	n_{G_2}
Total	n_{A_1}	n_{A_2}	n

2.4 Statistical Tests

Statistical tests can be used for accepting or rejecting the *null hypothesis* (H_0), which states that there is no statistically significant difference between two distributions with respect to some *significance level* α. The null hypothesis, in simpler words, states that two distributions are different due to a chance and not due to a regularity.

In the following, we describe three tests that are relevant for our work. These tests are applied to compare two distributions and test the significance of the results with respect to some α. The tests are: χ^2*-test*, *G-test*, and the *P-test* which uses results of one the first two tests. A result is statistically significant if it is very unlikely to obtain this result when H_0 holds true. The significance level is the probability of mistaking and, therefore, rejecting H_0 given that it is true.

Choosing $\alpha = 0.05$ is considered to be a good practice in statistics [Fis25], but in GWAS much smaller values are used for α, e.g. $5 \cdot 10^{-8}$ [BCGW12]. However, the method of significance testing was questioned by many researchers because of its unreliability in some cases and misuse to offer excuses for lower-quality scientific works [Car78; Ioa05]. Some journals even banned the publications that used significance testing in their research papers [Woo15].

Although the P-test on its own is considered to be a weak evidence of the result, it is still widely used as an additional method in combination with other testing techniques. This problem was controversially discussed in the scientific community. For example, Antonakis writes the following in [Ant17]:

> "Banning the reporting of p-values, as Basic and Applied Social Psychology recently did, is not going to solve the problem because it is merely treating a symptom of the problem. There is nothing wrong with hypothesis testing and p-values per se as long as authors, reviewers, and action editors use them correctly."

2.4.1 Chi-squared-test

The term χ^2-*test* refers to a statistical hypothesis test group which assumes that the sample distribution is the χ^2 distribution if H_0 holds true. The χ^2 distribution is a probability distribution. Its only parameter is the *degrees of freedom*, which is defined as $n = (columns - 1) \cdot (rows - 1)$, where *columns* is the number of columns and *rows* is the number of rows in the frequency table. We reject H_0 only if there is a statistically significant difference between the distribution of the sample and the distribution of the population. By rejecting H_0, we accept the alternative hypothesis H_1 - also known as H_a. The alternative hypothesis states that the distributions come from different populations.

In this work, we apply the χ^2-*test of independence* and the χ^2-*goodness-of-fit* test.

Test of Independence

The test of independence is used for comparing multiple nominal variables and evaluating whether the proportions of these variables significantly differ. The test is described in Equation (2.1):

$$\chi^2 = \sum_{i,j} \frac{(O_{i,j} - E_{i,j})^2}{E_{i,j}}, \tag{2.1}$$

where $O_{i,j}$ are the observed values of index i and $E_{i,j}$ are the expected values.

While the observed value $O_{i,j}$ equals the corresponding cell value in column i and row j, e.g. value a in the cell (1,1) in Table 2.1 for $O_{1,1}$, the expected value $E_{i,j}$ is computed using Equation (2.2):

$$E_{i,j} = \frac{n_{A_i} \cdot n_{G_j}}{n}, \tag{2.2}$$

where $E_{i,j}$ is the expected value in the cell (i, j), n_{A_i} is the number of observations in the column i and n_{G_j} is the number of observations in the row j.

In Equation (2.3), we provide an example for the calculation of $E_{1,1}$ in Table 2.1 to exemplify Equation (2.2):

$$E_{1,1} = \frac{n_{A_1} \cdot n_{G_1}}{n} = \frac{(a+b)\cdot(a+c)}{(a+b+c+d)} \tag{2.3}$$

Goodness-of-Fit Test

Goodness-of-fit indicates how good a statistical model can explain the dataset. In other words, it measures the deviation of an observed from the expected value. Its only difference from the test of independence is the calculation of the expected value, which equals the value in the corresponding cell. In GWAS, the expected value equals the number of observations in the corresponding cell in the control group. However, since the case and control groups do not necessarily contain the same number of observations, we weight the values in the contingency table by the number of observations in a group and then compare the proportions of the groups.

The numbers of weighted observations are hence computed as follows:

$$c'_{i,j} = \frac{c_{i,j}}{\sum_{k=1}^{n} c_{i,k}}, \tag{2.4}$$

where $c_{i,j}$ it the count of observations in the cell (i, j) and $c'_{i,j}$ is the weighted count of observations.

Computing the χ^2-value based on a p-value

The critical χ^2-test value is calculated from the inverted χ^2 *Cumulative Distribution Function (CDF)* given a p-value as described in Equation (2.5) [Mat17]:

$$\begin{aligned} x &= F^{-1}(p, \nu) = \{x : F(x, \nu) = p\} \\ p &= F(x, \nu) = \int_0^x \frac{t^{(\nu-1)/2} e^{-t/2}}{2^{\nu/2}\Gamma(\nu/2)} \mathrm{d}t, \end{aligned} \tag{2.5}$$

where x is the critical χ^2-value, $\Gamma(\cdot)$ is the Gamma function, p is the p-value and v is the degrees of freedom parameter.

There exist already available functions in high-level programming languages that allow one to compute the required critical value for the χ^2-distribution, which is a threshold value for the one-tailed χ^2-test stepping out of which the result of the statistic becomes significant. An example of such a function can be found in the Python library *SciPy*[1] as demonstrated in Listing 2.1.

```
1 alpha = 0.05
2 degrees_of_freedom = 1
3 critical_chi2 = scipy.stats.chi2.isf(alpha, degrees_of_freedom)
```

Listing 2.1: Critical value calculation for the χ^2-distribution using the Python library SciPy

Essentially, the χ^2-test is an approximation of the log-likelihood based statistics that was developed to avoid the calculation of logarithmic operations in log-likelihood statistics. On the one hand, this was due to unfeasibility of the logarithm calculation before computers and electronic calculators were massively developed and manufactured, whereas χ^2-tests can be performed manually by a human for small values. On the other hand, the χ^2-test is nowadays present in many textbooks [Hoe12]. Although the χ^2-test is easy to compute and it remains in many textbooks as one of the most standard statistics, it slowly gets replaced by the G-test [McD09].

2.4.2 G-test

The *G-test* is a likelihood-ratio or maximum likelihood statistical significance test. It has the following advantages over the χ^2-test: it can better deal with a small number of observations and its result is additive, thus, the test can be performed stepwise and after that summed for the complete statistic. The calculation of the G-test is shown in Equation (2.6):

$$g = 2\sum_{i,j} O_{i,j} \cdot \ln\left(\frac{O_{i,j}}{E_{i,j}}\right), \quad (2.6)$$

where $O_{i,j}$ is the observed value in the cell (i, j), $E_{i,j}$ is the expected value when H_0 is true and ln stands for the natural logarithm function.

Here, $O_{i,j}$ and $E_{i,j}$ are calculated as for the χ^2-test (see Section 2.4.1).

[1] https://www.scipy.org/

2.4.3 P-test

The *P-test* is a statistical test for proving that the result of a statistic exceeds a critical threshold that is based on a significance parameter α. This threshold is determined as a parameter and corresponds to the probability of rejecting H_0 given it is true, i.e. the significance level α. Hence, we reject H_0 if p-value $p < \alpha$. The calculation of the P-test is shown in Equation (2.7).

$$e_i = \begin{cases} 1 & \text{if } \mathbf{S}(X_i) > S_\alpha \\ 0 & \text{if } \mathbf{S}(X_i) \leq S_\alpha \end{cases}, \tag{2.7}$$

where e_i is an indicator flag for exceeding the critical threshold, $\mathbf{S}(X_i)$ is the result of the statistic $\mathbf{S} \in \{g, \chi^2\}$ for SNP i and S_α is the critical value for the statistic $\mathbf{S}$. We compute S_α using a lookup table for p-values (computed as described in Equation (2.5)) in the χ^2-distribution given a predefined α and reject H_0 if $\mathbf{S}(X_i) > S_\alpha$.

2.5 Secure Multi-Party Computation

The main technique that is used in this thesis for privacy-preserving computation of statistics on genome data is called Secure Multi-Party Computation (SMPC). The goal of SMPC is to jointly compute a function f on the inputs of multiple parties while keeping the inputs private.

For example: protocol participants $P_1, \ldots, P_n$ have their respective inputs $d_1, \ldots, d_n$ that correspond to their salaries. They want to know the highest salary among them, but without revealing their own salary to each other. They securely compute the function $f(d_1, \ldots, d_n) = \max(d_1, \ldots, d_n)$ and obtain the highest salary as an output. The biggest challenge of SMPC is to create a protocol that allows secure computation without relying on a trusted third party.

There exist different adversary models for proving the protocol security, e.g. passive (honest but curious), active (malicious), and covert:

- A passive adversary does not deviate from the protocol, but tries to learn as much as possible from the protocol execution itself. Parties can also collude to try to gain more information. The passive model yields only weak security regarding real-world threats, but constructing protocols assuming a passive adversary is often an important step for creating also maliciously secure protocols. In addition to that, they are usually very efficient.
- The active adversary model assumes that an attacker can arbitrarily deviate from the protocol in attempt to cheat. Protocols designed with active security are considered to be very secure for real-world applications.

- In the covert model, the adversary can arbitrarily deviate from the protocol, but he is caught with some probability, e.g. 20%. The deviation from the protocol can be attractive for this kind of adversary, but might be very costly when caught cheating.

2.5.1 Oblivious Transfer

OT is a cryptographic protocol where a sender holds two or more strings and a receiver obtains one of those strings based on its choice. The sender does not know which of the strings was chosen by the receiver and the receiver obtains only the chosen string and learns nothing about the other strings.

A common OT scheme is 1-out-of-2 OT, which was introduced by Rabin [Rab81]. We depict it in Figure 2.2. It is organized in the following way: the sender holds two messages m_0 and m_1, and the receiver holds a choice bit c. The receiver wants to obtain m_c without disclosing the value of c, while the sender wants the receiver to receive only m_c, but no information about m_{1-c}.

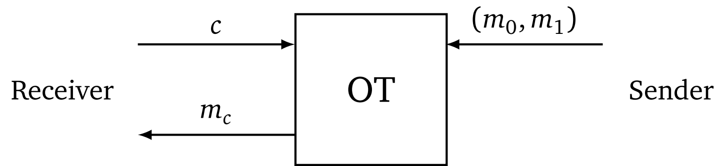


Figure 2.2: 1-out-of-2 Oblivious Transfer protocol.

There also exist other OT protocols, e.g. 1-out-of-n and k-out-of-n OT. In the 1-out-of-n OT protocol, the sender holds n messages $m_1, \dots, m_n$ and the receiver holds index i. The receiver wants to obtain knowledge about m_i and the sender wants to ensure that the receiver learns nothing new about other messages. In the k-out-of-n protocol, the sender holds a collection of n messages and the receiver wants to obtain k elements from this collection.

OT is an important building block for SMPC, namely for evaluating interactive operations securely, but it inherently lacks efficiency. Furthermore, it was shown that OT always requires public-key cryptography [IR89]. There exist many solutions that improve efficiency of OT protocols, e.g. *OT Extension*, *Random Oblivious Transfer (R-OT)* and *Correlated Oblivious Transfer (C-OT)* [ALSZ13]. Application of OT extension allows one to use symmetric cryptography to extend a few base OTs instead of using demanding public-key cryptography for each OT. In R-OT, the sender has no inputs and it obtains random m_0 and m_1 as an output, and the receiver obtains the message selected by a random choice bit. In C-OT, the sender has a correlation function $f_\Delta(\cdot)$ as input and it obtains m_0 and m_1, where m_0 is random and $m_1 = f_\Delta(m_0)$.

2.6 ABY Framework

ABY [DSZ15] is a state-of-the-art and highly efficient framework for generic SMPC. It is freely accessible on GitHub[2].

ABY supports three different SMPC protocols: **A**rithmetic (see Section 2.6.1), **B**oolean (see Section 2.6.2) and **Y**ao (see Section 2.6.2) circuits. For complex functions, the protocols can be interchanged inside of the computation process, i.e. a part of the function will be evaluated for example in Arithmetic and the other part in Yao sharing. However, the structure of the circuit must be determined before the execution. Since the protocols have different advantages and disadvantages and thus are better suited for different computations, they can be combined for many application scenarios to improve the overall efficiency.

Demmler *et al.* provide many code examples of how the framework can be efficiently applied, e.g. for complex cryptographic algorithms, such as *AES* and *SHA1*, biometric identification [DSZ15], and *Private Set Intersection (PSI)* [PSSZ15]. Through application of the mixed-protocol strategy, not only the computation, but also the communication overhead can be significantly reduced. While the inter-protocol conversion can result in better performance, it is hard to automate and optimize the overall design process especially considering different application scenarios and different network settings: *Wide Area Network (WAN)* and *Local Area Network (LAN)*. Since there are protocols that require a non-constant number of communication rounds, they are less efficient in the WAN setting because of the higher network latency compared to the LAN setting.

ABY supports IEEE 754 floating point arithmetic in Goldreich-Micali-Wigderson (GMW) and Yao protocols [DDK+15]. However, we construct our algorithms only considering GMW because of the correctness problems with floating point arithmetic in the Yao protocol. Floating point arithmetic is very costly in SMPC in terms of interactive operations, e.g. the addition operation in 64-bit and 32-bit floating point arithmetic in ABY requires 9x more AND-gates than in unsigned integer arithmetic. This drawback can be avoided by implementing integer to floating point conversion gates as we do in this thesis (see Section 4.1).

Among all operations in the circuits, we differentiate between interactive and non-interactive operations whereby the first term describes gates that require interaction between parties for the gate evaluation and the second term describes those that do not. As indicated by the name of the non-interactive gate, it can be evaluated locally without any communication between parties. For interactive gates, ABY uses multiplication triples (see Section 2.6.1). Since the benchmarking of DGK [DGK08], Paillier [DJN10] and OT-Extension [Bea96; IKNP03; ALSZ13] in [DSZ15] indicated that the OT-Extension based approach is the best in most cases, we use it in this work.

The evaluation of a protocol in ABY is divided into two phases: *setup* and *online*. Setup phase precomputes as much as possible, but independent of the inputs, and online phase does interactive computations depending on the inputs.

[2] `https://github.com/encryptogroup/ABY`

2.6.1 Arithmetic Sharing

In this section we describe the protocols for arithmetic sharing that were introduced in [ABL+04; PBS12; KSS14] and implemented in ABY [DSZ15]. We denote arithmetic sharing as additive sharing of an l-bit value x as the sum of two integer values in the ring $\mathbb{Z}_{2^l}$. More formally, $\langle x\rangle_0^A + \langle x\rangle_1^A \equiv x \mod 2^l$ with $\langle x\rangle_0^A, \langle x\rangle_1^A \in \mathbb{Z}_{2^l}$. The value is shared as follows: P_i chooses random value $r \in_R \mathbb{Z}_{2^l}$ and computes $\langle x\rangle_i^A = x - r$. After that, P_i sends r to P_{i-1} and P_{i-1} sets $\langle x\rangle_{i-1}^A = r$. The shared value can be reconstructed in the following way: P_{i-1} sends $\langle x\rangle_{i-1}^A$ to P_i and P_i sets $x = \langle x\rangle_0^A + \langle x\rangle_1^A$.

There are two types of operations in arithmetic sharing: addition and multiplication. While addition can be computed locally as $\langle z\rangle_i^A = \langle x\rangle_i^A + \langle y\rangle_i^A$ for each party P_i, multiplication must be evaluated interactively in the online phase. In the following, we describe how multiplication in arithmetic sharing can be computed using multiplication triples. P_i wants to compute $\langle z\rangle^A = \langle x\rangle^A \cdot \langle y\rangle^A$. For that, we require a multiplication triple $\langle c\rangle^A = \langle a\rangle^A \cdot \langle b\rangle^A$. First, P_i sets $\langle e\rangle_i^A = \langle x\rangle_i^A - \langle a\rangle_i^A$ and $\langle f\rangle_i^A = \langle y\rangle_i^A - \langle b\rangle_i^A$. After that, parties reconstruct values e and f and P_i computes $\langle z\rangle_i^A = i \cdot e \cdot f + f \cdot \langle a\rangle_i^A + e \cdot \langle b\rangle_i^A + \langle c\rangle_i^A$.

Multiplication triples

Multiplication triples are used to evaluate interactive operations, e.g. in the *GMW* protocol (see Section 2.6.1) and for additive sharing (see Section 2.6.2). Typically, multiplication triples are defined as $\langle a\rangle^t \odot \langle b\rangle^t = \langle c\rangle^t$, where $\odot \in \{\cdot, \wedge\}$ and $t \in \{A, B, Y\}$. Many strategies exist for the generation of multiplication triples, such as DGK [DGK08], Paillier [DJN10] and OT-Extension [Bea96; IKNP03; ALSZ13].

Here, we provide an example of generating a multiplication triple using OT in additive sharing [DSZ15]. We make use of OT-Extension to improve the efficiency of the generation routine. This protocol was proposed in [Gil99] and works as follows: we want to generate a multiplication triple $\langle c\rangle_i^A = \langle a\rangle_i^A \cdot \langle b\rangle_i^A$ for party P_i. We can write $\langle a\rangle^A \cdot \langle b\rangle^A = \left(\langle a\rangle_0^A + \langle a\rangle_1^A\right) \cdot \left(\langle b\rangle_0^A + \langle b\rangle_1^A\right) = \langle a\rangle_0^A \cdot \langle b\rangle_0^A + \langle a\rangle_0^A \cdot \langle b\rangle_1^A + \langle a\rangle_1^A \cdot \langle b\rangle_0^A + \langle a\rangle_1^A \cdot \langle b\rangle_1^A$. For that, P_0 generates random $\langle a\rangle_0^A, \langle b\rangle_0^A \in_R \mathbb{Z}_{2^l}$ and P_1 generates random $\langle a\rangle_1^A, \langle b\rangle_1^A \in_R \mathbb{Z}_{2^l}$. Knowing $\langle a\rangle_0^A \cdot \langle b\rangle_1^A$ in plaintext would leak the information about the shared value. This is why we compute it as $\langle u\rangle^A = \langle\langle a\rangle_0^A \cdot \langle b\rangle_1^A\rangle^A$, and the same for $\langle a\rangle_1^A \cdot \langle b\rangle_0^A$. The share $\langle u\rangle^A$ is computed securely, i.e. it is shared between two parties as $\langle u\rangle_0^A$ and $\langle u\rangle_1^A$.

The parties perform a C-OT_l^l, where P_0 acts as the sender and P_1 acts as the receiver. P_1 uses $\langle b\rangle_1^A[i]$ as the choice bit for each bit in $\langle b\rangle_1^A$ and P_0 uses the correlation function $f_{\Delta_i}(x) = \left(\langle a\rangle_0^A \cdot 2^i - x\right) \mod 2^l$ as the input. P_0 obtains $\left(s_{i,0}, s_{i,1}\right)$ where $s_{i,0} \in_R \mathbb{Z}_{2^l}$ and $s_{i,1} = f_{\Delta_i}\left(s_{i,0}\right) = \left(\langle a\rangle_0^A \cdot 2^i - s_{i,0}\right)$ and P_1 obtains $s_{i,\langle b\rangle_1^A[i]} = \left(\langle b\rangle_1^A[i] \cdot \langle a\rangle_0^A \cdot 2^i - s_{i,0}\right) \mod 2^l$. After that, P_0 and P_1 compute their corresponding $\langle u\rangle_0^A = \left(\sum_{i=1}^l s_{i,0}\right)$ and $\langle u\rangle_1^A = \left(\sum_{i=1}^l s_{i,\langle b\rangle_1^A[i]}\right)$. Similarly, the parties compute $\langle v\rangle^A = \langle\langle a\rangle_1^A \cdot \langle b\rangle_0^A\rangle^A$. After all precomputations are done, the party P_i sets $\langle c\rangle_i^A = \langle a\rangle_i^A \cdot \langle b\rangle_i^A + \langle u\rangle_i^A + \langle v\rangle_i^A$.

2.6.2 Boolean Sharing

A *Boolean Circuit (BC)* is a directed acyclic graph consisting of boolean gates which are based on the boolean logic. Usually, there are *AND*, *XOR* and *NOT* gates and all other gates are constructed from those primitive gates. More complex combined gates can perform addition, multiplication, and even more demanding operations [Vol13]. The complexity of the circuit is typically measured in the number of gates and/or the depth of the circuit, but for specific implementations other measures can be used, e.g. the number of AND-gates in SMPC because of their high cost compared to free XOR gates. The last is due to the interactiveness that is required for evaluating an AND-gate.

In Boolean sharing, the function to be computed is converted to a BC and then evaluated securely. ABY supports two types of Boolean sharing: GMW and Yao. Although both protocols operate on BCs, we reference GMW as Boolean and Yao as Yao sharing as it was done in [DSZ15].

GMW

The protocol of *Goldreich-Micali-Wigderson (GMW)* is an interactive SMPC protocol that was first introduced in [GMW87]. In GMW, two parties P_0 and P_1 interactively compute a function f, represented as a BC. Each bit in the circuit corresponds to a so-called wire and each wire is shared among the participants using a 2-out-of-2 secret sharing scheme meaning that both parties are required to reconstruct the secret. The value v of a wire is represented as $v = v_1 \oplus v_2$, where v_1 and v_2 are the shares of the parties. The shares are random-looking and thus do not leak any information about the value v. Since XOR is an associative operation, XOR gates can be evaluated non-interactively. However, AND gates require interaction between the parties.

For the evaluation of an interactive AND gate, oblivious transfer or multiplication triples can be used. OT can be used as follows: for the secure evaluation of an AND gate on input shares x_1, x_2 and y_1, y_2, the parties can run a 1-out-of-4 OT protocol. P_1 inputs x_1 and y_1 as input, P_2 chooses a random output share z_2 and four inputs. At the end of the OT protocol, P_1 receives $z_1 = z_2 \oplus ((x_1 \oplus x_2) \wedge (y_1 \oplus y_2))$. Multiplication triples, on the other hand, can be used as follows: a_i, b_i, c_i are generated in the setup phase of the protocol such that $(c_1 \oplus c_2) = (a_1 \oplus a_2) \wedge (b_1 \oplus b_2)$. In the online phase, the parties mask their inputs x_i and y_i and exchange $d_i = x_i \oplus a_i$ and $e_i = y_i \oplus b_i$. After that, the parties reassemble output shares $z_1 = (d \wedge e) \oplus (b_1 \wedge d) \oplus (a_1 \wedge e) \oplus c_1$ and $z_2 = (b_2 \wedge d) \oplus (a_2 \wedge e) \oplus c_2$. The usage of multiplication triples generates less overhead as per multiplication triple only a single message with a smaller size must be sent during the protocol execution.

Yao

Yao's Garbled Circuits (GCs) is a secure two-party computation protocol introduced by Yao [Yao86]. It uses BCs for the computation of an arbitrary function f, where there are two parties P_0 and P_1 that want to compute f on their corresponding inputs d_0 and d_1 without revealing inputs to each other.

In the setup phase of the protocol, the parties locally construct a BC for the function f. P_0 acts as a *garbler*, i.e. it encrypts the circuit and its input d_0 and sends it to P_1 who acts as an *evaluator*, i.e. evaluates the received circuit. To obtain its encrypted input d_1, P_1 uses OTs. By evaluating the circuit, P_1 obtains the encrypted output. Finally, P_1 decrypts the result.

For creating a garbled circuit out of the initial BC, P_0 creates two *labels* of bit-length k for each wire in the circuit, where k is the security parameter. Labels are random bit strings that replace 0 and 1 values on the wires in the truth tables of the gates.

In Yao's garbled circuit protocol, unlike in GMW, there is a constant number of communication rounds in the online phase, where only symmetric cryptography is required. Thus, the evaluation of all gates becomes non-interactive. Various strategies were proposed in the literature to improve the speed of the protocol: point-and-permute [MNP+04], free XOR [KS08], fixed-key AES garbling [BHKR13], and half-gates [ZRE15]. These techniques can be combined efficiently to allow free non-cryptographic and non-interactive evaluation of XOR gates in the protocol for both, setup and online phases.

2.6.3 SIMD Gates

ABY implements the *Single Instruction Multiple Data (SIMD)* approach. Bogdanov *et al.* show in [BJL12] that the usage of SIMD gates can result in significant efficiency improvements. The idea of SIMD is to evaluate one sub-circuit on an n-bit value instead of evaluating n identical sub-circuits on 1-bit values. Since for SIMD gates we generate a circuit only once, the routine requires much less Random Access Memory (RAM).

In [SZ13], Schneider and Zohner benchmarked GMW in the two-party setting. They compute a huge benchmarking circuit consisting of millions of gates using two gate configurations: SIMD and non-SIMD gates. Their results are the following: the usage of SIMD gates in GMW slightly decreases setup time and significantly decreases online time, namely by a factor 7.5x. Memory demands of the circuit were reduced even more significant, i.e. by a factor $\sim$15x. It is worth to mention that the overhead that SIMD gates cause due to conversion between bitwise and SIMD operations is negligible.

2.6.4 Conversion Gates

The concepts of inter-protocol conversion for SMPC was proposed by many researchers, e.g. [BPSW07; BFK+09; KSS13]. However, these concepts lack real improvements that outperform single protocol performance. This is due to the expensive conversion between garbled circuits and homomorphic encryption which was used mostly.

ABY implements efficient protocols for the protocol conversion which we detail below [DSZ15].

Yao to Boolean Sharing (Y2B)

Conversion from $\langle x \rangle^Y$ to $\langle x \rangle^B$ is the most natural one since it can be computed locally. The permutation bits [MNP+04] in the Yao shares already contain a valid Boolean sharing of x. To convert the value $\langle x \rangle_i^Y$, P_i locally computes $\langle x \rangle_i^B$.

Boolean to Yao Sharing (B2Y)

Conversion from Boolean to Yao sharing corresponds to sharing a one-bit value in Yao. For performing a conversion on an n-bit value, n parallel operations are performed for each bit of that value.

Yao to Arithmetic Sharing (Y2A)

To convert a Yao share to an Arithmetic share, P_0 generates a random $r \in_R \mathbb{Z}_{2^l}$ and shares it in Yao sharing. Afterwards, both of the parties evaluate a subtraction circuit $\langle d \rangle^Y = \langle x \rangle^Y - \langle r \rangle^Y$ and set $\langle x \rangle_0^A = r$ and $\langle x \rangle_1^A = \mathsf{Rec}_1^Y\big(\langle d \rangle^Y\big)$.

Demmler *et al.* [DSZ15] consider it to be more efficient to locally compute $Y2B$ and after that the cheaper $B2A$ which results in $\langle x \rangle^A = Y2A\big(\langle x \rangle^Y\big) = B2A\big(Y2B\big(\langle x \rangle^Y\big)\big)$. The $B2A$ conversion is explained below.

Arithmetic to Yao Sharing (A2Y)

To convert an Arithmetic share into a Yao share, as proposed in [HSS+10], we can evaluate an addition circuit. P_i computes $x_i = \langle x \rangle_i^A$ and sets $\langle x_i \rangle^Y = \mathsf{Shr}_i^Y(x_i)$. After that, the parties can compute $\langle x \rangle^Y = \langle x_0 \rangle^Y + \langle x_1 \rangle^Y$.

Arithmetic to Boolean Sharing (A2B)

There are two approaches proposed in ABY to perform Arithmetic to Boolean share conversion. The first and less efficient one evaluates a bit-extraction circuit as proposed in [CD10]. The second one, similar to A2Y, makes use of the equation $\langle x \rangle^B = A2B\big(\langle x \rangle^A\big) = Y2B\big(A2Y\big(\langle x \rangle^A\big)\big)$. This results in better performance since $Y2B$ is "for free" and an addition circuit in Yao is efficient as confirmed by the benchmarking results in [DSZ15].

Boolean to Arithmetic Sharing (B2A)

Two solutions are provided in [DSZ15] for Boolean to Arithmetic share conversion. The first solution is to evaluate a Boolean addition circuit. Here, P_0 has $\langle x \rangle_0^B$ and random $r \in_R \{0,1\}^l$ and then sets $\langle x \rangle_0^A = r$. P_1 has $\langle x \rangle_1^B$ as input and computes $\langle x \rangle_1^A = x - r$. This approach would be inefficient because of its either big circuit size or big depth.

The second solution is similar to the generation of multiplication triples for Arithmetic sharing. In this approach, the parties perform an OT for each bit in the value. The sender transfers two additively correlated values. The receiver obtains only one of the values. The sum of the values forms a valid Arithmetic share. This solution results in a performance improvement compared to the first solution.

2.7 Outsourcing Computation

There exist functions in ABY for outsourcing computation to semi-trusted third parties, i.e., the parties do not receive any plaintext values, but are not supposed to collude. The goal of outsourcing could be for example moving the SMPC execution from WAN to LAN. This is due to efficiency losses for SMPC in the WAN setting compared to the LAN setting. Furthermore, using this approach, $n > 2$ parties can efficiently use secure computation protocols constructed for 2 parties. The outsourcing functions were constructed for and applied in [CDC+16].

The execution of the protocols in the LAN setting would be a hard task for the medical institutions that want to share their genome data, because they will very unlikely be near to each other. By considering a scenario with few or dozens of institutions, the probability of them being in the same LAN is extremely low. The protocol, however, can be applied in the LAN setting by outsourcing the computations to semi-trusted third parties.

The outsourcing protocol works as shown in Figure 2.3: each party P that wants to compute the statistic **S** by outsourcing the computations, secret-shares its value s locally, thus knowing both $\langle s \rangle_0^t$ and $\langle s \rangle_1^t$. Afterwards, it sends the share $\langle s \rangle_j^t$ to the server T_j. The non-colluding servers receive shares from all of the parties and compute the statistic **S** interactively on the aggregate share $\langle a \rangle^t$. The main SMPC protocol is performed between T_0 and T_1 which send their

output shares to each party P after the computation is done. Each party receives both shares $\langle r\rangle_0^t$ from T_0 and $\langle r\rangle_1^t$ from T_1 and locally computes $\mathsf{Rec}_*^t(\langle r\rangle_0^t, \langle r\rangle_1^t)$.

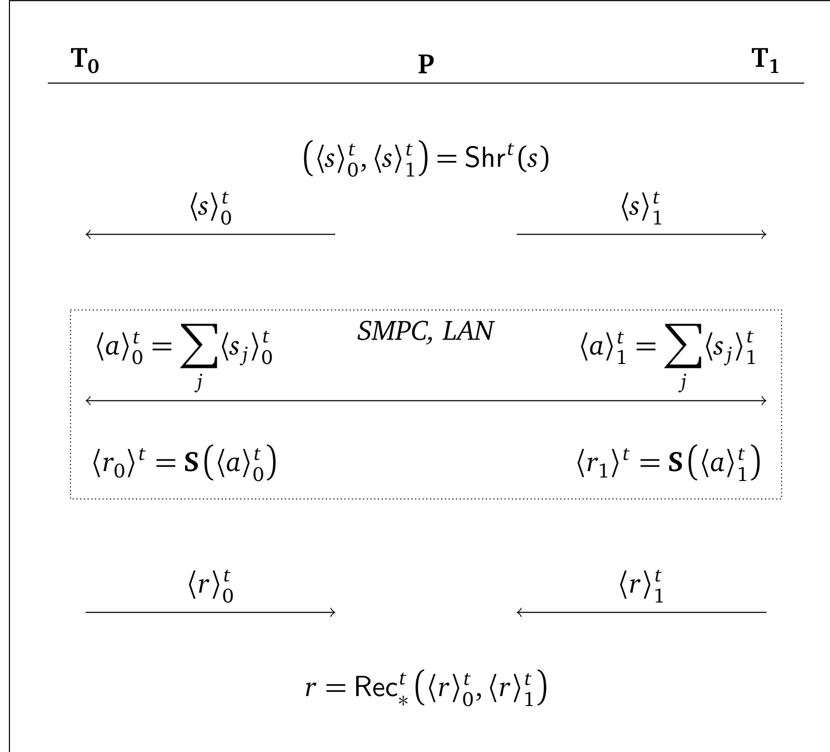


Figure 2.3: Outsourcing computation scheme for computing a statistic **S** on aggregate data received from multiple parties.

3 Related Work

In this chapter, we describe different existing approaches for computation of privacy-preserving *Genome-Wide Association Studies (GWAS)*.

Homer *et al.* [HSR+08] introduced an attack showing that the participation of an individual in GWAS can be revealed from the aggregate data. After that, many attacks were proposed to deanonymize individuals in distributed GWAS [WLW+09; ZPL+11]. The attacks started from demands of extremely large databases for the de-anonymization and moved to more realistic sizes, and finally, a practical attack was constructed in [CHW+15].

Considering these attacks, the research community started to move towards privacy-preserving solutions for distributed GWAS, which are nowadays divided into three distinct fields based on the underlying techniques that guarantee privacy:

- Noise-based
- Based on *Homomorphic Encryption (HE)*
- Based on *Secure Multi-Party Computation (SMPC)*, e.g. *Yao's Garbled Circuits (GCs)*
- Other techniques

3.1 GWAS using Noise-based Approaches

Noise-based is the most popular approach for ensuring privacy of individuals in privacy-preserving GWAS. Many solutions were proposed in the last decade for applying differential privacy to privacy-preserving GWAS, e.g. [FSU11; USF13; JS13; YFSU14; YJ14; JZW+14; ZWJ+14; SB16; RK17].

Unfortunately, these methods have drawbacks. Adding noise to the data generally reduces its utility and can influence results. Furthermore, researchers will have difficulties convincing approving authorities of this method.

3.2 GWAS using Homomorphic Encryption

HE is a form of public-key cryptography that allows to perform computations on ciphertexts and get correct results after decryption. The operations are performed without disclosing encrypted data to the party performing the calculation. The ciphertext can be decrypted only by the party possessing the private key [Rap04].

At the *Integrating Data for Analysis, Anonymization and SHaring (iDASH)* research competition in 2015[1], several attempts to apply HE for distributed GWAS were presented, which we describe below.

Zhang et al. [ZDJ+15] used the FORESEE framework to fully outsource GWAS into the cloud. The main advantage of their framework is that it is able to compute the division operation unlike many other frameworks. They state their best run-times of about 52 ms for a single SNP in the χ^2-test.

In [LYS15], Lu *et al.* performed GWAS in the cloud. They use a packing technique for the frequency table to improve the efficiency of the routine. They state 35 ms run-time for a χ^2-test evaluation on a single SNP with 10 000 observations.

Kim and Lauter [KL15] apply the BGV [GHS12] and YASHE schemes [BLLN13] for fully homomorphic encryption to compute the χ^2-test. The YASHE-based solution resulted in better efficiency, namely in 5 ms amortized run-time for a SNP.

3.3 GWAS using Yao's Garbled Circuits

To the best of our knowledge, the latest work that performs GWAS using secure circuit evaluation is presented by Constable *et al.* in [CTW+15]. The statistical tests they use are the χ^2-test and the MAF, which yields the most infrequent allele in a SNP. They perform GWAS in the LAN setting on two servers in the local network of Syracuse University. Their servers have 6 GB RAM and Intel Core i5 750 processors with 4 cores which is quite similar to our benchmarking environment (cf. Section 5.1).

In their work, they simulate floating point arithmetic using 16-bit half-precision numbers. This is an important constraint since there are multiplications in the χ^2-test that can lead to an overflow or to precision interference. Beyond that, they provide very few information on the benchmarking results of their algorithms which makes it hard to compare run-times of the algorithms presented in this thesis and the algorithms presented in [CTW+15]. Constable *et al.* provide only two values for run times of the χ^2-test. These are 47.27 seconds for 311 SNPs and 1 342 seconds for 9 330 SNPs, i.e. 143 ms for a single SNP.

[1] http://www.humangenomeprivacy.org/2015/index.html

They employ the PCF circuit framework [KSMB13] for performing SMPC. SNPs are preprocessed locally using Python in exactly the same ways as in this thesis, before the execution of the routine starts.

For the preprocessing, they parse the required number of SNPs and count how many times alleles occurred in the observations. For example, the base pair AA will be parsed as two occurrences of the allele A, whereas AT and TA are both counted as one allele A and one T. However, by ignoring the positions of the alleles, they ignore an important information in genes.

For the communication evaluation, they unfortunately provide only a figure and no exact values. Hence, it is hard to determine exact communication demands of their protocol implementation. An execution of the protocol with 311 SNPs requires roughly a few gigabytes of communication and with 9 330 SNPs roughly 50-100 GB.

In this thesis, we improve the work of Constable *et al.* [CTW+15] and use it to compare with the performance of our mixed-protocol algorithms. Additionally, we extend the number of alleles in the contingency table. Finally, we consider an outsourcing scenario where multiple institutions perform SMPC on aggregate data by outsourcing the computation to semi-trusted third parties. We reach 4 ms amortized run-time per SNP for our most efficient χ^2-test algorithm which outperforms [CTW+15] by a factor 35.3x, and 4.8 ms for the algorithm that allows millions of participants and by a factor 29.8x faster than [CTW+15].

3.4 GWAS using Other Techniques

Zhang *et al.* [ZBA15] propose a solution using Shamir's secret sharing scheme with (n, t)-threshold where n is the number of protocol participants, $t+1$ shares are required to reconstruct the secret, $t=1$, and $n=3$, thus assuming non-colluding parties. They bring up a promising protocol for the privacy-preserving χ^2-test stating the amortized run-time of 4.5 ms for a SNP.

In [CDD+16], Chen *et al.* proposed to compute several GWAS statistics in a secure enclave using *Intel Software Guard Extensions (SGX)* [BBB+16].

4 Implementation

In this chapter we describe the implementation details of the statistical tests applied to the genetic data as well as the implementation of the required underlying techniques.

The algorithms that we implement are the following:

Preprocessing The raw data (sequences of alleles, e.g. "SNP #1: $\overbrace{\text{AA TA} \ldots \text{TT}}^{\text{observations}}$") must be processed and converted to a suitable format, i.e. the counts of alleles. For this purpose, we compose a very simple Python script which we do not describe further. We operate on data from the iDASH competition 2015[1] and duplicate it to produce the required number of observations and SNPs.

Integer to floating point number conversion gates These are the prerequisite for most of our algorithms. The goal of the conversion gates is to make it possible to perform the most calculations in unsigned integer arithmetic in additive sharing, which is very efficient, and then to perform a conversion to floating point numbers where all further computations are performed.

Statistical tests We implements 4 versions of χ^2- and G-test which differ in the way they are mathematically computed, the SMPC protocols in which they are computed and the number of alleles/codewords used in the contingency table. Additionally, we implement the P-test, which is generic and can be applied after each of the aforementioned tests in order to evaluate statistical significance of the result of the statistic.

Outsourcing computation As the last step, we implement a scenario where the institutions holding their genomic data send their secret-shared data to two semi-trusted third parties, which aggregate the received data and perform the computation of the statistical test in the SMPC protocol and send the resulting shares to each institution. The institutions then reconstruct the result locally.

4.1 Integer to Floating Point Number Conversion

Multiple algorithms were proposed for computing floating point arithmetic in SMPC [HSS+10; ABZS13; ZSB13; KW14; DDK+15; PS15] and in HE [LDD+16]. The corresponding algorithms

[1] http://www.humangenomeprivacy.org/2015/index.html

were implemented either in Sharemind [BLW08] and PICCO [ZSB13], or as standalone applications.

For the conversion of integer to floating point numbers we use the state-of-the-art algorithm introduced by Aliasgari *et al.* in [ABZS13] and adapt it to ABY. The algorithm is implemented in a simplified fashion and optimized for our purposes. We describe different optimization steps (labeled $O_0 \dots O_4$) in the following.

The initial version of the algorithm is described in Protocol 4.1, where additive secret-sharing of value v is denoted as $[v]$ and vectors containing values v_1, v_2 as $\langle v_1, v_2 \rangle$. Here, Aliasgari *et al.* use the integer number $[a]$, integer bit-length γ and floating-point number representation bit-length l as input arguments. The function returns a vector containing the floating point number $[v]$, exponent $[p]$, indicator of $[a]$ being 0 $[z]$ and indicator of being less than 0 $[s]$.

In the protocol, they first assign the length of γ cutting out the sign bit of the signed integer to λ. Next, they determine whether a is less than 0 using the according function. Then, they assign 1 to z if a equals 0 and 1 otherwise. The value of a is inverted if it was negative before. After that, a is split into bits using the *bit decoder* function. In Line 6 of Protocol 4.1, the *prefix-OR* function is applied to the decomposed value of a, where it computes $b_i = \bigvee_{j=1}^{i} a_j$. Then, the value of v is calculated by shifting the bit-representation of a by k bits, where $k = y_i + 1$. This gives us $a2^k$, hence, the output exponent needs to be set to $-k$, which happens in Line 8. In Lines 10 and 11, the value is truncated if it is too long for the representation or it is shifted if it is too short. At the end, the number of shifted bits in Line 10 is added to the exponent p and it is reset if the value equals 0.

$\langle [v], [p], [z], [s] \rangle \leftarrow$ **INT2FP**$([a], \gamma, l)$

1 : $\lambda \leftarrow \gamma - 1$
2 : $[s] \leftarrow$ **LTZ**$([a], \gamma)$
3 : $[z] \leftarrow$ **EQZ**$([a], \gamma)$
4 : $[a] \leftarrow (1 - 2[s])[a]$
5 : $[a_{\gamma-1}], \dots, [a_0] \leftarrow$ **BitDec**$([a], \lambda, \lambda)$
6 : $[b_0], \dots, [b_{\lambda-1}] \leftarrow$ **PreOR**$([a_{\lambda-1}], \dots, [a_0])$
7 : $[v] \leftarrow [a](1 + \sum_{i=0}^{\lambda-1} 2^i (1 - [b_i]))$
8 : $[p] \leftarrow -(\lambda - \sum_{i=0}^{\lambda-1} [b_i])$
9 : **if** $(\gamma - 1) > l$ **then** $[v] \leftarrow$ **Trunc**$([v], \gamma - 1, \gamma - l - 1)$
10 : **else** $[v] \leftarrow 2^{l-\gamma+1}[v]$
11 : $[p] \leftarrow ([p] + \gamma - 1 - l)(1 - [z])$
12 : **return** $\langle [v], [p], [z], [s] \rangle$

Protocol 4.1: Integer to floating point number conversion algorithm from [ABZS13].

For our goals, we can immediately exclude the following parts of the protocol:

- All lines where negative values are handled (Lines 2 and 4), because in the ABY framework only unsigned integer numbers are used.
- Bit decomposition, because in the framework all values in binary circuits are decomposed by default (Line 5).
- Bit shifting/truncation (Lines 9 and 10). Since we require a limited amount of observations to perform expressive statistical analyses, and the number of possible observations is naturally limited, we assume that the integer numbers will fit exactly into the representation. This reduces the computation overhead.

As an effect of these exclusions, we can also ignore the bit shifting in Line 11, because we fit exactly into the representation of the fraction. Moreover, the subtraction $(1-[z])$ in Line 11 can be simplified for BCs by replacing it with the XOR operation. Therefore, we simplify it to $(1 \oplus [z])$ as $z \in \{0,1\}$. Since the exponentiation in Line 7 is very costly, we replace it with MUX gates of precomputed constant values of form 2^i and use the inverted value b as the selection bit. In addition to the aforementioned simplifications, we provide the optimizations described below where O_{i+1} contains O_i.

Optimization 1 (O_1): In Line 7, we still need λ additions. This implies communication overhead because of the large number of AND-gates that are needed for addition in boolean circuits. To optimize this step, we can simply invert b and reverse its bit order. This gives us the correct value for padding $[a]$ to the size of the significand. In ABY, bit-reversing needs no gates and an inversion gate can be evaluated without cryptographic computation.

Optimization 2 (O_2): In Line 11, there is a need for multiplication, which is extremely costly in BCs. In the original simplified version of the protocol, we compute $[p] \leftarrow [p] \cdot ([1] \oplus [z])$. This multiplication can be avoided by choosing either p or constant 0 by using a MUX gate with selection bit z. Hence, we compute $[p] \leftarrow \textbf{MUX}([p],[0],[z])$, where the MUX operation yields $[p]$ if $([z]=[0])$, and $[0]$ otherwise.

Optimization 3 (O_3): In Line 7, we perform an addition of 1 to (2^k-1) in the original protocol. We can avoid this addition by using a chain of XOR gates, where we apply the XOR operation to bits a_i and a_{i-1}. As a result, we obtain a bit b_i, which indicates whether the input variables contained different values. Since we know that the bit representation of the value (2^k-1) is a string of 1-bits followed by 0-bits, we want to indicate the bit next after the last 1-bit, which represents the value 2^k. The applied scheme is graphically represented in Figure 4.1 for an 8-bit integer value.

Optimization 4 (O_4): In Line 7, we need to multiply a by 2^k which, in BCs, can be represented as a bit-shift of k-bits. Since there is no bit-shifter in the standard functionality of the ABY framework, we implemented the barrel shifter algorithm introduced in [PSW02]. Using a barrel shifter instead of multiplication, we significantly reduce the total number of gates as well as the number of AND-gates which are considered to be most costly. A graphical

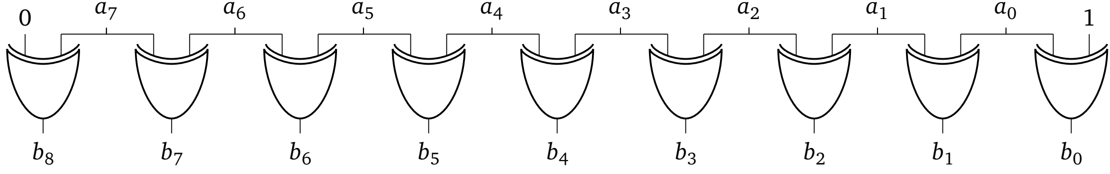


Figure 4.1: XOR-chain for computing $(2^k - 1) \mapsto 2^k$.

representation of an 8-bit barrel shifter is shown in Figure 4.2. Here, to shift the bits $a_7, \ldots, a_0$ of the value a we use the selection bits $s_2, \ldots, s_0$. By setting $s_0 = 1$ and other bits to 0, we shift a by one bit, and by setting also $s_2 = 1$, we shift the value by 5 bits.

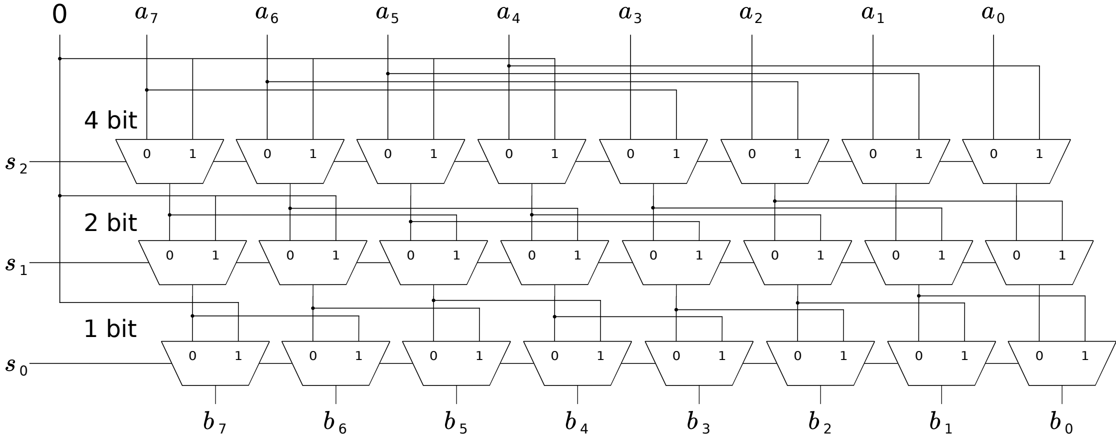


Figure 4.2: 8-bit logical barrel right shifter [PSW02].

The final version of the algorithm is described in Protocol 4.2. The function **INT2FP** takes a share $\langle a \rangle^B$, the value bit-length γ and the desired representation length l as arguments and it returns a floating point number $\langle f \rangle^B$ as share. In the first step, we assign the return value of the equality test **EQZ** based on the share $\langle a \rangle^B$ to the share $\langle z \rangle^B$. Afterwards, the prefix-OR function **PreOR** is calculated using the input share $\langle a \rangle^B$ and its return value is assigned to $\langle p \rangle^B$. After that, we invert $\langle b \rangle^B$ bit-wise and calculate the Hamming weight [BP08] on $\langle b \rangle^B$ which we then assign to $\langle v \rangle^B$, where the Hamming weight is the number of 1-bits in a bit string. As explained before, this value is required to shift $\langle a \rangle^B$ in order to fit the size of the significand. In Line 6, the barrel right shifter function is applied on $\langle a \rangle^B$ to shift it by the value of $\langle v \rangle^B$. We resize the bit-length of the fraction to form the correct IEEE 754 [IEE08] representation. The fraction size is computed using a lookup table. We truncate the most significant bits of the fraction if it is too long or pad it with constantly shared zero bits if it is too short. In Line 8, we calculate the Hamming weight [BP08] of the share $\langle p \rangle^B$ and assign it to the exponent share $\langle e \rangle^B$ after which we add a constantly shared bias for the representation length l using the lookup table. In Line 10, we check if $\langle a \rangle^B$ equals $\langle 0 \rangle^B$ and if it is the case, we assign $\langle 0 \rangle^B$ to $\langle e \rangle^B$. To form the final representation of the floating point number, we concatenate the fraction share $\langle v \rangle^B$, the exponent share $\langle e \rangle^B$ and

a constant $\langle 0 \rangle^B$. The last one is constant due to the fact that we operate only on unsigned integer values and the most significant bit in floating point arithmetic being 0 means that the number is positive. After that we return the computed share $\langle f \rangle^B$. To convert integer values that are shared in Arithmetic sharing to floating point numbers, we have to invoke the function **INT2FP**($\langle a \rangle^A, \gamma, l$), which solely performs a conversion $\langle a \rangle^B \leftarrow$ **A2B**($\langle a \rangle^A$) before the main routine.

$\langle f \rangle^B \leftarrow$ **INT2FP**($\langle a \rangle^B, \gamma, l$)

1 : $\langle z \rangle^B \leftarrow$ **EQZ**($\langle a \rangle^B$)
2 : $\langle p \rangle^B \leftarrow$ **PreOR**($\langle a \rangle^B$)
3 : $\langle b \rangle^B \leftarrow$ **Reverse**($\langle p \rangle^B$)
4 : $\langle b \rangle^B \leftarrow$ **Invert**($\langle b \rangle^B$)
5 : $\langle v \rangle^B \leftarrow$ **HammingWeight**($\langle b \rangle^B$)
6 : $\langle v \rangle^B \leftarrow$ **RShifter**($\langle a \rangle^B, \langle v \rangle^B$)
7 : $\langle v \rangle^B \leftarrow$ **Resize**($\langle v \rangle^B$, **FractionSize**(l))
8 : $\langle e \rangle^B \leftarrow$ **HammingWeight**($\langle p \rangle^B$)
9 : $\langle e \rangle^B \leftarrow \langle e \rangle^B + \langle$**FPBias**($l$)$\rangle^B$
10 : $\langle e \rangle^B \leftarrow$ **MUX**($\langle e \rangle^B, \langle 0 \rangle^B, \langle z \rangle^B$)
11 : $\langle f \rangle^B \leftarrow$ **Concatenate**($\langle v \rangle^B, \langle e \rangle^B, \langle 0 \rangle^B$)
12 : **return** $\langle f \rangle^B$

Protocol 4.2: Implementation of the fully optimized integer to floating point number conversion algorithm.

The effects of the improvements are shown in Table 4.1 and visualized in Figure 4.3. It is obvious that every improvement brought more efficiency benefits in the number of gates, but the last improvement is the most important one. In both, Table 4.1 and Figure 4.3, the last 2 columns denote an addition of 64-bit floating point (FP+) and unsigned integer numbers (UINT+) and are illustrated here for the purpose of comparison. Again, the notation O_i stands for the optimization number i of the 64-bit conversion. Figure 4.3 contains statistics not only for the amount of AND- and XOR-gates, but also for the amount of MUX-gates, which are optimized in ABY [DSZ15].

Altogether, the improvements reduce the total number of gates and the number of AND-gates by factor ∼7x. Compared to the 64-bit floating point number addition, a conversion operation requires a similar total number of gates, but ∼6 times fewer AND-gates. This can be interpreted as a clear indicator of the benefit of using integer operations whenever possible in BCs to reduce the communication overhead of the protocol before using floating point operations.

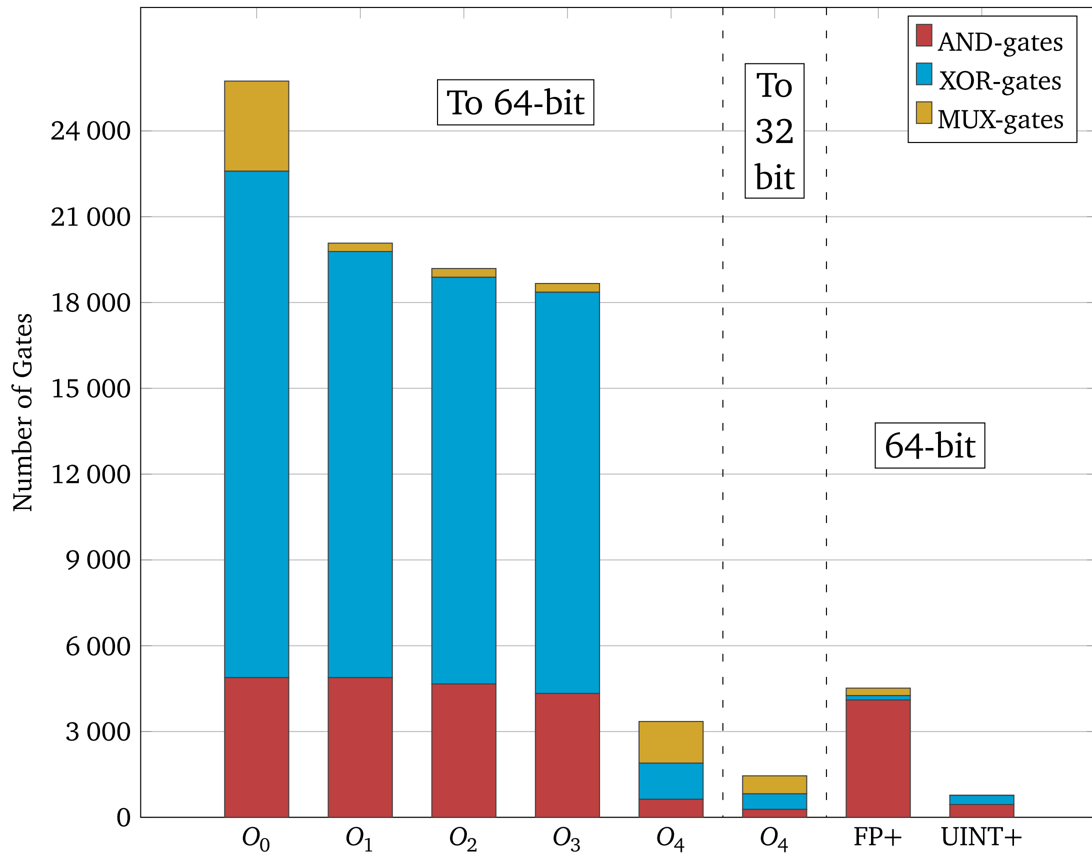


Figure 4.3: Integer to floating point number conversion and optimizations.

Table 4.1: Integer to floating point number conversion and optimizations.

	To 64-bit					To 32-bit	64-bit	
Gates	O_0	O_1	O_2	O_3	O_4	O_4	FP+	UINT+
AND	4 884	4 884	4 663	4 331	629	277	4 103	448
XOR	17 707	14 899	14 219	14 209	1 264	545	154	154
Other	3 151	293	307	305	1 458	630	260	1
Total	25 745	20 076	19 189	18 665	3 351	1 452	4 517	768

4.2 Chi-squared-test

For the χ^2-test we implement three different algorithm variants: the first one is not optimized, the second one is fully optimized and the last one is partially optimized. Using the term optimization we mean a deviation from the "classical" protocol for the purpose of improving its efficiency. We denote replacing all multiplication and addition operations in integer BCs with the corresponding operations in arithmetic sharing whenever possible as full optimization whereas replacing only addition operations is denoted as partial optimization. With non-optimized implementation we denote a pure BC containing only floating point number gates in ABY. A description of the χ^2-test algorithm is provided in Section 2.4.1 that also contains Equation (2.1) on p. 8.

4.2.1 Non-optimized

We use Equation (2.1) and boolean floating point number gates in ABY to implement the non-optimized algorithm. As mentioned in Section 2.6, floating point number gates are slower and larger than integer gates because of the more demanding operations that floating point arithmetic rely on. But, due to the precision and the variety of operations that floating point arithmetic allows, the usage of those gates can not always be avoided. An example of such kind of situation is the division operation in the χ^2-test. The result of the χ^2-test can vary in small ranges and hence integer arithmetic becomes unreliable when high precision is needed. Constable *et al.* [CTW+15] emulate 16-bit floating point division which is 2x less than our lowest precision (32-bit). The exact steps of the algorithm can be found in Protocol 4.3.

In the following, we give a detailed description of Protocol 4.3. The χ^2 function takes an array of matrices (see Table 2.1) *snps* as an input. At the very beginning, we create a return vector v and assign an empty set to it. In Line 2 we go into a loop for each matrix m in *snps* containing shares for case and control groups and two alleles and, again, reserve a return vector r as a result buffer. We precompute sums of observations in the alleles and groups assigning it to the list *sums*. We assign the total number of observations in m to $\langle n \rangle^B$ which equals $a + b + c + d$ in Equation (2.3) using the precomputed values from *sums*. After that, we perform a loop iteration for each entry $\langle entry \rangle^B$ in m. In Line 7, the number of observations in the allele at the position of $\langle entry \rangle^B$ in m is assigned to $\langle C \rangle^B$. Analogously, $\langle P \rangle^B$ gets assigned the number of observations for the group based on the position of $\langle entry \rangle^B$ in m (cf. Section 2.4.1). Both operations are performed on the precomputed values in *sums* to prevent redundant computation of the values. In Line 9, we compute the expected value $\langle e \rangle^B$ followed by assigning the value of *entry* to $\langle o \rangle^B$. Next, we append the statistic for the current entry to r, which is based on the already well-known χ^2 Equation (2.1). After that, we sum up all the values in r as a tree to get low depth, which form the χ^2 statistic for the current SNP. It is then appended to v which, at the end of the algorithm, contains statistic score for each SNP. Finally, v is returned as the result.

$v \leftarrow \chi^2(snps)$

1 : $v \leftarrow \emptyset$
2 : **foreach** m **in** $snps$ **do**
3 : $r \leftarrow \emptyset$
4 : $sums \leftarrow$ **PrecomputeSums**(m)
5 : $\langle n \rangle^B \leftarrow$ **TotalObs**$(sums)$
6 : **foreach** $\langle entry \rangle^B$ **in** m **do**
7 : $\langle C \rangle^B \leftarrow$ **SumAlleles**$(\langle entry \rangle^B, sums)$
8 : $\langle P \rangle^B \leftarrow$ **SumGroups**$(\langle entry \rangle^B, sums)$
9 : $\langle e \rangle^B \leftarrow \langle P \rangle^B \cdot \langle C \rangle^B / \langle n \rangle^B$
10 : $\langle o \rangle^B \leftarrow \langle entry \rangle^B$
11 : **Append**$(r, (\langle o \rangle^B - \langle e \rangle^B)^2 / \langle e \rangle^B)$
12 : **Append**$(v, \sum_i \langle r[i] \rangle^B)$
13 : **return** v

Protocol 4.3: Non-optimized χ^2-test algorithm.

The naive implementation of this algorithm in ABY requires a huge amount of RAM. An execution of the algorithm on 500 SNPs for one party reserves ~ 7 GB of memory, which does not satisfy our expectations of performing an efficient large-scale analysis. The amount of required RAM can be significantly reduced by using so-called SIMD gates in ABY (see Section 2.6.3). These gates evaluate one gate with longer input constructed from the initial inputs rather than building a separate gate for each input. Thus, we require much fewer gates for the same operations which are even a little bit faster compared to simple gates. The application of the SIMD functionality decreases the amount of used RAM by at least an order of magnitude. Occupying a similar amount of memory, it is possible to execute the protocol on $2^{13} = 8\,192$ SNPs in the aforementioned setting. In this algorithm we use only 32-bit floating point numbers.

4.2.2 Fully Optimized

For the optimized version of the protocol, we use the formula provided by [Gif14]:

$$\chi^2 = \frac{(ad - bc)^2(a + b + c + d)}{(a + b)(c + d)(b + d)(a + c)}, \tag{4.1}$$

where a, b, c and d are the entries of the matrix for a SNP. Values a and b correspond to the entries of allele A_1 for case (a) and control (b) group. Similarly, values c and d correspond to the entries of A_2 for cases and controls.

The aforementioned formula allows us to efficiently precompute almost all values using ABY's Arithmetic circuits (see Section 2.6). We can split the formula into 4 arithmetic operations: addition, subtraction, division and multiplication. The last one covers also the squaring operation. Since we operate on unsigned integers in ABY, we have to prevent subtraction operations yielding negative values. For that purpose, we make use of the extended expression $(a-b)^2 = a^2 + b^2 - 2ab$. The implementation details are shown in Protocol 4.4. Precomputation of values in arithmetic circuits for χ^2 leads to the need of only one floating point number operation for each SNP, i.e. the division of nominator by denominator in Equation (4.1).

$v \leftarrow \chi_f^2(snps)$

```
1 :  v ← ∅
2 :  foreach m in snps do
3 :    ⟨ad⟩^A ← m.⟨a⟩^A · m.⟨d⟩^A
4 :    ⟨bc⟩^A ← m.⟨b⟩^A · m.⟨c⟩^A
5 :    ⟨sum_1⟩^A ← m.⟨a⟩^A + m.⟨b⟩^A
6 :    ⟨sum_2⟩^A ← m.⟨c⟩^A + m.⟨d⟩^A
7 :    ⟨sum_3⟩^A ← m.⟨b⟩^A + m.⟨d⟩^A
8 :    ⟨sum_4⟩^A ← m.⟨a⟩^A + m.⟨c⟩^A
9 :    ⟨nom_left⟩^A ← ((⟨ad⟩^A · ⟨ad⟩^A) + ((⟨bc⟩^A · ⟨bc⟩^A) − ((⟨ad⟩^A · ⟨bc⟩^A)
10 :   ⟨nom_right⟩^A ← ⟨sum_1⟩^A + ⟨sum_2⟩^A
11 :   ⟨nom⟩^A ← ⟨nom_left⟩^A · ⟨nom_right⟩^A
12 :   ⟨denom⟩^A ← ((⟨sum_1⟩^A · ⟨sum_2⟩^A) · ((⟨sum_3⟩^A · ⟨sum_4⟩^A)
13 :   ⟨nom⟩^B ← INT2FP(⟨nom⟩^A, 64, 64)
14 :   ⟨denom⟩^B ← INT2FP(⟨denom⟩^A, 64, 64)
15 :   Append(v, ⟨nom⟩^B / ⟨denom⟩^B)
16 : return v
```

Protocol 4.4: Fully optimized χ^2-test algorithm.

In Protocol 4.4, the only input for the algorithm is a set of SNPs *snps*. The algorithm begins with reserving the result vector v by assigning an empty set to it. After that, we perform an iteration of the main routine for each matrix m, i.e. a SNP table, in *snps*. Then, in Lines 5 and 6, we compute $\langle ad \rangle^A$ and $\langle bc \rangle^A$ by multiplying the corresponding values in the matrix m. In Lines 5 to 8, we compute the sums of the elements according to the denominator computation in Equation (4.1). Afterwards, we precompute the left and right parts of the nominator in the same equation. It is worth to mention, that we use the aforementioned extended formula for $(a-b)^2$ to calculate it in unsigned integer arithmetic in Line 9. Having both nominator parts, we need to only multiply them as we do in Line 10 which results in $\langle nom \rangle^A$. After that, we calculate the denominator by multiplying the precomputed sums of the rows and columns of

the matrix m. We do this by multiplying the left and right parts of the equation and then we multiply the results. This action reduces the depth of the circuit from 3 to 2 multiplications, because we perform the first two multiplications in parallel. In Lines 13 and 14, we convert the arithmetic shares $\langle nom \rangle^A$ and $\langle denom \rangle^A$ to the 64-bit double precision floating point numbers shared in the Boolean sharing. Finally, we calculate the last division of the nominator $\langle nom \rangle^A$ by the denominator $\langle denom \rangle^A$ and append the result to the vector v, which is returned after the main routine ends.

For this setting, because of a big number of multiplications, we use 64-bit input values for arithmetic sharing and convert the values into 64-bit double precision floating point numbers. The reason for not converting the Arithmetic shares to the 32-bit representation is that in Equation (4.1) we perform many additions followed by many multiplications. This results in an integer number overflow or in the resulting number being greater than the fraction size which we want to prevent to not lose the precision of the results. Since we have to perform only one operation using floating point arithmetic, i.e. division of the nominator by the denominator, we assume it to be a better choice.

4.2.3 Partially Optimized

In this section, we describe the partially optimized version of the χ^2-test algorithm. Similarly to Section 4.2.2, we use Equation (4.1) to compute the statistic, but here we optimize only the addition operation in Arithmetic sharing. This is due to an integer overflow in the fully optimized algorithm when the number of participants becomes relatively large, i.e. $>3\,000$ participants. Since we can accept only correct results and want to perform computations assuming a very large number of participants, we implement this version of the protocol. It can be seen as a trade-off between the better efficiency of conversion gates (see Section 2.6.4) and the broader functionality of slower floating point operations. The algorithm is described in Protocol 4.5.

In the following, we describe the single steps of Protocol 4.5. This algorithm is very similar to the fully optimized χ^2-test algorithm. The only one exception here is that we optimize only the addition gates and perform all other operations in a BC. Similarly to the previous algorithm, we precompute sums of the rows and the columns $\langle sum_i \rangle^A$ of the matrix m as well as the total number of observations n. After that, we convert all entries of m, $\langle sum_i \rangle^A$, and n to 32-bit floating point numbers in the Boolean sharing. In Lines 15 and 16, we calculate the multiplications $a \cdot d$ and $b \cdot c$ which we then use to compute the left part of the nominator equation in Line 17 where we do not need the extended formula for $(a - b)^2$ anymore and calculate it directly. Afterwards, we calculate the nominator $\langle nom \rangle^B$ by multiplying the precomputed left part of the nominator $\langle nom_{left} \rangle^B$ by the right part which is $\langle n \rangle^B$. We compute the multiplication of $\langle sum_i \rangle^B$ using the depth-optimized multiplication strategy analogously to Section 4.2.2. As the last step, we divide $\langle nom \rangle^B$ by $\langle denom \rangle^B$ and append it to the vector v which is returned as the result of the function after we processed all SNPs.

$v \leftarrow \chi_p^2(snps)$

1 : $v \leftarrow \emptyset$
2 : **foreach** m **in** $snps$ **do**
3 : $\langle sum_1 \rangle^A \leftarrow m.\langle a \rangle^A + m.\langle b \rangle^A$
4 : $\langle sum_2 \rangle^A \leftarrow m.\langle c \rangle^A + m.\langle d \rangle^A$
5 : $\langle sum_3 \rangle^A \leftarrow m.\langle b \rangle^A + m.\langle d \rangle^A$
6 : $\langle sum_4 \rangle^A \leftarrow m.\langle a \rangle^A + m.\langle c \rangle^A$
7 : $\langle n \rangle^A \leftarrow \langle sum_1 \rangle^A + \langle sum_2 \rangle^A$
8 : $\langle a \rangle^B \leftarrow$ **INT2FP**$(m.\langle a \rangle^A, 32, 32)$
9 : $\langle b \rangle^B \leftarrow$ **INT2FP**$(m.\langle b \rangle^A, 32, 32)$
10 : $\langle c \rangle^B \leftarrow$ **INT2FP**$(m.\langle c \rangle^A, 32, 32)$
11 : $\langle d \rangle^B \leftarrow$ **INT2FP**$(m.\langle d \rangle^A, 32, 32)$
12 : **for** i **in** $1:4$ **do**
13 : $\langle sum_i \rangle^B \leftarrow$ **INT2FP**$(\langle sum_i \rangle^A, 32, 32)$
14 : $\langle n \rangle^B \leftarrow$ **INT2FP**$(\langle n \rangle^A, 32, 32)$
15 : $\langle ad \rangle^B \leftarrow \langle a \rangle^B \cdot \langle d \rangle^B$
16 : $\langle bc \rangle^B \leftarrow \langle b \rangle^B \cdot \langle c \rangle^B$
17 : $\langle nom_{left} \rangle^B \leftarrow (\langle ad \rangle^B - \langle bc \rangle^B)^2$
18 : $\langle nom \rangle^B \leftarrow \langle nom_{left} \rangle^B \cdot \langle n \rangle^B$
19 : $\langle denom \rangle^B \leftarrow (\langle sum_1 \rangle^B \cdot \langle sum_2 \rangle^B) \cdot (\langle sum_3 \rangle^B \cdot \langle sum_4 \rangle^B)$
20 : **Append**$(v, \langle nom \rangle^B / \langle denom \rangle^B)$
21 : **return** v

Protocol 4.5: Partially optimized χ^2-test algorithm.

Since we optimize only addition, there is only a small probability that we will exceed the 23-bit fraction size of the floating point number. Hence, we use 32-bit additive sharing in this algorithm which we then convert into 32-bit single precision floating point numbers. We operate on 32-bit numbers and the fraction size of a 32-bit floating point number is 23 bits. Consequently, we assume that the overall number of GWAS participants will be always smaller than 2^{23} (8 388 608) for the partially optimized algorithm. This is a relatively big number and it is supposed to be more than enough for most application scenarios. However, one could use 64-bit numbers for a scenario requiring a larger number of participants.

4.3 G-test

Analogously to the previous section, we implement three versions of the G-test: non-optimized, fully optimized and partially optimized. The non-optimized version uses only floating point arithmetic in Boolean sharing. The full optimization uses additions and multiplications in arithmetic sharing and conversion gates to convert numbers into boolean floating point numbers. The partial optimization optimizes only addition operations in the Arithmetic sharing.

4.3.1 Non-optimized

The structure of the non-optimized algorithm is shown in Protocol 4.6. The general idea of this algorithm is a straightforward implementation of Equation (2.6) on p. 10. The only deviation from the protocol is that the sums of observations are precomputed for a SNP at the beginning of the protocol in order to prevent repeated computation of the same values.

In the following, we describe Protocol 4.6 in detail. The function $g(snps)$ takes the vector of SNPs *snps* as input and returns the vector of resulting shares v as output. At the beginning of the protocol, we reserve the result vector v and perform a loop iteration for each matrix m in *snps*. The matrix m consists of four values containing the number of observations for two alleles and two groups, i.e. case and control groups. Afterwards, we reserve the result vector for the statistic r and precompute the sums of observations in the columns and rows, and assign the resulting vector to *sums*. After that, we compute the total number of observations in m using the precomputed sum values and assign the result to the share $\langle n\rangle^B$. In Line 6, we perform a loop iteration for each entry $\langle entry\rangle^B$ of the matrix m. We assign the numbers of observations in alleles and groups to the corresponding values $\langle C\rangle^B$ and $\langle P\rangle^B$ also using the precomputed values in *sums* based only on the position of $\langle entry\rangle^B$ in m. In Line 9, we compute the expected value $\langle e\rangle^B$ for $\langle entry\rangle^B$ in m. We assign the value of $\langle entry\rangle^B$ to $\langle o\rangle^B$ for better readability and compute the complete statistic by dividing the observed value $\langle o\rangle^B$ by the expected value $\langle e\rangle^B$ and calculating the natural logarithm of the result which we assign to $\langle s\rangle^B$. At the end of the protocol, we perform the last operation for the statistic, which is the

$v \leftarrow g(snps)$

```
1 :  v ← ∅
2 :  foreach m in a do
3 :    r ← ∅
4 :    sums ← PrecompSums(m)
5 :    ⟨n⟩^B ← TotalObs(sums)
6 :    foreach ⟨entry⟩^B in m do
7 :      ⟨C⟩^B ← SumAlleles(⟨entry⟩^B, sums)
8 :      ⟨P⟩^B ← SumGroups(⟨entry⟩^B, sums)
9 :      ⟨e⟩^B ← ⟨P⟩^B · ⟨C⟩^B / ⟨n⟩^B
10 :     ⟨o⟩^B ← ⟨entry⟩^B
11 :     ⟨s⟩^B ← ⟨o⟩^B / ⟨e⟩^B
12 :     ⟨s⟩^B ← ln(⟨s⟩^B)
13 :     ⟨s⟩^B ← ⟨s⟩^B · ⟨o⟩^B
14 :     Append(r, (⟨s⟩^B)
15 :   Append(v, ∑_i ⟨r[i]⟩^B)
16 : return v
```

Protocol 4.6: Non-optimized G-test algorithm.

multiplication of the aforementioned result by $\langle o \rangle^B$ and appending the result to the vector r. Now we can compute the sum of the computed values which corresponds to the statistic for the current SNP and append it to the vector v.

4.3.2 Fully Optimized

The fully optimized G-test is implemented as follows: the algorithm takes 32-bit additively secret-shared values as input and produces 32-bit boolean shares as output. The protocol uses unsigned integer arithmetic in the arithmetic sharing and floating point arithmetic in the boolean sharing. In the fully optimized algorithm, we optimize addition as well as multiplication operations to maximally utilize the advantages of the protocol conversion gates. We show the implementation details of the current algorithm in Protocol 4.7.

$v \leftarrow g_f(snps)$

```
1 :  v ← ∅
2 :  foreach m in snps do
3 :    r ← ∅
4 :    sums ← PrecompSums(m)
5 :    ⟨n⟩^A ← TotalObs(sums)
6 :    ⟨n⟩^B ← INT2FP(⟨n⟩^A, 32, 32)
7 :    foreach ⟨entry⟩^A in m do
8 :      ⟨C⟩^A ← SumAlleles(⟨entry⟩^A, sums)
9 :      ⟨P⟩^A ← SumGroups(⟨entry⟩^A, sums)
10 :     ⟨e⟩^A ← ⟨P⟩^A · ⟨C⟩^A
11 :     ⟨entry⟩^B ← INT2FP(⟨entry⟩^A, 32, 32)
12 :     ⟨e⟩^B ← INT2FP(⟨e⟩^A, 32, 32)
13 :     ⟨e⟩^B ← ⟨e⟩^B / ⟨n⟩^B
14 :     ⟨o⟩^B ← ⟨entry⟩^B
15 :     ⟨s⟩^B ← ⟨o⟩^B / ⟨e⟩^B
16 :     ⟨s⟩^B ← ln(⟨s⟩^B)
17 :     ⟨s⟩^B ← ⟨s⟩^B · ⟨o⟩^B
18 :     Append(r, (⟨s⟩^B)
19 :   Append(v, ∑_i ⟨r[i]⟩^B)
20 :  return v
```

Protocol 4.7: Fully optimized G-test algorithm

In Protocol 4.7 the function g_f takes an array of SNPs *snps* as input and produces a result vector v as output. The algorithm begins with allocation of the result vector v after which a

loop iteration is performed for each matrix *m* in *snps*. In Line 3, we reserve a result vector *r* for *m* and then precompute sums of the columns and rows of *m*, and store them in the variable *sums*. Based on *sums*, we also compute the total number of observations in *m* using the function **TotalObs** and convert the resulting share into a floating point number shared in the Boolean sharing. Afterwards, an iteration of the routine is performed for each entry $\langle entry \rangle^A$ in *m*. First, we compute the sums of alleles and groups in *m* based on the position of $\langle entry \rangle^A$ using the precomputed array *sums*. After that, the expected value $\langle e \rangle^A$ is calculated by multiplying $\langle P \rangle^A$ and $\langle C \rangle^A$, converting $\langle e \rangle^A$ and $\langle entry \rangle^A$ to Boolean floating point numbers and dividing $\langle e \rangle^B$ by $\langle n \rangle^B$. As the last step, we divide the observed value $\langle o \rangle^B$ by $\langle e \rangle^B$ and perform the calculation of the natural logarithm of the resulting value, and then we multiply it by $\langle o \rangle^B$. At the end of the routine, the resulting value $\langle s \rangle^B$ is appended to *r* and after performing all iterations we append the sum of the shares in *r* to *v*.

When testing the fully optimized version of the G-test we already faced problems for >2 000 participants which is due to the integer overflow caused by many multiplication operations and small 32-bit numbers, thus yielding incorrect results. However, the G-test requires much more operations to be computed in floating point arithmetic after the conversion compared to the χ^2-test. That is why the use of 64-bit values will make the routine very inefficient, and this is the reason to keep 32-bit numbers for this protocol.

4.3.3 Partially Optimized

The implementation of the partially optimized G-test algorithm is shown in Protocol 4.8. The implementation uses a similar strategy as was used in Section 4.2.3, but using the standard Equation (2.6) on p. 10. The difference to Section 4.3.2 is that only addition operations are optimized where it is possible using the Arithmetic sharing. This strategy results in a good performance and allows a large number of participants which is limited to 8 388 608 (2^{23}). In this protocol, 32-bit unsigned integer numbers in Arithmetic sharing are used as an input and 32-bit floating point numbers in Boolean sharing as an output.

The function g_p described in Protocol 4.8 takes an array of SNPs *snps* as an input and yields a result vector *v* as an output. First of all, we reserve a vector *v* for the results and perform an iteration of the routine for each matrix *m* in *snps*. After that, a result vector *r* for *m* is reserved. In Line 4, sums of observations in the rows and columns of *m* are computed and stored in the variable *sums*. Next, the total number of observations in *m* is computed and converted to the floating point number representation in Boolean sharing which is then stored in $\langle n \rangle^B$. Each variable in *sums* is also converted to the floating point representation in Boolean sharing. In Line 9, we perform a loop iteration for each entry $\langle entry \rangle^A$ in *m*. First, $\langle entry \rangle^A$ is converted to a floating point number in Boolean sharing. Afterwards, the sums of observations in alleles and groups are computed based on the position of $\langle entry \rangle^B$ in *m* using the precomputed values in *sums*, and are stored in the corresponding shares $\langle C \rangle^B$ and $\langle P \rangle^B$. In Line 13, the expected value is computed using the basic χ^2-test equation. After that, we compute the resulting value $\langle s \rangle^B$ dividing $\langle o \rangle^B$ by $\langle e \rangle^B$. We also perform the logarithm operation on the result and

```
v ← g_p(snps)
1 :  v ← ∅
2 :  foreach m in snps do
3 :      r ← ∅
4 :      sums ← PrecompSums(m)
5 :      ⟨n⟩^A ← TotalObs(sums)
6 :      ⟨n⟩^B ← INT2FP(⟨n⟩^A, 32, 32)
7 :      for i in 1 : 4 do
8 :          ⟨sums[i]⟩^B ← INT2FP(⟨sums[i]⟩^A, 32, 32)
9 :      foreach ⟨entry⟩^A in m do
10 :         ⟨entry⟩^B ← INT2FP(⟨entry⟩^A, 32, 32)
11 :         ⟨C⟩^B ← SumAlleles(⟨entry⟩^B, sums)
12 :         ⟨P⟩^B ← SumGroups(⟨entry⟩^B, sums)
13 :         ⟨e⟩^B ← ⟨P⟩^B · ⟨C⟩^B / ⟨n⟩^B
14 :         ⟨o⟩^B ← ⟨entry⟩^B
15 :         ⟨s⟩^B ← ⟨o⟩^B / ⟨e⟩^B
16 :         ⟨s⟩^B ← ln(⟨s⟩^B)
17 :         ⟨s⟩^B ← ⟨s⟩^B · ⟨o⟩^B
18 :         Append(r, (⟨s⟩^B)
19 :     Append(v, ∑_i ⟨r[i]⟩^B)
20 : return v
```

Protocol 4.8: Partially optimized G-test algorithm

multiply it by $\langle o \rangle^B$. The resulting value is then stored in r. Finally, we append the sum of the statistics for single entries in m and append the result to the vector v.

4.4 Extended Chi-squared-test and G-test

Using the term "extended" test, we describe a column extension of the matrix shown in Table 2.1. We construct a contingency table with k gene counts for both case and control groups. While the columns in Table 2.1 correspond to counts of single alleles in observed genotypes, here, columns correspond to codewords, i.e. genotypes or genotype sequences, whose length can vary depending on the task. The idea of this approach is to prevent the loss of information, e.g. the information that the first allele in a genotype is dominant and the second one is recessive will be considered in this method. The loss occurs due to the dimension reduction of genotypes to only two counts of alleles. The data representation for the extended χ^2-test is shown in Table 4.2, where columns correspond to the codeword IDs and rows to the groups. Therefore, the variable in a cell (i, j) corresponds to the number of observations of the codeword with id_j in the $group_i$.

Table 4.2: Contingency table for the extended χ^2-test.

	id_1	id_2	...	id_n
$group_1$	$obs_{1,1}$	$obs_{1,2}$	...	$obs_{1,n}$
$group_2$	$obs_{2,1}$	$obs_{2,2}$	...	$obs_{2,n}$

A statistic can be computed on this table using Equations (2.1) and (2.6). The only one difference is the calculation of observed and expected values. The calculation of the χ^2-test causes a large overhead using the extended contingency table. In order to reduce the overhead, the values are calculated as described in Section 2.4.1 for the goodness-of-fit test.

For summing up columns we construct an addition binary tree. This does not change the number of operations we have to perform, but reduces the depth of the subtree to $O(\log n)$ from $O(n)$ compared to the sequential addition.

4.5 P-test

The P-test is based on the other statistics. It checks if the result of the statistic is significant with respect to some significance value α (see Section 2.4.3). We use the expression $\mathbf{S}(snps)$ to denote performing a generic statistical test on an array of SNPs *snps*.

In the following, we describe Protocol 4.9 in detail which computes the P-test. The algorithm gets an array of SNPs *snps*, as also used in the underlying statistics, containing matrices m and

$v \leftarrow g_p(snps, \langle c \rangle^B)$

1 : $v \leftarrow \mathbf{S}(snps)$
2 : **foreach** $\langle s \rangle^B$ **in** v **do**
3 : $\quad \langle d \rangle^B \leftarrow \langle c \rangle^B - \langle s \rangle^B$
4 : $\quad \langle s \rangle^B \leftarrow \mathbf{MUX}(\langle 1 \rangle^B, \langle 0 \rangle^B, \langle d[msb] \rangle^B)$
5 : **return** v

Protocol 4.9: P-test algorithm

a share in boolean sharing containing the critical value c derived from α (see Section 2.4.1). As in the first step of this algorithm, we apply a static **S** on *snps* and obtain a result vector v with a computed test value for each SNP. After that, an iteration of the routine is performed on each test result share $\langle s \rangle^B$ in v. We subtract $\langle s \rangle^B$ from the critical value and assign its result to difference share $\langle d \rangle^B$ which is then used as follows: the most significant bit *msb* in $\langle d \rangle^B$ indicates whether the value is negative and thus implies that $\langle s \rangle^B$ was greater than $\langle c \rangle^B$ if the value is negative. Finally, if $\langle s \rangle^B$ was greater than $\langle c \rangle^B$, $\langle s \rangle^B$ is set to $\langle 1 \rangle^B$ and $\langle 0 \rangle^B$ otherwise. After the routine is finished, the altered vector v containing only secret-shared zeros and ones is returned as the result.

4.6 Outsourcing Computation

In the outsourcing routine, multiple parties send their pre-shared inputs to the semi-trusted third parties in order to perform collaborative SMPC. The routine consists of the following three parts:

Outsourcing inputs Here, we use the already available ABY routine for creating shares to avoid implementing the routines on our own as computing shares locally without ABY leads to very low-level programming, especially for floating point numbers. After shares are created locally using two ABY instances, an institution sends shares s_0 and s_1 to the non-collaborative semi-trusted third parties T_0 and T_1.

Executing the algorithm The servers, after receiving the shares, run the routine for calculating the algorithm and send their outputs back to the institutions.

Revealing results After receiving the result shares, an institution reconstructs plaintext results from shares using, again, two local ABY instances.

Hence, the outsourcing algorithm differs from the standard algorithms only in one aspect, namely handling input and output gates. Instead of the usual *PutINGate* and *PutOUTGate* methods in the ABY circuit, we use the *PutSharedINGate* and *PutSharedOUTGate* routines. The first one requires pre-shared values as input and produces shared values, whereas the second one performs it vice-versa.

5 Evaluation

5.1 Benchmarking Environment

We run all our benchmarks on two servers that are equipped with Intel Core i7-4770K processors, 16 GB of RAM, and 1 TB *Solid-State Drives (SSDs)*.

In the LAN setting, our servers have a bandwidth of 1 GB/s and the network latency is restricted to 1.2 ms, whereas for the WAN setting we restrict the bandwidth to 100 MB/s and the network latency to 100 ms.

5.2 Benchmarking in the LAN Setting

In the following, we describe and analyze the benchmarking results of the implemented statistical tests in the LAN setting.

5.2.1 Chi-squared-test

The benchmarking results of all the variants of the χ^2-test in the LAN setting are depicted in Figure 5.1. We benchmark the algorithms with as many SNPs as possible while keeping the execution using only RAM to not influence run-times by using the swap space on the hard drive, which is fast on the SSD, but still slower by a factor 2-3x compared to the computation using only RAM.

It can be seen that the non-optimized version of the algorithm is significantly (1.5-4x) slower than the two optimized versions. This is not only due to the optimization using conversion gates, but also due to the more suitable equation for this task. The crucial remark is that the partially optimized algorithm is only slightly slower than the fully optimized algorithm, but allows a huge number of participants in GWAS.

More precisely, the fully optimized χ^2 algorithm is by a factor 1.6-3.5x faster than the non-optimized, and the partially optimized algorithm is by a factor 1.5-3x faster than the non-optimized. The fully-optimized algorithm is by a factor 1.4x faster than the partially optimized. The extended algorithm with 2 IDs is on average by a factor 1.35x faster than the non-optimized.

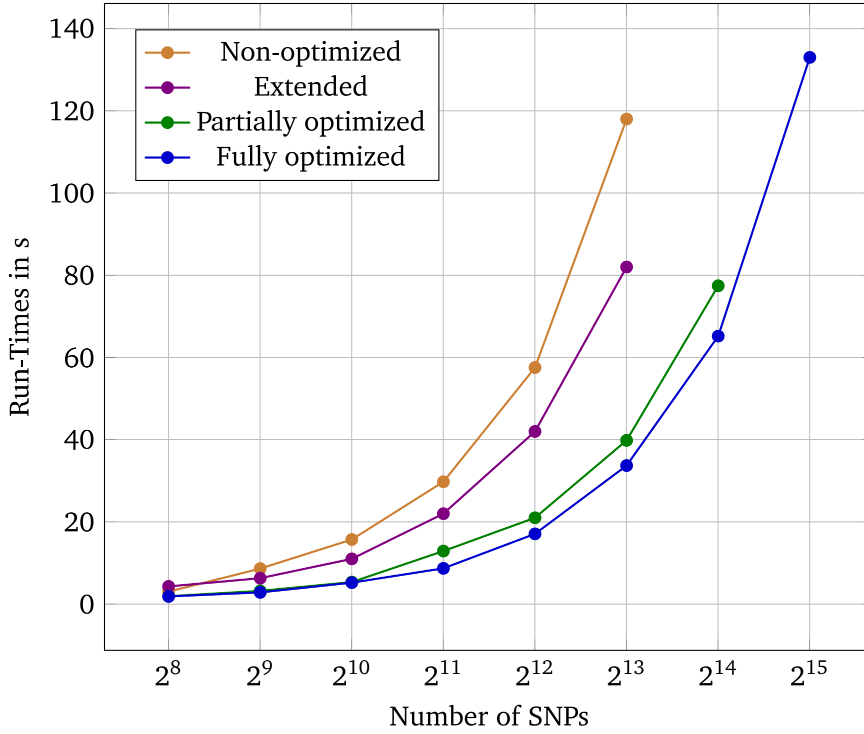


Figure 5.1: Run-times of the χ^2-test algorithms in ABY in the LAN setting.

5.2.2 G-test

The performance results of the G-test are depicted in Figure 5.2. Since we use only one equation for all implementations, the difference between the versions is not so significant as for the χ^2-test. However, a tendency of the performance improvement can be seen directed to the optimized algorithms. The fully optimized algorithm is by a factor 1.5-1.7x faster than the non-optimized, and the partially optimized algorithm is by factor 1.2-1.5x faster than the non-optimized. The fully-optimized algorithm is by a factor 1.1-1.3x faster than the partially optimized. The extended algorithm is on average by a factor 1.4x faster than the non-optimized.

In addition, the running times of the χ^2- and G-test are shown in Table 5.1. As it can be seen, the optimized versions of the χ^2 algorithms are more efficient than the other algorithms. On the other hand, the G-test nearly reaches the performance of the non-optimized χ^2-test algorithm only in the fully optimized algorithm. The non-optimized χ^2-test is by a factor 1.3-2x faster than the corresponding G-test. We benchmarked only the settings where the ABY instance fits into RAM. That is why most of the statistics are marked with "-" in the last two rows and only the most light-weight ones (χ^2_p, χ^2_f, and p) succeeded.

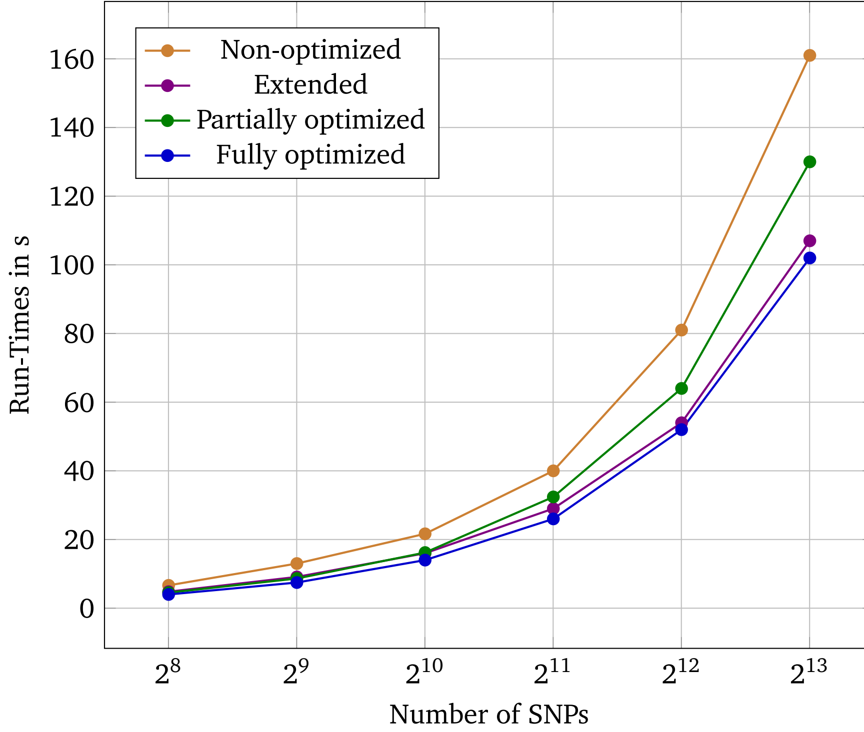


Figure 5.2: Run-times of the G-test algorithms in ABY in the LAN setting.

Table 5.1: Run-times for the χ^2-, G- and P-test algorithms in ABY in the LAN setting in seconds.

# SNPs	χ^2	χ_e^2	χ_p^2	χ_f^2	g	g_e	g_p	g_f	p
2^8	3.0	4.3	1.9	1.8	6.6	4.8	4.5	3.9	0.2
2^9	8.6	6.3	3.2	2.8	13	9.1	8.6	7.4	0.4
2^{10}	15	11	5.3	5.2	21	16	16	14	0.7
2^{11}	29	22	12	8.6	40	29	32	26	1.3
2^{12}	57	42	21	17	81	54	64	52	2.6
2^{13}	118	82	39	33	161	107	130	102	4.8
2^{14}	-	-	77	65	-	-	-	-	9.5
2^{15}	-	-	-	133	-	-	-	-	19

5.2.3 Comparison with [CTW+15]

In Figure 5.3, we compare the run-times of our χ^2 algorithms and the χ^2 algorithm introduced in [CTW+15]. Their implementation is not publicly accessible. Hence, we use the benchmarking results mentioned in the paper. We also use the number of SNPs for the benchmarks that they use in the paper. They do not provide much information about their

results, but only for 311 and 9 330 SNPs. That is why we show only the results for those two SNP counts. Since we have a similar benchmarking environment, the results are roughly comparable in the LAN setting.

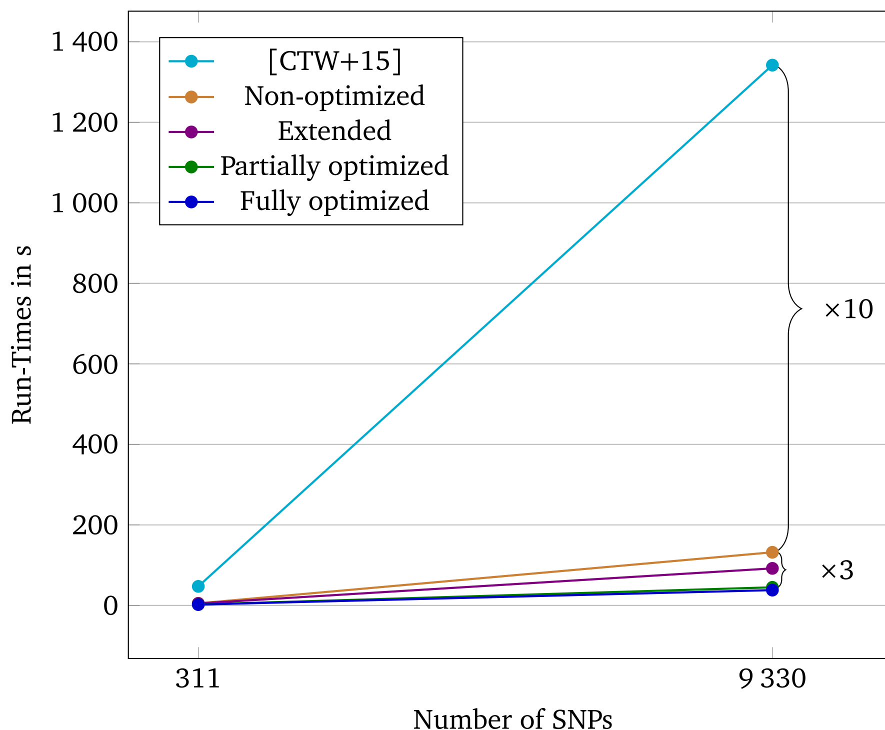


Figure 5.3: Comparison of the run-times of our χ^2-test algorithms with those of [CTW+15] in the LAN setting.

As appears from Figure 5.3, the run-times of the algorithm introduced in [CTW+15] are always worse than ours, even compared to our slowest algorithms. As depicted in the figure, the non-optimized χ^2-test outperforms the solution of Constable *et al.* by factor 10x. Both, the fully and the partially optimized algorithm, produce roughly 30x better results compared to [CTW+15]. Therefore, we assume their algorithms to be slower also in our benchmarking environment.

The exact comparison results are also shown in Table 5.2. More precisely, the non-optimized, fully, and partially optimized algorithms are accordingly 9.2x, 22.3x, and 20.4x faster for the 311 SNPs setting and 10.1x, 35.3x, and 29.8x faster for the 9330 SNPs setting. We suggest using the partially optimized algorithm, because it is almost as fast as the fully optimized algorithm, but can handle a large number of participants in GWAS.

Table 5.2: Comparison of the run-times of our χ^2-test algorithms with those of [CTW+15] in the LAN setting in seconds.

	311 SNPs	9330 SNPs
[CTW+15]	47	1342
Non-optimized	5.1	132
Extended	4.9	92
Partially optimized	2.3	45
Fully optimized	2.1	38

5.2.4 P-test

We depict the benchmarking results of the P-test in Figure 5.4. As the underlying statistic for the P-test, we use our three variants of the χ^2, and G-test. The overhead caused by the P-test is marked in yellow. This benchmark is performed on 8 192 SNPs, which is the upper bound for all statistics keeping the computation only in RAM. As appears from the figure, the P-test overhead is constant with a small jitter because of the network connection and comparably small compared to the run-time of any of the statistics (2-4.3 s, which amounts to 1-11 % of the total run-time). The P-test, as well as the underlying statistic, has linear complexity in the number of SNPs. Thus, the overhead will grow proportionally with an increasing number of SNPs.

We show the results of the P-test benchmarking in Table 5.3. The P-test routine definitely does not add a significant overhead to the protocol. That is why we assume that also the P-test will be much faster using any of the aforementioned tests as the underlying statistic compared to the χ^2-test in [CTW+15].

Table 5.3: P-test run-time overhead on 2^{13}=8 192 SNPs based on the run-times of the underlying statistic in seconds.

	χ^2	χ^2_p	χ^2_f	g	g_p	g_f
Used statistic	118	39	33	161	130	102
P-test overhead	4	2.5	4.2	2	3	2
Total run-time	122	42	38	163	133	104

Since the P-test can be performed without a significant overhead, one can efficiently apply it to hide the exact results of the statistic, but still allow to perform hypothesis testing on data, and parametrize the algorithm by setting different significance threshold values.

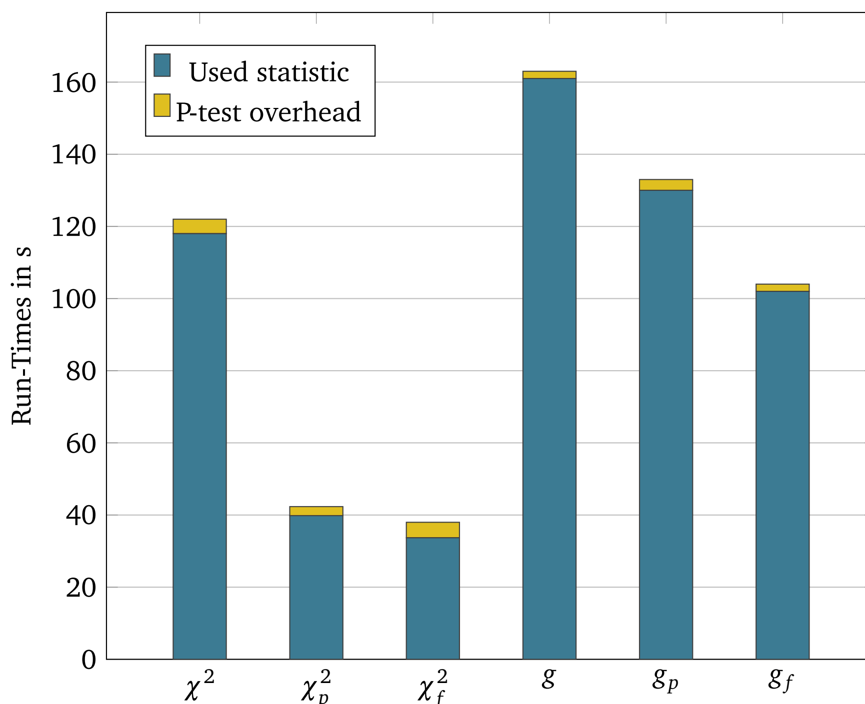


Figure 5.4: P-test run-time overhead on 2^{13}=8 192 SNPs based on the run-times of the underlying statistic with and without the P-test.

5.3 Benchmarking in the WAN Setting

In Figure 5.5, we depict the benchmarking results of the implemented χ^2-test algorithms in the WAN setting. We want to emphasize the substantial overhead for all protocols in the WAN setting compared to the LAN setting. Furthermore, the partially optimized χ^2-test algorithm performs better than the fully optimized due to a by 22% larger number of AND-gates.

In our benchmarks, the algorithms in the WAN setting were approximately by an order of magnitude slower than in the LAN setting. This is a big difference that was a motivation to find a way of using our algorithms in the LAN setting by keeping the usage scenarios practical.

5.4 Outsourcing Computation

In the outsourcing scenario, we have multiple medical institutions that want to perform a GWAS analysis on an aggregated dataset. The institutions are randomly located around the

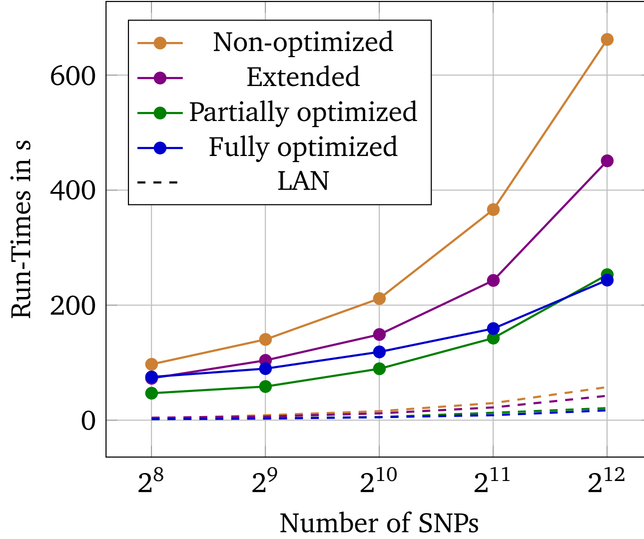


Figure 5.5: Run-times of the χ^2-test algorithms in ABY in the WAN setting.

world. Since SMPC is much slower in high-latency networks, they want to outsource the computation to the non-colluding *Semi-Trusted Third Parties (STTPs)*. Institutions are connected to the STTPs via WAN. STTPs, on the other hand, are connected via LAN.

Institutions locally secret share their data and send it to STTPs via a *Transport Layer Security (TLS)* channel, such that an attacker cannot intercept both shares during transfer and reconstruct the secret data from them. The STTPs receive the data and compute a predefined routine for statistical testing. After that, they send the data to each institution, which reconstructs the result from the shares received from the STTPs.

5.4.1 Local Share Creation

The time for the local generation of shares is shown in Table 5.4. We use ABY to generate shares in order to abstract from the details of the manual protocol processing. The evaluation of the circuit routine is slower than dealing with bit-values, but much more simple and demands much less manual work.

As can be seen from Table 5.4, Arithmetic shares from integer numbers are generally generated faster than Boolean shares from floating point numbers. More detailed, 64-bit Arithmetic share generation is 1.8-4.5x faster than Boolean, and 32-bit Arithmetic share generation is by a factor 2-3x faster than that for 64-bit numbers. In the table, we provide run-times only for the sharings and bit-lengths that were used to produce input shares in this thesis, i.e. 32-bit Boolean sharing for the non-optimized algorithms, 64-bit Arithmetic sharing for the fully optimized χ^2-test, and 32-bit Arithmetic sharing for all other algorithms.

Table 5.4: Run-times in milliseconds for the local generation of outsourced inputs in Arithmetic (A) and Boolean (B) sharings including instantiation time of the ABY routine.

# SNPs	32-bit (A)	64-bit (A)	32-bit (B)
2^8	1.5	3.1	5.7
2^9	2.4	5.5	17
2^{10}	4.2	10	38
2^{11}	7.7	19	79
2^{12}	14	38	171
2^{13}	28	76	353
2^{14}	56	153	704
2^{15}	113	306	1397

5.4.2 Benchmarking

For the purpose of comparison, we run our implemented tests for different numbers of parties in order to determine how much overhead an increased number of protocol participants brings in GWAS. The results of this experiment is shown in Table 5.5.

As it appears from the table, the tests where input numbers are shared in Arithmetic sharing have a nearly constant run-time on the number of involved parties. This is due to non-interactive addition in Arithmetic sharing, i.e. absolutely no communication between parties is required. The additionally required computation time however does not influence the results for even a 128-fold difference in the number of participants.

For the optimized χ^2-tests, the run-times are very similar and are roughly almost always 2.5 seconds, whereas for the G-tests we observe higher run-times for the fully optimized (4 s, which is slower by a factor 1.6x) and the partially optimized (5 s, which is slower by a factor 2x) algorithms, which also remain nearly constant in the number of participants.

Nevertheless, the algorithms that are run in only Boolean sharing are (i) much slower and (ii) constantly grow in the number of participants. Performing tests on a small number of participants, the overhead growth is not intensive, e.g. 1.26x (2 parties), 1.19x (4 parties), 1.37x (8 parties), etc. for the non-optimized χ^2-test. This changes with a large number of participants, e.g. for the same χ^2-test 1.86x (128 parties), 1.92x (256 parties). The run-times of the non-optimized G-test follow a similar tendency.

In Figure 5.6, we depict the information from Table 5.5. We limit it to 256 parties, such that the execution of the protocol fits into RAM in ABY. However, the run-times for Arithmetic sharing will be valid also for a larger number of participants, because we can sum up all the shares before the execution of the ABY instance. In the figure, the growth of run-times of the non-optimized algorithms can be perceived more intuitively. The main point of this figure

Table 5.5: Run-times of χ^2- and G-test on 256 SNPs for multiple parties in the LAN setting in seconds.

# Parties	χ^2	χ^2_f	χ^2_p	g	g_f	g_p
2	4.1	2.3	2.5	6.3	4	5
4	5.2	2.5	2.5	6.9	4	5
8	6.2	2.5	2.5	7.8	4	5
16	8.5	2.5	2.5	10	4	5
32	13	2.5	2.5	15	4	5
64	22	2.5	2.5	24	4	5
128	41	2.5	2.5	43	4	5
256	79	2.5	2.5	82	4	5

is that only the non-optimized algorithms (marked orange) have non-constant growth of run-times.

For the purpose of comparison, we fixed the number of SNPs to 256 which was the maximum such that all algorithms finish without running out of RAM to prevent increasing the run-times by using swap space.

We exclude the P-test from this benchmark, because it has a constant overhead on top of the underlying statistic that depends on the number of SNPs. Since the number of SNPs is constant in this case, the P-test will not change the tendency of growth of the single algorithms, and the tendency of growth is what we want to show in Figure 5.6.

Secure Trasfer of Shares

We performed an analysis of the transfer times in the WAN setting using TLS, which is used for securing the communication between institutions and STTPs in order to protect transferred shares from disclosure. We observed that the transfers have large variations in the transfer speed. Speed for transferring data from 128 B up to 512 MB varies in the range 32-98 Mbit/s. Despite these variances, the time required for transferring shares is negligible compared to the run-time for any of the protocols.

Reconstruction of Results

The time for parties to locally retrieve the results from the received shares for all protocols is ~ 1.6 ms for 256 SNPs, which is also negligible compared to the run-time of any of the protocols.

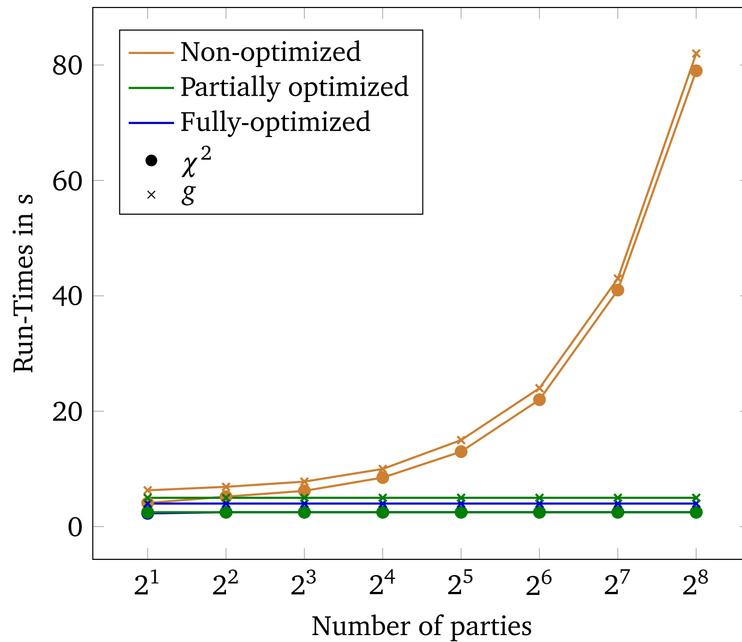


Figure 5.6: Run-times of our χ^2- and G-test algorithms on 256 SNPs for multiple parties in the LAN setting in the outsourcing scenario.

Extended algorithm

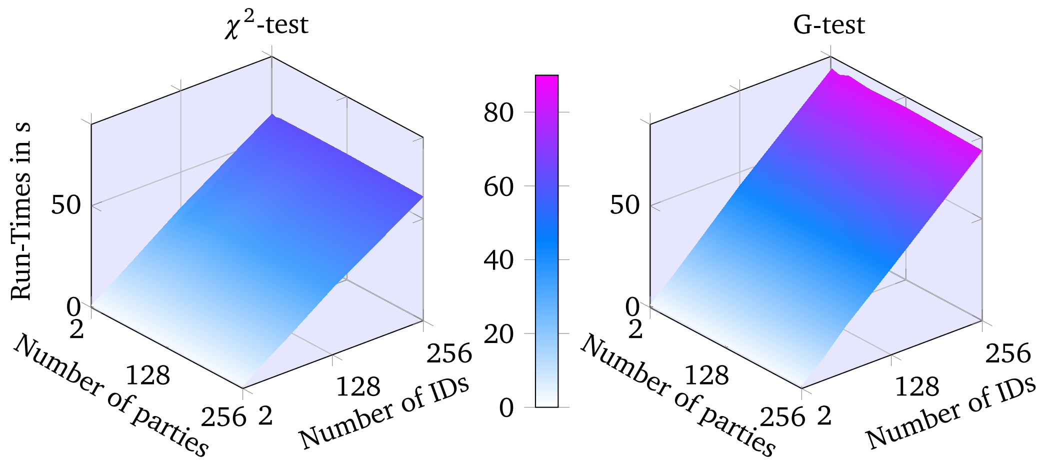


Figure 5.7: Run-times of the extended χ^2- and G-test on 64 SNPs for multiple parties and IDs in the LAN setting in the outsourcing scenario.

In Figure 5.7, we show the run-times of the extended χ^2- and G-test on 64 SNPs from 2 to 256 collaborating institutions and from 2 to 256 number of codewords. As it can be seen in

the figure, the overhead for the extended protocol grows in the number of codewords, but remains constant in the number of parties.

5.5 Communication

In Figure 5.8, we provide the communication requirements for all of the proposed algorithms. Since for multiple parties we aggregate the information in additive sharing (which is for free), and other sharings are suboptimal for this task, we do not detail the communication requirements for >2 parties.

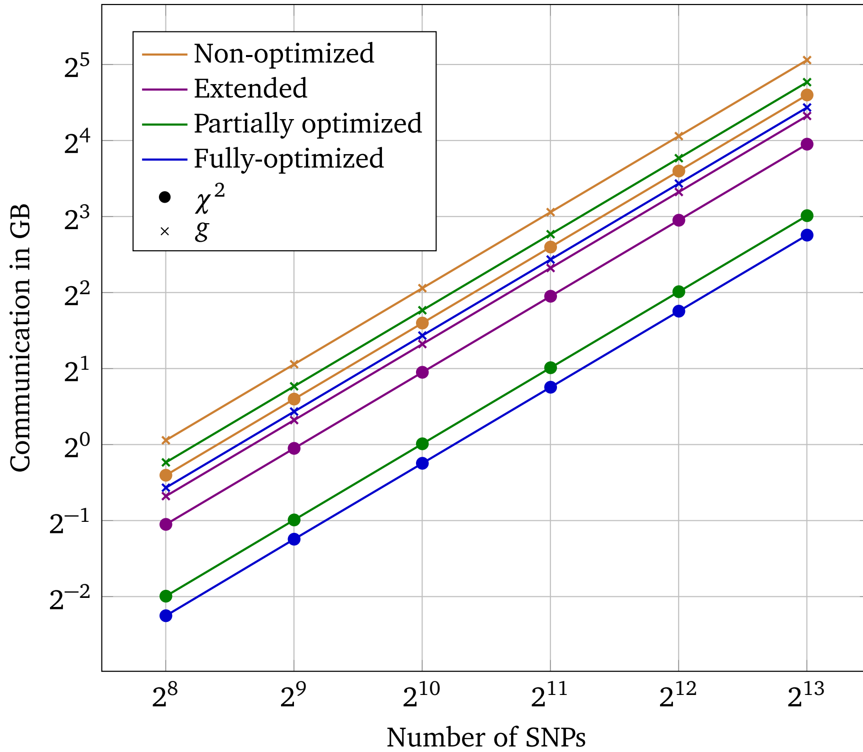


Figure 5.8: Communication of our χ^2- and G-test algorithms depending on number of SNPs.

The amortized communication costs for performing a test on one SNP are shown in Table 5.6. Communication is almost equally distributed among parties. Hence, for finding communication requirements for one party, the values from Table 5.6 have to be divided by 2.

In Table 5.7, we show the amortized communication costs for the extended χ^2- and G-test for multiple IDs. As it can be seen in the table, the algorithms scale linearly in the number of IDs.

Table 5.6: Total amortized communication of our χ^2- and G-test algorithms for one SNP in megabytes.

	χ^2	χ^2_e	χ^2_p	χ^2_f	g	g_e	g_p	g_f	p
Communication	2.95	1.88	0.98	0.82	4.06	2.44	3.32	2.63	0.05

Table 5.7: Total amortized communication of our extended χ^2- and G-test algorithms for multiple IDs for one SNP in megabytes.

# IDs	2	4	8	16	32	64	128	256
χ^2_e	1.6	3	5.9	11	23	46	92	184
g_e	2.3	4.6	9.1	18	36	71	143	286

6 Conclusion

In this chapter we summarize the results of this work and give an outlook for possible future work.

6.1 Summary and Discussion

In the recent years there were many attempts to design algorithms for privacy-preserving distributed GWAS. They were not successful though in terms of practicality, because they either reduced the utility of the results by adding noise or were too inefficient.

Using and extending the state-of-the-art SMPC framework ABY [DSZ15] enabled us to construct and implement very efficient algorithms for χ^2-, G-, and P-tests that outperform the best previous work by up to factor 35x.

In addition, we implemented a more realistic version of the protocols that operates on codeword counts instead of counts of only two alleles. This prevents information losses because of the dimension reduction in the preprocessing step of the protocols.

Finally, we considered an outsourcing scenario where multiple medical institutions privately share their data and conduct statistical tests on it. More specifically, the secret-shared data is securely transmitted via TLS to two Semi-Trusted Third Parties (STTPs) that perform the computation in a low-latency network,e.g. two data centers in the same local network or directly connected via a high-speed fiber-glass channel, and also return the results in secret-shared form. It turns out that the run-time of the optimized protocols only slightly depends on the number of involved parties, thereby making the approach scalable for a large number of participants.

All protocols are very efficient (especially in the LAN setting) in terms of run-time, which does not depend on the number of participants, i.e. observations. For example, it takes 4 ms to compute the fully optimized χ^2-test, 12.5 ms to compute the fully optimized G-test, and 0.5 ms overhead to compute the P-test on one SNP.

6.2 Future Work

The privacy-preserving GWAS protocols can be improved in many ways. We describe our ideas for possible future improvements below:

- **Security against malicious adversaries.** In this thesis, we consider only passive adversaries, which is commonly done before designing protocols that consider also active adversaries. The passive model is not realistic for real-world applications, because in the real world, adversaries can actively interfere in protocols and will not conform to rules. However, a semi-honest adversary protocol can be extended to malicious at a moderate cost.
- **Implementation of other statistical tests.** GWAS can be performed based on a variety of statistics. First of all, adding new statistical tests, e.g. Student's t-test [QCA+14] or Fisher's exact text [ABV+15], to the SMPC suite would contribute to analyzing the complexity of statistical tests in SMPC, and would allow to use the tests in other application scenarios. In addition, one could possibly find tests that better fit to today's GWAS and/or are more efficient in SMPC.
- **Extension to n parties in SMPC.** Our implementation assumes only two non-colluding parties. An extension to n parties [BNP08; CHK+12; AFL+16; FLNW17] allows more institutions to participate in privacy-preserving GWAS. This extension can also be applied to STTPs for the outsourcing scenario in this thesis.

List of Figures

2.1 DNA sequence illustration. . . . 6
2.2 1-out-of-2 Oblivious Transfer protocol. . . . 12
2.3 Outsourcing computation scheme for computing a statistic **S** on aggregate data received from multiple parties. . . . 19

4.1 XOR-chain for computing $(2^k - 1) \mapsto 2^k$. . . . 26
4.2 8-bit logical barrel right shifter [PSW02]. . . . 26
4.3 Integer to floating point number conversion and optimizations 28

5.1 Run-times of the χ^2-test algorithms in ABY in the LAN setting. . . . 42
5.2 Run-times of the G-test algorithms in ABY in the LAN setting. . . . 43
5.3 Comparison of the run-times of our χ^2-test algorithms with those of [CTW+15] in the LAN setting. . . . 44
5.4 P-test run-time overhead on 2^{13}=8 192 SNPs based on the run-times of the underlying statistic with and without the P-test. . . . 46
5.5 Run-times of the χ^2-test algorithms in ABY in the WAN setting. . . . 47
5.6 Run-times of our χ^2- and G-test algorithms on 256 SNPs for multiple parties in the LAN setting in the outsourcing scenario. . . . 50
5.7 Run-times of the extended χ^2- and G-test on 64 SNPs for multiple parties and IDs in the LAN setting in the outsourcing scenario. . . . 50
5.8 Communication of our χ^2- and G-test algorithms depending on number of SNPs. 51

List of Tables

2.1 SNP distribution table. 7

4.1 Integer to floating point number conversion and optimizations. 28
4.2 Contingency table for the extended χ^2-test. 39

5.1 Run-times for the χ^2-, G- and P-test algorithms in ABY in the LAN setting in seconds. 43
5.2 Comparison of the run-times of our χ^2-test algorithms with those of [CTW+15] in the LAN setting in seconds. 45
5.3 P-test run-time overhead on 2^{13}=8 192 SNPs based on the run-times of the underlying statistic in seconds. 45
5.4 Run-times in milliseconds for the local generation of outsourced inputs in Arithmetic (A) and Boolean (B) sharings including instantiation time of the ABY routine. 48
5.5 Run-times of χ^2- and G-test on 256 SNPs for multiple parties in the LAN setting in seconds. 49
5.6 Total amortized communication of our χ^2- and G-test algorithms for one SNP in megabytes. 52
5.7 Total amortized communication of our extended χ^2- and G-test algorithms for multiple IDs for one SNP in megabytes. 52

List of Abbreviations

A Adenine

BC Boolean Circuit

C Cytosine

C-OT Correlated Oblivious Transfer

CDF Cumulative Distribution Function

DNA Deoxyribonucleic Acid

G Guanine

GC Garbled Circuit

GMW Goldreich-Micali-Wigderson

GWAS Genome-Wide Association Studies

HE Homomorphic Encryption

iDASH Integrating Data for Analysis, Anonymization and SHaring

IEEE Institute of Electrical and Electronics Engineers

LAN Local Area Network

MAF Minor Allele Frequency

MUX Multiplexer

OT Oblivious Transfer

PSI Private Set Intersection

R-OT Random Oblivious Transfer

RAM Random Access Memory

SGX Software Guard Extensions

SIMD Single Instruction Multiple Data

SMPC Secure Multi-Party Computation

SNP Single-Nucleotide Polymorphism

SSD Solid-State Drive

STPC Secure Two-Party Computations

STTP Semi-Trusted Third Party

T Thymine

TLS Transport Layer Security

USD US-Dollars

WAN Wide Area Network

XOR Exclusive OR

χ^2 Chi-squared

Bibliography

[ABL+04] M. Atallah, M. Bykova, J. Li, K. Frikken, and M. Topkara, **"Private collaborative forecasting and benchmarking"**, in *WPES*, 2004 (cit. on p. 14).

[ABV+15] F. Aminkeng, A. P. Bhavsar, H. Visscher, S. R. Rassekh, Y. Li, J. W. Lee, L. R. Brunham, H. N. Caron, E. C. van Dalen, L. C. Kremer, *et al.*, **"A coding variant in RARG confers susceptibility to anthracycline-induced cardiotoxicity in childhood cancer"**, *Nature Genetics*, 2015 (cit. on p. 54).

[ABZS13] M. Aliasgari, M. Blanton, Y. Zhang, and A. Steele, **"Secure computation on floating point numbers"**, in *NDSS*, 2013 (cit. on pp. 23 sq.).

[AFL+16] T. Araki, J. Furukawa, Y. Lindell, A. Nof, and K. Ohara, **"High-throughput semi-honest secure three-party computation with an honest majority"**, in *CCS*, 2016 (cit. on p. 54).

[ALSZ13] G. Asharov, Y. Lindell, T. Schneider, and M. Zohner, **"More efficient oblivious transfer and extensions for faster secure computation"**, in *CCS*, 2013 (cit. on pp. 12 sqq.).

[Ant17] J. Antonakis, **"On doing better science: from thrill of discovery to policy implications"**, *The Leadership Quarterly*, 2017 (cit. on p. 8).

[BBB+16] R. Bahmani, M. Barbosa, F. Brasser, B. Portela, A.-R. Sadeghi, G. Scerri, and B. Warinschi, **"Secure multiparty computation from SGX"**, *FC*, 2016 (cit. on p. 22).

[BCGW12] G. S. Barsh, G. P. Copenhaver, G. Gibson, and S. M. Williams, **"Guidelines for genome-wide association studies"**, *PLoS Genet*, 2012 (cit. on p. 8).

[Bea96] D. Beaver, **"Correlated pseudorandomness and the complexity of private computations"**, in *STOC*, 1996 (cit. on pp. 13 sq.).

[BFK+09] M. Barni, P. Failla, V. Kolesnikov, R. Lazzeretti, A.-R. Sadeghi, and T. Schneider, **"Secure evaluation of private linear branching programs with medical applications"**, in *ESORICS*, 2009 (cit. on p. 17).

[BHKR13] M. Bellare, V. T. Hoang, S. Keelveedhi, and P. Rogaway, **"Efficient garbling from a fixed-key blockcipher"**, in *S&P*, 2013 (cit. on p. 16).

[BJL12] D. Bogdanov, R. Jagomägis, and S. Laur, **"A universal toolkit for cryptographically secure privacy-preserving data mining"**, in *PAISI*, 2012 (cit. on p. 16).

[BLLN13] J. W. Bos, K. E. Lauter, J. Loftus, and M. Naehrig, **"Improved security for a ring-based fully homomorphic encryption scheme"**, in *IMACC*, 2013 (cit. on p. 21).

[BLW08] D. Bogdanov, S. Laur, and J. Willemson, **"Sharemind: a framework for fast privacy-preserving computations"**, *ESORICS*, 2008 (cit. on p. 24).

[BNP08] A. Ben-David, N. Nisan, and B. Pinkas, **"FairplayMP: a system for secure multi-party computation"**, in *CCS*, 2008 (cit. on p. 54).

[BP08] J. Boyar and R. Peralta, **"Tight bounds for the multiplicative complexity of symmetric functions"**, *TCS*, 2008 (cit. on p. 26).

[BPSW07] J. Brickell, D. E. Porter, V. Shmatikov, and E. Witchel, **"Privacy-preserving remote diagnostics"**, in *CCS*, 2007 (cit. on p. 17).

[Car78] R. Carver, **"The case against statistical significance testing"**, *Harvard Educational Review*, 1978 (cit. on p. 8).

[CD10] O. Catrina and S. De Hoogh, **"Improved primitives for secure multiparty integer computation"**, in *SCN*, 2010 (cit. on p. 18).

[CDC+16] M. Chiesa, D. Demmler, M. Canini, M. Schapira, and T. Schneider, **"Towards securing internet exchange points against curious onlookers"**, in *ANRW*, 2016 (cit. on p. 18).

[CDD+16] F. Chen, M. Dow, S. Ding, Y. Lu, X. Jiang, H. Tang, and S. Wang, **"PREMIX: privacy-preserving estimation of individual admixture"**, in *AMIA*, 2016 (cit. on p. 22).

[CHK+12] S. G. Choi, K.-W. Hwang, J. Katz, T. Malkin, and D. Rubenstein, **"Secure multi-party computation of boolean circuits with applications to privacy in on-line marketplaces"**, in *CT-RSA*, 2012 (cit. on p. 54).

[CHW+15] R. Cai, Z. Hao, M. Winslett, X. Xiao, Y. Yang, Z. Zhang, and S. Zhou, **"Deterministic identification of specific individuals from GWAS results"**, *Bioinformatics*, 2015 (cit. on p. 20).

[CTW+15] S. D. Constable, Y. Tang, S. Wang, X. Jiang, and S. Chapin, **"Privacy-preserving GWAS analysis on federated genomic datasets"**, *BMC Medical Informatics and Decision Making*, 2015 (cit. on pp. 1 sqq., 21 sq., 29, 43 sqq.).

[DDK+15] D. Demmler, G. Dessouky, F. Koushanfar, A.-R. Sadeghi, T. Schneider, and S. Zeitouni, **"Automated synthesis of optimized circuits for secure computation"**, in *CCS*, 2015 (cit. on pp. 13, 23).

[DGK08] I. Damgard, M. Geisler, and M. Kroigard, **"Homomorphic encryption and secure comparison"**, *IJACT*, 2008 (cit. on pp. 13 sq.).

[DJN10] I. Damgård, M. Jurik, and J. B. Nielsen, **"A generalization of Paillier's public-key system with applications to electronic voting"**, *IJISS*, 2010 (cit. on pp. 13 sq.).

[DMNS06] C. Dwork, F. McSherry, K. Nissim, and A. Smith, **“Calibrating noise to sensitivity in private data analysis”**, in *TCC*, 2006 (cit. on p. 1).

[DSZ15] D. Demmler, T. Schneider, and M. Zohner, **“ABY-a framework for efficient mixed-protocol secure two-party computation”**, in *NDSS*, 2015 (cit. on pp. 2, 4, 13 sqq., 17 sq., 27, 53).

[Fis25] R. A. Fisher, **“Statistical methods for research workers”**. Genesis Publishing Pvt Ltd, 1925 (cit. on p. 8).

[FLNW17] J. Furukawa, Y. Lindell, A. Nof, and O. Weinstein, **“High-throughput secure three-party computation for malicious adversaries and an honest majority”**, in *EUROCRYPT*, 2017 (cit. on p. 54).

[FSU11] S. E. Fienberg, A. Slavkovic, and C. Uhler, **“Privacy preserving GWAS data sharing”**, in *ICDMW*, 2011 (cit. on p. 20).

[GHS12] C. Gentry, S. Halevi, and N. P. Smart, **“Homomorphic evaluation of the AES circuit”**, in *CRYPTO*, 2012 (cit. on p. 21).

[Gif14] D. K. Gifford. (2014). **“Foundations of computational and systems biology”**, [Online]. Available: `https://ocw.mit.edu/courses/biology/7-91j-foundations-of-computational-and-systems-biology-spring-2014/lecture-slides/MIT7_91JS14_Lecture20.pdf` (visited on 05/14/2017) (cit. on p. 30).

[Gil99] N. Gilboa, **“Two party RSA key generation”**, in *CRYPTO*, 1999 (cit. on p. 14).

[GMW87] O. Goldreich, S. Micali, and A. Wigderson, **“How to play any mental game, or a completeness theorem for protocols with an honest majority”**, in *STOC*, 1987 (cit. on p. 15).

[GWA+12] A. Gutmann, J. Wagner, Y. Ali, A. Allen, J. Arras, B. Atkinson, N. Farahany, A. Garza, C. Grady, S. Hauser, *et al.*, **“Privacy and progress in whole genome sequencing”**, *Presidential Committee for the Study of Bioethical Issues*, 2012 (cit. on p. 1).

[Hoe12] J. Hoey, **“The two-way likelihood ratio (G) test and comparison to two-way Chi squared test”**, *ArXiv:1206.4881*, 2012 (cit. on p. 10).

[HSR+08] N. Homer, S. Szelinger, M. Redman, D. Duggan, W. Tembe, J. Muehling, J. V. Pearson, D. A. Stephan, S. F. Nelson, and D. W. Craig, **“Resolving individuals contributing trace amounts of DNA to highly complex mixtures using high-density SNP genotyping microarrays”**, *PLoS Genet*, 2008 (cit. on p. 20).

[HSS+10] W. Henecka, A.-R. Sadeghi, T. Schneider, I. Wehrenberg, *et al.*, **“TASTY: tool for automating secure two-party computations”**, in *CCS*, 2010 (cit. on pp. 17, 23).

[IEE08] IEEE Computer Society, **“IEEE standard for floating-point arithmetic”**, *IEEE Std 754-2008*, 2008 (cit. on p. 26).

[IKNP03] Y. Ishai, J. Kilian, K. Nissim, and E. Petrank, **“Extending oblivious transfers efficiently”**, in *CRYPTO*, 2003 (cit. on pp. 13 sq.).

[Ioa05] J. P. Ioannidis, **“Why most published research findings are false”**, *PLoS Med.*, 2005 (cit. on p. 8).

[IR89] R. Impagliazzo and S. Rudich, **“Limits on the provable consequences of one-way permutations”**, in *STOC*, 1989 (cit. on p. 12).

[JS13] A. Johnson and V. Shmatikov, **“Privacy-preserving data exploration in genome-wide association studies”**, in *SIGKDD*, 2013 (cit. on p. 20).

[JZW+14] X. Jiang, Y. Zhao, X. Wang, B. Malin, S. Wang, L. Ohno-Machado, and H. Tang, **“A community assessment of privacy preserving techniques for human genomes”**, *BMC Medical Informatics and Decision Makingg*, 2014 (cit. on p. 20).

[KL15] M. Kim and K. Lauter, **“Private genome analysis through homomorphic encryption”**, *BMC Medical Informatics and Decision Making*, 2015 (cit. on p. 21).

[KS08] V. Kolesnikov and T. Schneider, **“Improved garbled circuit: free XOR gates and applications”**, *ICALP*, 2008 (cit. on p. 16).

[KSMB13] B. Kreuter, A. Shelat, B. Mood, and K. R. Butler, **“PCF: a Portable Circuit Format for scalable two-party secure computation”**, in *USENIX Security*, 2013 (cit. on p. 22).

[KSS13] V. Kolesnikov, A.-R. Sadeghi, and T. Schneider, **“A systematic approach to practically efficient general two-party secure function evaluation protocols and their modular design”**, *JCS*, 2013 (cit. on p. 17).

[KSS14] F. Kerschbaum, T. Schneider, and A. Schröpfer, **“Automatic protocol selection in secure two-party computations”**, in *ACNS*, 2014 (cit. on p. 14).

[KW14] T. Krips and J. Willemson, **“Hybrid model of fixed and floating point numbers in secure multiparty computations”**, in *ISC*, 2014 (cit. on p. 23).

[LDD+16] X. Liu, R. H. Deng, W. Ding, R. Lu, and B. Qin, **“Privacy-preserving outsourced calculation on floating point numbers”**, *IEEE TIFS*, 2016 (cit. on p. 23).

[LYS15] W.-J. Lu, Y. Yamada, and J. Sakuma, **“Privacy-preserving genome-wide association studies on cloud environment using fully homomorphic encryption”**, *BMC Medical Informatics and Decision Making*, 2015 (cit. on p. 21).

[Mat17] MathWorks. (2017). **“Chi-square inverse cumulative distribution function”**, [Online]. Available: `http://mathworks.com/help/stats/chi2inv.html` (visited on 08/21/2017) (cit. on p. 9).

[McD09] J. H. McDonald, **“Handbook of biological statistics”**. Sparky House Publishing Baltimore, MD, 2009 (cit. on p. 10).

[MKGV07] A. Machanavajjhala, D. Kifer, J. Gehrke, and M. Venkitasubramaniam, **“l-diversity: privacy beyond k-anonymity”**, *TKDD*, 2007 (cit. on p. 1).

[MNP+04] D. Malkhi, N. Nisan, B. Pinkas, Y. Sella, *et al.*, **“Fairplay-secure two-party computation system”**, in *USENIX Security*, 2004 (cit. on pp. 16 sq.).

[NAC+14] M. Naveed, E. Ayday, E. W. Clayton, J. Fellay, C. A. Gunter, J.-P. Hubaux, B. Malin, X. Wang, *et al.*, **“Privacy and security in the genomic era”**, *CCS*, 2014 (cit. on p. 1).

[PBS12] P. Pullonen, D. Bogdanov, and T. Schneider, **“The design and implementation of a two-party protocol suite for Sharemind 3”**, *CYBERNETICA Institute of Information Security, Tech. Rep*, 2012 (cit. on p. 14).

[PS15] P. Pullonen and S. Siim, **“Combining secret sharing and garbled circuits for efficient private IEEE 754 floating-point computations”**, in *FC*, 2015 (cit. on p. 23).

[PSSZ15] B. Pinkas, T. Schneider, G. Segev, and M. Zohner, **“Phasing: private set intersection using permutation-based hashing”**, in *USENIX Security*, 2015 (cit. on p. 13).

[PSW02] M. R. Pillmeier, M. J. Schulte, and E. G. Walters III, **“Design alternatives for barrel shifters”**, in *SPIE*, 2002 (cit. on pp. 25 sq.).

[QCA+14] E. E. Quillen, X.-D. Chen, L. Almasy, F. Yang, H. He, X. Li, X.-Y. Wang, T.-Q. Liu, W. Hao, H.-W. Deng, *et al.*, **“ALDH2 is associated to alcohol dependence and is the major genetic determinant of “daily maximum drinks” in a GWAS study of an isolated rural Chinese sample”**, *American Journal of Medical Genetics Part B: Neuropsychiatric Genetics*, 2014 (cit. on p. 54).

[Rab81] M. O. Rabin, **“How to exchange secrets with oblivious transfer”**, *Technical Report TR-81*, 1981 (cit. on p. 12).

[Rap04] D. K. Rappe. (2004). **“Homomorphic cryptosystems and their application”**, (visited on 07/15/2017) (cit. on p. 21).

[RK17] R. Rogers and D. Kifer, **“A new class of private Chi-square hypothesis tests”**, in *AISTATS*, 2017 (cit. on pp. 1, 20).

[RMG12] R. Rieger, A. Michaelis, and M. M. Green, **“Glossary of genetics and cytogenetics: classical and molecular”**. Springer Science & Business Media, 2012 (cit. on p. 6).

[SAW13] L. Sweeney, A. Abu, and J. Winn, **“Identifying participants in the personal genome project by name”**, *Data Privacy Lab, IQSS*, 2013 (cit. on p. 1).

[SB16] S. Simmons and B. Berger, **“Realizing privacy preserving genome-wide association studies”**, *Bioinformatics*, 2016 (cit. on p. 20).

[SSDM09] C. C. Spencer, Z. Su, P. Donnelly, and J. Marchini, **“Designing genome-wide association studies: sample size, power, imputation, and the choice of genotyping chip”**, *PLoS Genet*, 2009 (cit. on p. 1).

[Swe02] L. Sweeney, **“k-anonymity: a model for protecting privacy”**, *International Journal of Uncertainty, Fuzziness and Knowledge-Based Systems*, 2002 (cit. on p. 1).

[SZ13] T. Schneider and M. Zohner, **"GMW vs. Yao? Efficient secure two-party computation with low depth circuits"**, in *FC*, 2013 (cit. on p. 16).

[USF13] C. Uhler, A. Slavković, and S. E. Fienberg, **"Privacy-preserving data sharing for genome-wide association studies"**, *JPC*, 2013 (cit. on p. 20).

[Vol13] H. Vollmer, **"Introduction to circuit complexity: a uniform approach"**. Springer Science & Business Media, 2013 (cit. on p. 15).

[VSJO13] J. Vaidya, B. Shafiq, X. Jiang, and L. Ohno-Machado, **"Identifying inference attacks against healthcare data repositories"**, *AMIA Summits on Translational Science*, 2013 (cit. on p. 1).

[WLW+09] R. Wang, Y. F. Li, X. Wang, H. Tang, and X. Zhou, **"Learning your identity and disease from research papers: information leaks in genome wide association study"**, in *CCS*, 2009 (cit. on p. 20).

[Woo15] C. Woolston. (2015). **"Psychology journal bans P values"**, [Online]. Available: `https://www.nature.com/news/psychology-journal-bans-p-values-1.17001` (visited on 05/24/2017) (cit. on p. 8).

[WW16] A. Weintraub and I. Wünnenberg, **"Das Genom ist nicht genug"**, *Technology Review*, 2016 (cit. on p. 1).

[Yao86] A. C.-C. Yao, **"How to generate and exchange secrets"**, in *FOCS*, 1986 (cit. on p. 16).

[YFSU14] F. Yu, S. E. Fienberg, A. B. Slavković, and C. Uhler, **"Scalable privacy-preserving data sharing methodology for genome-wide association studies"**, *Journal of Biomedical Informatics*, 2014 (cit. on p. 20).

[YJ14] F. Yu and Z. Ji, **"Scalable privacy-preserving data sharing methodology for genome-wide association studies: an application to iDASH healthcare privacy protection challenge"**, *BMC Medical Informatics and Decision Making*, 2014 (cit. on p. 20).

[ZBA15] Y. Zhang, M. Blanton, and G. Almashaqbeh, **"Secure distributed genome analysis for GWAS and sequence comparison computation"**, *BMC Medical Informatics and Decision Making*, 2015 (cit. on p. 22).

[ZDJ+15] Y. Zhang, W. Dai, X. Jiang, H. Xiong, and S. Wang, **"Foresee: fully outsourced secure genome study based on homomorphic encryption"**, *BMC Medical Informatics and Decision Making*, 2015 (cit. on p. 21).

[ZPL+11] X.-y. Zhou, B. Peng, Y. F. Li, Y. Chen, H. Tang, and X. Wang, **"To release or not to release: evaluating information leaks in aggregate human-genome data"**, in *ESORICS*, 2011 (cit. on p. 20).

[ZRE15] S. Zahur, M. Rosulek, and D. Evans, **"Two halves make a whole"**, in *EUROCRYPT*, 2015 (cit. on p. 16).

[ZSB13] Y. Zhang, A. Steele, and M. Blanton, **"PICCO: a general-purpose compiler for private distributed computation"**, in *CCS*, 2013 (cit. on pp. 23 sq.).

[ZWJ+14] Y. Zhao, X. Wang, X. Jiang, L. Ohno-Machado, and H. Tang, **“Choosing blindly but wisely: differentially private solicitation of DNA datasets for disease marker discovery”**, *Journal of the American Medical Informatics Association*, 2014 (cit. on p. 20).