Bioinformatics, 38(8), 2022, 2202–2210
https://doi.org/10.1093/bioinformatics/btac070
Advance Access Publication Date: 12 February 2022
Original Paper

Genetics and population analysis

Efficient privacy-preserving whole-genome variant queries

Mete Akgün [1,2,3,4,*], **Nico Pfeifer**[2,5,6] **and Oliver Kohlbacher** [2,3,7]

[1]Medical Data Privacy and Privacy-Preserving ML on Healthcare Data, Department of Computer Science, University of Tübingen, Tübingen, Germany, [2]Institute for Bioinformatics and Medical Informatics, University of Tübingen, Tübingen, Germany, [3]Translational Bioinformatics, University Hospital Tübingen, Tübingen, Germany, [4]Department of Computer Engineering, İzmir Institute of Technology, İzmir, Turkey, [5]Methods in Medical Informatics, Department of Computer Science, University of Tübingen, Tübingen, Germany, [6]Statistical Learning in Computational Biology, Max Planck Institute for Informatics, Saarbrücken, Germany and [7]Applied Bioinformatics, Department of Computer Science, University of Tübingen, Tübingen, Germany

*To whom correspondence should be addressed.

Associate Editor: Russell Schwartz

Received on October 6, 2021; revised on January 13, 2022; editorial decision on January 17, 2022; accepted on February 3, 2022

Abstract

Motivation: Diagnosis and treatment decisions on genomic data have become widespread as the cost of genome sequencing decreases gradually. In this context, disease–gene association studies are of great importance. However, genomic data are very sensitive when compared to other data types and contains information about individuals and their relatives. Many studies have shown that this information can be obtained from the query-response pairs on genomic databases. In this work, we propose a method that uses secure multi-party computation to query genomic databases in a privacy-protected manner. The proposed solution privately outsources genomic data from arbitrarily many sources to the two non-colluding proxies and allows genomic databases to be safely stored in semi-honest cloud environments. It provides data privacy, query privacy and output privacy by using XOR-based sharing and unlike previous solutions, it allows queries to run efficiently on hundreds of thousands of genomic data.

Results: We measure the performance of our solution with parameters similar to real-world applications. It is possible to query a genomic database with 3 000 000 variants with five genomic query predicates under 400 ms. Querying 1 048 576 genomes, each containing 1 000 000 variants, for the presence of five different query variants can be achieved approximately in 6 min with a small amount of dedicated hardware and connectivity. These execution times are in the right range to enable real-world applications in medical research and healthcare. Unlike previous studies, it is possible to query multiple databases with response times fast enough for practical application. To the best of our knowledge, this is the first solution that provides this performance for querying large-scale genomic data.

Availability and implementation: https://gitlab.com/DIFUTURE/privacy-preserving-variant-queries.

Contact: mete.akguen@uni-tuebingen.de

Supplementary information: Supplementary data are available at *Bioinformatics* online.

1 Introduction

As genome sequencing technologies become faster and more efficient, genome are increasingly used in a variety of areas including direct-to-consumer services (23andMe, Ancestry, Gene By Gene, MyHeritage) (Khan and Mittelman, 2018), forensic medicine (Børsting and Morling, 2016), personalized medicine (Amendola *et al.*, 2018; Deng and Nakamura, 2017; Sürün *et al.*, 2020) and, genome-disease association studies (Consortium *et al.*, 2007; Hyde *et al.*, 2016). For example, it is now possible for doctors to give the right drug at the right time (for some drugs) by examining the patient's genome (Akgün *et al.*, 2015; Erlich and Narayanan, 2014; Naveed *et al.*, 2015). Having access to genome data for individual patients has paved the way for individualized therapies, especially in oncology. Many applications in genomic medicine call for a comparison of genomes either locally ('who else has this variant in a particular gene?') or on a global scale ('are there patients with a similar genome?'). While these queries are fundamentally trivial, the sensitive nature of genomic data implies constraints on how such queries can be implemented. Since genomic data can be used to distinguish two individuals from one another and to obtain genotype, phenotype and ancestry (Naveed *et al.*, 2015), disclosure of this data results in different privacy vulnerabilities comparing the disclosure of other data. Once genomic information has been disclosed to third parties, there is no way to undo that damage (in contrast, for example, to stolen credit card numbers, one cannot change one's genome). Consequently, access and use of genomic data are strictly regulated.

One way to solve the aforementioned privacy problem is to implement data sharing policies and consent processes. Unfortunately, it is very difficult and time consuming to implement these solutions because of the different regulations of the institutions. In addition, privacy violations that may occur in the future might be overlooked (Aziz *et al.*, 2017). For privacy-preserving sharing of genomic data, the Global Alliance for Genomics and Health (GA4GH) launched the Beacon network (Global Alliance for Genomics and Health, 2016), where distributed genomic data can be queried over the web. The Beacon network enables queries whether there is a specific variant present in any of the genomes stored at one of the Beacon nodes and the system responds with 'Yes' or 'No'. Although the Beacon network is designed to share individuals' genomic data in a privacy-protected way, Shringarpure and Bustamante (2015) have shown that re-identification attacks can be carried out with a small fraction of a target genome. Methods that can prevent this attack but also reduce the use of the system are provided in Raisaro *et al.* (2017). von Thenen *et al.* (2019) improved the original attack by reducing the number of queries required to determine the presence of an individual in a beacon.

In Demmler *et al.* (2017), the authors proposed a solution based on secure multi-party computation (MPC) for private and secure federated variant queries. This solution was proposed as an alternative to the Beacon network. Unlike that Beacon network, it hides which institution contributes to the answer and applied a threshold value to the answer. In this solution, data from multiple sources is being sent to the two proxy servers. It is assumed that the proxy servers do not collude with each other. The knowledge of which database contributes to the output and which variants are accessed is unknown in this solution. For this reason, data privacy, query privacy and output privacy are provided. The performance of this solution when working on multiple patient data is not sufficient. The execution times of the protocol increase linearly with the increase in the number of variants, query variants and patients. Thus, we can calculate the execution time of the protocol for a given number of variants, query variants and patients. Querying 1 000 000 genomes with 3 000 000 variants each for the presence of five different query variants take approximately 748 333 h and querying a single genome with 3 000 000 variants for the presence of 1000 different query variants take approximately 150 h. High performance can be achieved with a dedicated system, high memory capacity and a high number of processors.

1.1 Our contributions

In this article, we propose a method by which we can perform queries without violating the privacy of individuals on the genomic data using secure MPC. We adapt MPC to transfer genomic data from multiple sources to the two proxy servers (Kamara and Raykova, 2011) and to run researchers' queries privately on these proxy servers. We solve the security and privacy problems caused by outsourcing of the genomic data to the proxy servers due to sharing and computation. First, we ensure data privacy. Security of genomic data stored on the proxy servers is provided by evaluating the entire computation process. Genomic data are secured even if one of the proxy servers is compromised. Second, we provide query privacy. The proxy servers do not learn anything about queries being executed. Third, we provide privacy to the query output. The query output is only learned by the researcher. Unlike the Beacon network (Global Alliance for Genomics and Health, 2016), our solution hides the contribution of each data owner (medical institute or hospital) to the query output. The attacker can perform the re-identification attack (Shringarpure and Bustamante, 2015) but cannot know in which medical institution the victim's genome is stored.

The proposed solution establishes a querying method that paves the way for an efficient privacy-preserving system for secure genome queries. We express each variant database in a binary balanced tree. The queries are executed by traversing a tree representing variants. At the end of this process, the servers reach variant values in leaf node and compare them with the query values using secure MPC. Unlike previous studies (Demmler *et al.*, 2017; Hasan *et al.*, 2018), which had a drawback that they cannot answer queries on databases having thousands of patient's data in a reasonable time, our solution responds to such queries in a very short time. Query performance of our protocol is constant with respect to the number of variants due to the use of tree structures to represent variants and linear with respect to the number of patients and query variants. Our solution offers unprecedented performance for secure variant queries by taking the advantage of the tree-based protocol.

We prove our method to be secure in the presence of honest-but-curious proxy servers and malicious clients. We implement and test our method by considering various numbers of variants, patients and query variants. Our experiments show that in 3.23 h we can query five variants against a database of 1 048 576 patients, each containing 1 000 000 variants in a WAN setting. The query time is reduced to approximately 6 min using a small amount of dedicated hardware. This shows that our method has the ability to work in current clinical and research settings practically and efficiently.

2 Related work

There are many studies on genomic privacy in the literature. Interested readers are referred to some excellent reviews (Akgün *et al.*, 2015; Aziz *et al.*, 2017; Mittos *et al.*, 2019; Naveed *et al.*, 2015). In this section, we provide an overview of recent work related to our solution.

The edit distance (ED) is a measure of the proximity of two strings. It is calculated by finding the minimum number of operations to convert one string to another. In bioinformatics, ED is used to calculate the similarity of DNA sequences that are encoded with letters A, C, G and T. There are many studies that calculate ED on protected genomic databases in a privacy-preserving way. In Jha *et al.* (2008), a privacy-preserving edit distance calculation algorithm based on Yao's garbled circuits was developed. With this solution, the edit distance calculation of several hundred-character-long genome sequences takes several minutes. Thus, applying it to the whole-genome scale is computationally infeasible. Another study (Huang *et al.*, 2011) improved the performance of this solution 29 times.

When the whole genome is evaluated, the genome is 99% similar in all humans, and variations in 1% are mostly simple substitutions. These facts were utilized in two studies (Asharov *et al.*, 2018; Wang *et al.*, 2015). In Wang *et al.* (2015), the authors proposed a distributed query system. They tested the proposed system in 250 hospitals, each with 4000 genomes with length of 75 million nucleotides each. It took 200 min to search for one million cancer patients. In Asharov *et al.* (2018), the authors have studied the same problem. They divide the genome sequences into small blocks and precompute ED between these blocks. With the power of precomputation and their own approximation function, their method is more accurate and faster than Wang *et al.* (2015).

In Salem *et al.* (2019), the authors developed a privacy-preserving protocol for similar patient queries for combined medical databases. The proposed methods work on various types of biomedical data, including genomic, epigenomic and transcriptomic data as well as their combination. The authors use the ED and Pearson correlation coefficient as similarity measures. Their solution securely queries a database of 1000 patients in less than 5 seconds with the ED and in less than thirty seconds with the Pearson correlation coefficient. It performs the similarity computation with weighted ED over 1 million patient data in 14.6 h.

In Sousa *et al.* (2017), the authors proposed a solution that stores genomic data in the cloud and provides a secure querying service. In this solution, Variant Call Format (VCF) files are symmetrically encrypted and then sent to the cloud. Homomorphic encryption and private information retrieval are used to ensure data and query security. Since each VCF file uses a different encryption key, the solution cannot work on multiple patient data. An enhanced version of this system that works on multiple patient data was proposed in Froelicher *et al.* (2017). In this solution, security and privacy features can be predetermined to set query performance. Another factor influencing system performance is output size. It is possible to infer information about the query from the output. This problem can be solved by padding the query.

In Hasan *et al.* (2018), the authors proposed a solution in which the aggregated genomic data was secretly stored and queried privately. The system only supports count queries revealing how many records contained in the database match the given criteria. Aggregated variants are stored using a tree. This tree is stored in the cloud in encrypted form. The queries are executed by traversing the encrypted tree. Traversing decisions are made according to which branch of the current node matches the query. The proposed solution claims to provide data privacy, query privacy and output privacy. This solution can only work on a set of previously known variants and needs a single and trusted certificated institution that knows all genomic data coming from different sources.

3 Materials and methods

In this section, we introduce preliminaries, the proposed privacy-preserving method to query multiple genomic variant databases. We also present the security considerations.

3.1 Secure MPC

Secure MPC is a method that allows parties to compute an arbitrary function over their inputs together in a privacy-protected manner without learning each other's input. The expected security of MPC is that the method does not disclose anything besides the output. MPC was first introduced by Andrew Yao (1986) for two-party secure computation. This method was then generalized by Goldreich, Micali and Widgerson (GMW) (Goldreich *et al.*, 1987) for multiple parties. Due to the high computational cost, MPC has been considered impractical for a long time. Only recently this perception has been changing due to the availability of more efficient MPC algorithms. Most MPC protocols are based on oblivious transfer (OT) as a building block.

MPC calculations may require many interaction rounds and large data conversions between parties. For these reasons, it may be very difficult to implement MPC in practice. Some successful works have been done to reduce the complexity of MPC and to implement it in practice for some problems (Bogdanov *et al.*, 2008; Demmler *et al.*, 2015; Huang *et al.*, 2011; Liu *et al.*, 2015; Malkhi *et al.*, 2004). The schemes proposed in these studies allow us to utilize the benefits of MPC for some real-life applications.

In the rest of the article, we denote a shared value x as $\langle x \rangle$.

3.1.1 Boolean sharing

In Boolean sharing, an l-bit value x is shared using an XOR-based sharing scheme as the XOR of two values. For l-bit secret sharing of x, we have $\langle x \rangle_0 \oplus \langle x \rangle_1 = x$ where $\langle x \rangle_0, \langle x \rangle_1 \in \mathbb{Z}_2$ and I_i knows only $\langle x \rangle_i$ where $i \in \{0, 1\}$. For Boolean sharing, we use the protocol of GMW (Goldreich *et al.*, 1987).

XOR. $\langle z \rangle = \langle x \rangle \oplus \langle y \rangle$. I_i locally computes $\langle x \rangle_i \oplus \langle y \rangle_i$.

AND. $\langle z \rangle = \langle x \rangle \wedge \langle y \rangle$. AND is performed using a pre-computed Boolean multiplication triple $\langle c \rangle_i = \langle a \rangle_i \wedge \langle b \rangle_i$. I_i cannot perform AND locally. A Boolean multiplication triple is pre-computed efficiently using random OT (R-OT) in the offline phase. More details can be found in Asharov *et al.* (2013) and Demmler *et al.* (2015).

3.2 System model

In this article, we propose a scalable solution that allows researchers and clinicians to query genomic variations with privacy protection. This solution protects the privacy of queries, results and genomic data. Figure 1 shows the general architecture of the proposed system. The system has three types of participants: a series of genomics stores/hospitals $H_1, \ldots H_N$ storing the variants from a large number of patients each, two proxy servers D_1 and D_2 and an arbitrary number of researchers (clients). The variants for each patient are stored at H_i in a standardized manner (VCF files). Variants from each patient can then be parsed to construct a single variant tree. Hospitals use XOR-based sharing to transmit their data privately to D_1 and D_2. This is similar to the cloud computing solution that is preferred for efficiently storing and processing growing data. In our settings, we assume that there is no collusion between the two proxy servers. In real-world settings this can be achieved by organizational and physical separation of access to the proxy servers.

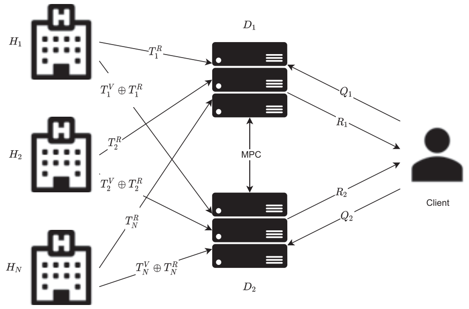

Fig. 1. General system architecture of our solution. Genomic variant stores $H_1 \ldots H_N$ communicate with the two non-colluding proxy servers D_1 and D_2. Users can query all data through these proxy servers in a secure manner

As described in Section 3.4, a hospital H_i creates a tree T_i^V for each VCF file. It creates another tree T_i^R that is isomorphic to T_i^V, but stores random values. It sends T_i^R to the proxy server D_1 and $T_i^V \oplus T_i^R$ to the proxy server D_2. H_i shares the variant tree to the two proxy servers with XOR-based sharing. This ensures that the genomic data are protected against the two non-colluding servers. The queries are run using secure MPC in a privacy-preserving manner. The researcher receives the query result as two shares from the proxy servers and XORes these two shares to get the final answer. As a result, data, query and result privacy are ensured.

In our proposed solution, we construct a Boolean circuit to query the genomic data in a privacy-protected manner. We used the GMW (Goldreich *et al.*, 1987) protocol to run Boolean circuits on XOR shared values.

3.3 Security definition

In our solution, participants in the secure computation are two proxy servers D_1, D_2 and arbitrary number of clients $C_1, C_2, \ldots, C_n$. We consider the problem of secure variant query under the semi-honest adversary model. In this model, semi-honest parties follow the protocol specification but try to learn additional information by analyzing the transcript of messages received during the protocol execution. A semi-honest adversary $\mathcal{A}$ can corrupt at most one of the two proxy servers, and any subset of the clients. This corresponds with the assumption of non-colluding servers. While one server is controlled by the adversary and the other behaves honestly.

Our protocol securely queries variant databases and returns the result to query client. It provides data privacy, query privacy and output privacy and protect the data access patterns. We list desired privacy properties:

- D_1 and D_2 learn nothing about the variant databases.
- D_1 and D_2 learn nothing about the client queries.
- D_1 and D_2 learn nothing about query output.
- D_1 and D_2 do not learn the data access patterns such as the path of query variants in the variant trees.
- C_i learn nothing about the variant databases except the final output.

3.4 Database outsourcing

In this section, we explain how hospitals outsource their genomic data to the two proxy servers.

3.4.1 Variant representation

We encode each variant as a bit string using data columns in VCF files. In a VCF file, the variant is expressed by REF and ALT columns representing reference allele and alternate non-reference allele respectively. The REF column shows the reference bases replaced by the non-reference allele in the ALT column. CHROM column represents the chromosome on which the variant is and POS column shows the position of the variant on the chromosome. The position of the variant on the reference genome is calculated using these two pieces of information. This information is expressed as a 32-bit unsigned integer. We map the CHR, POS, ALT, REF and 32-bit location values of each variant in the VCF file to a fixed number of bits by hashing. This value is expressed as 48 bits in order to encode SNPs, insertions, deletions and other variant types. The encoding of a single variant with 48-bit is illustrated in Figure 2.

3.4.2 Secure outsourcing of genomic data

We describe how a patient data (VCF file) is outsourced to the two proxy servers. This process is done for each patient. A medical unit H_i creates two trees for each patient's variant data as described in Algorithm 2.H_i creates a binary tree for each patient's genome data. The root node of the tree does not store a bit value. The depth of the tree is determined based on the number of variants. If we assume a patient can have at most N variants, the depth of trees is $\lceil \log N \rceil$. The bits represented by the two child nodes of each node are determined randomly. This means if the right child node represents a random bit b, the left child node represents $b \oplus 1$. The remaining $48 - \lceil \log N \rceil$ bits of each variant are kept in leaf nodes. In the system, F random variant masks are determined. These masks are publicly known by every party. Each variant of the patient is XORed with each mask. Masked variants are also added to the variant tree. Each leaf node represents at most $f = F + 1$ variants. Leaf nodes having less than $F + 1$ variants are completed with randomly chosen dummy variants. Each variant tree has a marker bit x. x is added to the end of each real variant. $x \oplus 1$ is added to the end of each dummy variant. Thus, real and dummy variants are distinguished from each other. An example of how to build a tree is shown in Figure 3. In this example, we have three random masks and five variants which are—for simplicity's sake—expressed by five bits each and the marker bit is not added to the variants.

H_i then constructs a second tree T_i^R which is isomorphic to T_i^V and assign uniformly distributed binary random bits to its nodes and random values to its leaves. H_i sends T_i^R to D_1 and $T_i^R \oplus T_i^V$ to D_2. H_i also divides the marker bit x_i of T_i^V into two shares and sends them to the proxy servers. D_1 and D_2 record the source of each transferred tree.

3.5 Privacy-preserving querying

In this section, we provide an MPC-based efficient privacy-preserving method to query genomic databases with the received query.

3.5.1 Query generation

The client maps each variant in the query to 48-bit unsigned integer using the MapVariant() method described in Algorithm 1. It generates an arbitrary number of dummy variants. It randomly chooses masks from the set of known masks as many as the number of variants it has (including dummy variants). It XORes query variants with the chosen masks. It concatenates all 48-bit unsigned integers to a bit array Q. In our example, there are four real variants and four dummy variants. The length of Q becomes $8 \cdot 48 = 384$. The client creates another bit array Q_R of the same length filled with random bit values and divides its query into two shares to both proxy servers via XOR-based sharing. It sends Q_R to the proxy server D_1 and $Q \oplus Q_R$ to the proxy server D_2. It also generates a bit vector B that shows whether variants are dummy. Dummy variants are represented by one and real variants are represented by zero. The client creates another bit array B_R having the same length with B and divides B into two shares via XOR-based sharing. It sends B_R to the proxy server D_1 and $B \oplus B_R$ to the proxy server D_2. Dummy variants provide the following benefits.

- The proxy servers do not know the exact number of variants in the client query.
- The proxy servers cannot differentiate two queries consisting of same variants.
- Dummy variants obfuscate the statistics such as the number of times a variant was queried.

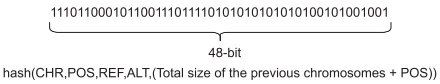
111011000101100111011110101010101010100101001001
48-bit
hash(CHR,POS,REF,ALT,(Total size of the previous chromosomes + POS))

Fig. 2. Encoding of a single variant

Algorithm 1: Mapping of a Variant to Unsigned Integer MapVariant()

```
input : $v$, $v$ is a variant in a patient's VCF file
$len \leftarrow calc\_chrs\_len(v.CHR)$   /* the size of all chromosomes are known, the total length of previous chromosomes is calculated */
$pos \leftarrow v.POS + len$   /* calculate global position of a variant on the genome */
$res \leftarrow HASH(CHR||POS||ALT||REF||pos)$
return $res$
```

Algorithm 2: Outsourcing Phase

```
input : $(v_0, v_1, \ldots, v_n)$, $v_i$ is a variant in a patient's VCF file, n is the number of variants in the VCF file
Initialize a tree $T_i^V$ with a root node. Root node do not store any data
while $i \leq n$ do
    $s \leftarrow MapVariant(v_i)$
    Insert $s$ to $T_i^V$ in a random fashion (see description below)
Initialize a tree $T_i^R$ with a root node which is isomorphic to $T_i^V$ and assign uniformly distributed binary random bits to its nodes
Send $T_i^R$ to the proxy server $D_1$
Send $T_i^V \oplus T_i^R$ to the proxy server $D_2$
```

3.5.2 Querying phase

We assume that the two proxy servers initialized a pseudo-random number generator (PRNG) with the same random value (seed) for the tree traversal in our protocol. Thus, the two proxy servers pick the same random values during the tree traversal.

Each proxy stores a pointer on the variant tree for each variant in the query, starting at the root node. On each level iteration, the proxies select one child of the pointer randomly. Then, they compare the bit stored at the pointer location with the corresponding bit of the query variant. This comparison is done locally by calculating XOR of two values, without any communication between the two proxy servers.

They reconstruct the comparison result by sending shares to each other. They update the pointers dependent on the comparison results. If the comparison result is 0, the pointer is set to the selected child of the pointer. If the comparison is 1, the pointer is set to the unselected child of the pointer. This process is repeated for all levels of the tree and in the end, each pointer is set to leaf nodes.

The variants hold in the selected leaf node are compared to the corresponding query variant. Before this comparison is performed, a share of the mask bit of the corresponding variant tree is added to

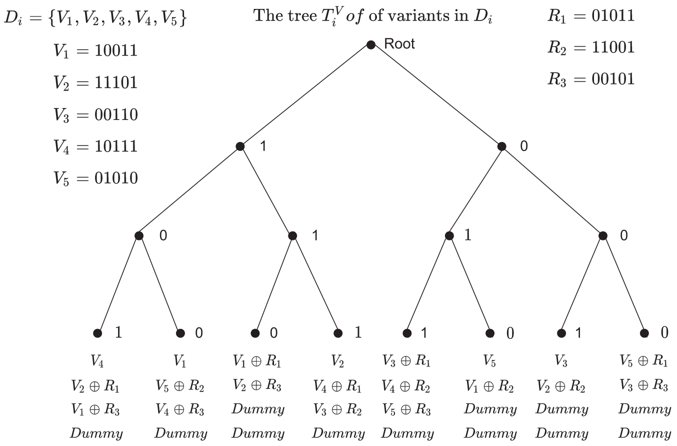
$D_i = \{V_1, V_2, V_3, V_4, V_5\}$
$V_1 = 10011$
$V_2 = 11101$
$V_3 = 00110$
$V_4 = 10111$
$V_5 = 01010$
The tree T_i^V of variants in D_i
Root
$R_1 = 01011$
$R_2 = 11001$
$R_3 = 00101$
1
0
0
1
1
0
1
0
0
1
1
0
1
0
V_4 $V_2 \oplus R_1$ $V_1 \oplus R_3$ $Dummy$
V_1 $V_5 \oplus R_2$ $V_4 \oplus R_3$ $Dummy$
$V_1 \oplus R_1$ $V_2 \oplus R_3$ $Dummy$ $Dummy$
V_2 $V_4 \oplus R_1$ $V_3 \oplus R_2$ $Dummy$
$V_3 \oplus R_1$ $V_4 \oplus R_2$ $V_5 \oplus R_3$ $Dummy$
V_5 $V_1 \oplus R_2$ $Dummy$ $Dummy$
V_3 $V_2 \oplus R_2$ $Dummy$ $Dummy$
$V_5 \oplus R_1$ $V_3 \oplus R_3$ $Dummy$ $Dummy$

Fig. 3. Generation of the variant tree

the end of the share of the query variant. The logical OR of all bits in the comparison result is computed. Then dummy query variants are pruned and the query result for each patient is computed. Finally, the query results of all patients are summed. Details of this phase are described in Algorithm 3 formally.

3.6 Security considerations

We discuss the security of our scheme in this section. The main purpose of our protocol is to provide privacy for patients whose data are outsourced to our service. The privacy of our scheme is based on the proven security of the GMW protocol (Goldreich *et al.*, 1987). We assume that the two proxy servers used for secure computation do not collude. Genome data are shared between the two servers using arithmetic sharing. A semi-honest adversary corrupting at most one of the two proxy servers can observe a share of patients' data. Because the data are shared with arithmetic sharing it looks like uniformly distributed random data and this prevents the leakage of the patients' data.

The proxy servers cannot learn the data access patterns such as the path of query variants in the variant trees. Because topology of each variant tree is the same, it is not possible to match a variant tree to any VCF file. In the querying process, for each query variant, one of the previously determined masks is randomly selected and the query variants are XORed with these masks. Therefore, the proxy servers can detect if a query for the same variants is received twice with a probability of $\frac{1}{(F+1)^{(m+v)}}$ where F is the number masks, m is the number of query variants and v is the number of dummy variants in query. If F is 16, m is 5 and v is 3 the proxy servers can detect if a query with the same variants is received twice with the probability of $\frac{1}{17^8}$. This probability is further reduced as the number of dummy variants added to the user query increases. Furthermore, the probability of a query with the same variants from the same user or different users to have the same dummy variants is negligible. Therefore, the probability of sending two queries with the same variants to proxy servers is negligible.

Our solution is vulnerable to re-identification attacks (Shringarpure and Bustamante, 2015), although it hides which hospitals' genomic databases the matches occur in. One way to reduce the risk of re-identification is to return the query result if it is larger than a certain threshold as stated in Shringarpure and Bustamante (2015). It is possible to check the threshold value with an extra comparison gate as done in Demmler *et al.* (2017). However, this extension will negatively affect the usability of our solution in rare disease studies.

Confidentiality, integrity and, authentication between all parties are provided using state-of-the-art technologies such as TLS (Dierks and Rescorla, 2008).

The formal security proof of the proposed solution is given in the Supplementary Material.

4 Implementation

In this study, we choose the GMW protocol (Goldreich *et al.*, 1987) for our implementations. We have implemented the proposed solution using the C++ programming language and ABY framework (Demmler *et al.*, 2015), which provides an efficient implementation of secure two-party computation protocols. This framework works like a virtual machine that abstracts secure computation protocols. Since we have XOR-based sharing, we benefit from the implementation of the GMW protocol in the ABY framework.

Implementation of the proposed solution requires the construction of multiple Boolean circuits. First, we need a circuit to compare the bits stored on the pointers with the corresponding query bits. $m \times s$ bits comparison is calculated where m is the number of queries and s is the number of patients. The comparison is done as many times as the depth of variant trees. We perform the single bit comparison using XOR operation locally.

We need a separate circuit to compare the variants stored in leaf nodes with the query variants. This is the largest circuit in our implementation which compares $m \times s \times f$ variants where f is the number of variants (including dummy variants) in each node. Each comparison produces a single bit. The comparison result of each patient is given to the OR tree in order to find whether there is a match. Thus, a single comparison result is produced for each query variant from each patient tree. The comparison results and the bit vector showing whether query variants are dummy are given to OR operation to prune the dummy variants in the query. The pruned comparison results obtained for each patient are given to the AND tree. We get s bits, each representing a query result for a different patient. Each patient result is given to an ADD tree and the final query result is obtained. In our implementation, we use AND and ADD trees to logarithmically reduce the circuit depth.

5 Results

We conducted our experiments on two Amazon EC2 t2.xlarge instances having Ubuntu 16.04 with the 4.13.0-36-generic kernel each from the Frankfurt and London regions, compiled our implementation with gcc v5.4.0, and use a symmetric security level of 128 bits. Since the implementation of Demmler *et al.* (2017) was not available, we reimplement their algorithm based on the details given in their article. This implementation is also available alongside our own implementation. We reported two different execution times in the experimental results. In our solution, in order to avoid high memory requirements, operations on variant trees are performed by

Algorithm 3: Querying Phase

```
input : $(\langle q_1\rangle, \langle q_2\rangle, \ldots, \langle q_m\rangle), (\langle b_1\rangle, \langle b_2\rangle, \ldots, \langle b_m\rangle),$
        $(\langle T_1\rangle, \langle T_2\rangle, \ldots, \langle T_s\rangle)$, $\langle q_i\rangle$ is a share of a variant mapped to 48-bit
        unsigned integer, $\langle b_i\rangle$ is a share of a bit indicating whether a variant is a
        dummy, $\langle T_j\rangle$ is one share of a patient's variant tree, $d$ is bit-length of a
        variant, $m$ is the number of query variants, $s$ is the number of patients, $f$
        is the number of variants stored in each leaf node
Initialize a pointer vector $p$ with size of $m \times s$. Set $m$ different pointers on each
  variant tree. Initially, all the pointers are the root nodes of the trees
Initialize a pointer vector $tmp\_p$ with size of $m \times s$
Initialize an unsigned integer vector $\langle path\rangle$ of size $m \times s$, that stores path to
  possible variant values. Set all elements to zero: $\langle path[i][j]\rangle \leftarrow 0$
$r \leftarrow 0$
while $r \leq d$ do
    Initialize a bit array $\langle tree\_bits\rangle$ of size $m \times s$ and a bit array
      $\langle variant\_bits\rangle$ of size $m \times s$
    for $i \leftarrow 1$ to $s$ do
        for $j \leftarrow 1$ to $m$ do
            $tmp\_p[i][j] \in \{p[i][j].left, p[i][j].right\}$ /* choose left
              or right child node randomly */
            $\langle tree\_bits[i][j]\rangle \leftarrow tmp\_p[i][j].val$
            $\langle variant\_bits[i][j]\rangle \leftarrow \langle q_j[r]\rangle$ /* gets rth bit of jth
              query variant */
    $\langle result\_bits\rangle \leftarrow \langle tree\_bits\rangle \oplus \langle variant\_bits\rangle$        /* local
      comparison */
    $result\_bits \leftarrow reconstruct(\langle result\_bits\rangle)$
    for $i \leftarrow 1$ to $s$ do
        for $j \leftarrow 1$ to $m$ do
            if $result\_bits[i][j] = 0$ then
                $p[i][j] \leftarrow tmp\_p[i][j]$
            else
                if $tmp\_p[i][j] = p[i][j].left$ then
                    $p[i][j] \leftarrow p[i][j].right$
                else
                    $p[i][j] \leftarrow p[i][j].left$
            $\langle path[i][j][r]\rangle \leftarrow p[i][j].val$
    $r \leftarrow r + 1$
Initialize an unsigned integer vector $\langle v\rangle$ of size $m \times s \times f$
Initialize an unsigned integer vector $\langle tmp\_q\rangle$ of size $m \times s \times t$
  /* repetition of query variants $\langle q\rangle$ by $s \times f$ times */
Initialize a bit vector $\langle tmp\_b\rangle$ of size $m \times s$          /* repetition of
  cancellation bits $\langle b\rangle$ by the number of patients $s$ */
for $i \leftarrow 1$ to $s$ do
    for $j \leftarrow 1$ to $m$ do
        $\langle tmp\_b[i][j]\rangle \leftarrow \langle b_j\rangle$
        for $k \leftarrow 1$ to $f$ do
            $\langle v[i][j][k]\rangle \leftarrow \langle path[i][j]\rangle || \langle variants\_of\_path[k]\rangle$
            $\langle tmp\_q[i][j][k]\rangle \leftarrow \langle q_j\rangle || \langle x_i\rangle$
$\langle res\_comp\rangle \leftarrow secure\_comparison(\langle v\rangle, \langle tmp\_q\rangle)$
for $i \leftarrow 1$ to $s$ do
    for $j \leftarrow 1$ to $m$ do
        $\langle res\_q[i][j]\rangle \leftarrow \langle res\_comp[i][j][1]\rangle$
        for $k \leftarrow 2$ to $f$ do
            $\langle res\_q[i][j]\rangle \leftarrow$
              $secure\_or(\langle res\_q[i][j]\rangle, \langle res\_comp[i][j][k]\rangle)$
$\langle res\_q\rangle \leftarrow secure\_or(\langle res\_q\rangle, \langle tmp\_b\rangle)$          /* pruning dummy
  variants */
$\langle res\rangle \leftarrow 0$
for $i \leftarrow 1$ to $s$ do
    $\langle and\_res\rangle \leftarrow \langle res\_q[i][1]\rangle$
    for $j \leftarrow 2$ to $m$ do
        $\langle and\_res\rangle \leftarrow secure\_and(\langle and\_res\rangle, \langle res\_q[i][j]\rangle)$
    $\langle res\rangle \leftarrow secure\_add(\langle res\rangle, \langle and\_res\rangle)$
Send $\langle res\rangle$ to the client
```

reading the relevant bits from the hard drive. Therefore, there is a delay due to IO operations. We reported this delay as IO time.

In our experiments, we compare our solution with Demmler *et al.*'s solution in terms of execution time and communication cost. We show how the execution times of both solutions scale for varying variant count, and query length. The execution time and communication cost of both solutions increase linearly with the variant count and query length. Figure 4 shows the execution time of both solutions. The increase in the execution time and communication cost of our protocol with respect to the increasing variant count and query length is negligible when compared to those of Demmler *et al.*'s solution (see Tables 1 and 2). The reason for this performance improvement is that patients' variants are stored in a binary tree.

We show the execution time of our solution increases linearly with the number of masks in Table 3 and in Figure 5a. Since the number of variants remains constant, the IO time remains constant. However, the MPC time increases linearly because each leaf node stores as many variants as the number of masks.

Unlike the previous studies, we show that how our solution scales with varying number of patients in our experiments. We consider the number of patients between 1024 and 1 048 576 (see Fig. 5b and Table 4). These results show the ability of our solution to scale up to tens of millions of patients and it is much more practical than Demmler *et al.*'s solution. For querying 1 048 576 patients' exome databases each containing 1 000 000 variants, assuming a query of length 5, their solution takes 196 754 h approximately. Our solution takes 3.23 h for the same operation. Demmler *et al.* performed all tests on a database having single patient data. They stated that there is a need for dedicated hardware and faster networks to query multiple patient data and that this querying cannot give instant answers. The cost of the dedicated hardware required for Demmler *et al.*'s solution to work effectively on the data of millions of patients will be very high. For example, querying 1 048 576 patients' exome databases each containing 1 000 000 variants with a query of length 5 in 12 min, Demmler *et al.*'s solution requires 951 816 CPU cores, ethernet cards, internet connections and 31 457 of GB memory. We assume that a CPU core is needed for each parallel execution. Our solution needs 16 CPU cores, ethernet cards, internet connections and 0.50 GB of memory to answer the same query in 12 min. This shows that with a small amount of dedicated hardware, our solution can respond to queries running on millions of patient data in reasonable times.

Furthermore, under the assumption that the total number of variants that all databases have is fixed, we show that how the running time and communication cost of our solution scale with the varying number of databases (see Table 5). Although the total number of variants in the system remains constant, as the number of databases increases, the costs of our solution increase linearly. The first reason for this is that the number of IO operations required to traverse the trees increases. The second reason is that each tree yields a set of

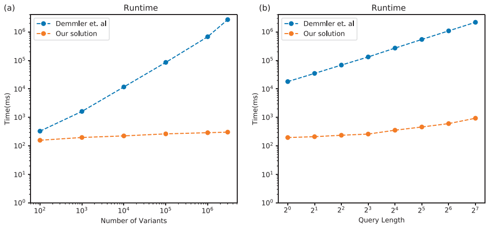

Fig. 4. Comparison of time performance of our solution and Demmler *et al.*'s solution (Demmler *et al.*, 2017) under various numbers of variants/numbers of query variants. (**a**) Runtime with a single patient, a varying number of variants, a fixed variant length = 48 bit, and 5 query variants, (**b**) runtime with a single patient, 100 000 variants, a fixed variant length = 48 bit and a varying number of query variants

Table 1. Benchmark results and circuit properties for varying variant count at fixed variant length 48 bit, query length 5 and mask count 16

	Demmler *et al.*'s solution				Our solution					
No. of variant	No. of ANDs	Depth	Time (ms)	Comm (MB)	No. of ANDs	Depth	IO time (ms)	MPC time (ms)	Total time (ms)	Comm (MB)
100	3.0×10^4	17	330	<1	3.8×10^3	13	<1	158	158	<1
1000	2.4×10^5	20	1623	4	3.8×10^3	13	<1	198	198	<1
10 000	3.9×10^6	24	11 721	63	3.8×10^3	13	1	224	225	<1
100 000	3.1×10^7	27	85 126	510	3.8×10^3	13	1	264	265	<1
1 000 000	2.5×10^8	30	675 562	4089	3.8×10^3	13	1	290	291	<1
3 000 000	1.0×10^9	32	2 694 093	16 357	3.8×10^3	13	1	304	305	<1

Table 2. Benchmark results and circuit properties for varying query length at fixed variant length 48 bit, variant count 100 000 and mask count 16

	Demmler *et al.*'s solution				Our solution					
No. of queries	No. of ANDs	Depth	Time (ms)	Comm (MB)	No. of ANDs	Depth	IO Time (ms)	MPC Time (ms)	Total Time (ms)	Comm (MB)
1	6.3×10^6	24	18 070	103	767	11	<1	197	197	<1
2	1.2×10^7	25	34 982	204	1535	12	<1	211	211	<1
4	2.5×10^7	26	68 587	409	3071	13	1	237	238	<1
8	5.1×10^7	27	132 171	830	6143	14	2	261	263	<1
16	1.0×10^8	28	270 283	1635	12 287	15	4	356	360	<1
32	2.0×10^8	29	542 836	3271	24 575	16	7	462	469	<1
64	4.0×10^8	30	1 085 198	6542	49 151	17	14	607	621	<1
128	8.0×10^8	31	2 171 901	13 086	98 303	21	29	937	966	1

Table 3. Benchmark results and circuit properties for varying mask count at fixed variant length 48 bit, variant count 1 000 000 and query length 5

No. of masks	No. of ANDs	Depth	IO time (ms)	MPC time (ms)	Total time (ms)	Comm (MB)
16	3.8×10^3	13	1	277	278	<1
64	1.5×10^4	15	1	699	700	<1
256	6.1×10^4	17	1	1174	1175	1
1024	2.4×10^5	19	1	1901	1902	4
4096	9.6×10^5	21	1	5108	5109	16
16 384	3.9×10^6	23	1	11 286	11 287	63

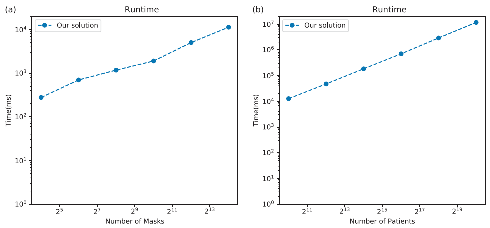

Fig. 5. Time performance of our solution under various numbers of masks/numbers of patients. (**a**) Runtime with a varying number of masks, 1 000 000 variants, a fixed variant length = 48 bit, and 5 query variants, (**b**) Runtime with a varying number of patients, 1 000 000 variants, a fixed variant length = 64 bit and 5 query variants

candidate variants to correspond to each query variant. As the number of trees increases, the number of candidate variant sets and therefore the amount of data that will be input to secure comparison increases.

6 Conclusion

In this study, we develop a solution able to perform privacy-preserving genomic variant queries on multiple center genomic databases. Our solution enables secure storage of genomic databases in cloud environments. It provides data privacy, query privacy and output privacy. Most importantly, unlike the previous studies, it allows secure variant queries to work on millions genomic databases and significantly reduce the execution time of these queries. We prove the security of our solution against honest-but-curious proxy servers and malicious clients, and we implement and test the performance of our protocol. Our experimental results demonstrate that the running time of a query having five query variants on a single genomic database with 3 000 000 variants is 305 ms. Furthermore, the running time of a query having five query variants on 1 048 576 genomic databases, each with 1 000 000 variants, is 3.23 h on a single core. If this query is executed on 32 cores with some other dedicated hardwares it takes 6 min approximately. These running times are very convincing in terms of integration into real

Table 4. Benchmark results and circuit properties for varying patient count at fixed query length 5, variant length 48 bit, variant count 1 000 000 and the number of masks 16

No. of patient	No. of ANDs	Depth	IO time (ms)	MPC time (ms)	Total time (ms)	Comm (MB)
2^{10}	3.9×10^{6}	29	1048	11 602	12 650	64
2^{12}	1.5×10^{7}	34	3998	43 190	47 188	255
2^{14}	6.2×10^{7}	39	15 663	167 019	182 682	1023
2^{16}	2.5×10^{8}	46	68 307	631 831	700 138	4092
2^{18}	1.0×10^{9}	49	255 113	2 634 416	2 889 529	16 371
2^{20}	4.0×10^{9}	54	1 021 196	10 607 612	11 628 808	65 485

Table 5. Benchmark results and circuit properties for varying database count at fixed query length 5, variant length 48 bit, total number of variants of all patients $2^{14} \times 10^{6}$ and the number of masks 16

No. of databases	No. of ANDs	Depth	IO time (ms)	MPC time (ms)	Total time (ms)	Comm (MB)
2^{8}	3.9×10^{6}	29	302	3307	3609	64
2^{10}	3.9×10^{6}	29	1129	12 270	13 399	64
2^{12}	1.5×10^{7}	34	4197	45 603	49 800	255
2^{14}	6.2×10^{7}	39	15 663	167 019	182 682	1023
2^{16}	2.5×10^{8}	46	59 471	628 844	688 315	4092
2^{18}	1.0×10^{9}	49	223 026	2 352 573	2 575 599	16 371
2^{20}	4.0×10^{9}	54	835 895	8 779 585	9 615 480	65 485

medical systems. To the best of our knowledge, this is the first solution that provides this performance for querying multiple genomic databases.

Funding

This study is supported by the German Ministry of Research and Education (BMBF), project number 01ZZ2010. Furthermore, MA, NP, and OK acknowledge funding from the German Federal Ministry of Education and Research (BMBF) within the ’Medical Informatics Initiative’ (DIFUTURE, reference number 01ZZ1804D)..

Conflict of Interest: none declared.

References

Akgün,M. *et al.* (2015) Privacy preserving processing of genomic data: a survey. *J. Biomed. Inf.*, **56**, 103–111.

Amendola,L.M. *et al.*; CSER Consortium. (2018) The clinical sequencing evidence-generating research consortium: integrating genomic sequencing in diverse and medically underserved populations. *Am. J. Hum. Genet.*, **103**, 319–327.

Asharov,G. *et al.* (2013) More efficient oblivious transfer and extensions for faster secure computation. In: Sadeghi,A. *et al.* (eds.) *2013 ACM SIGSAC Conference on Computer and Communications Security, CCS'13, Berlin, Germany, November 4–8, 2013*. ACM, Berlin, Germany, pp. 535–548.

Asharov,G. *et al.* (2018) Privacy-preserving search of similar patients in genomic data. *PoPETs*, **2018**, 104–124.

Aziz,M.M.A. *et al.* (2017) Privacy-preserving techniques of genomic data—a survey. *Brief Bioinform.*, **20**, 887–895.

Bogdanov,D. *et al.* (2008) Sharemind: a framework for fast privacy-preserving computations. In: *Computer Security – ESORICS 2008, 13th European Symposium on Research in Computer Security*, October 6–8, 2008, Málaga, Spain. pp. 192–206.

Børsting,C. and Morling,N. (2016) Genomic applications in forensic medicine. In: Kumar,D. and Antonarakis,S (eds.) *Medical and Health Genomics*. Academic Press, Oxford, pp. 295–309.

Consortium,W.T.C.C. *et al.* (2007) Genome-wide association study of 14,000 cases of seven common diseases and 3,000 shared controls. *Nature*, **447**, 661.

Demmler,D. *et al.* (2015) ABY – a framework for efficient mixed-protocol secure two-party computation. In: *22nd Annual Network and Distributed System Security Symposium, NDSS 2015*, San Diego, California, USA, February 8–11, 2015.

Demmler,D. *et al.* (2017) Privacy-preserving whole-genome variant queries. In: *Cryptology and Network Security - 16th International Conference, Hong kong, CANS 2017*.

Deng,X. and Nakamura,Y. (2017) Cancer precision medicine: from cancer screening to drug selection and personalized immunotherapy. *Trends Pharmacol. Sci.*, **38**, 15–24.

Dierks,T. and Rescorla,E. (2008) The Transport Layer Security (TLS) Protocol Version 1.2. RFC 5246 (Proposed Standard). Updated by RFCs 5746, 5878, 6176.

Erlich,Y. and Narayanan,A. (2014) Routes for breaching and protecting genetic privacy. *Nat. Rev. Genet.*, **15**, 409–421.

Froelicher,D. *et al.* (2017) Unlynx: a decentralized system for privacy-conscious data sharing. *PoPETs*, **2017**, 232–250.

Global Alliance for Genomics and Health. (2016) A federated ecosystem for sharing genomic, clinical data. *Science*, **352**, 1278–1280.

Goldreich,O. *et al.* (1987) How to play any mental game. In: *Proceedings of the Nineteenth Annual ACM Symposium on Theory of Computing*, STOC '87, New York, NY, USA. ACM, pp. 218–229.

Hasan,M.Z. *et al.* (2018) Secure count query on encrypted genomic data. *J. Biomed. Inf.*, **81**, 41– 52.

Huang,Y. *et al.* (2011) Faster secure two-party computation using garbled circuits. In: *20th USENIX Security Symposium*, August 8–12, 2011, San Francisco, CA, USA.

Hyde,C.L. *et al.* (2016) Identification of 15 genetic loci associated with risk of major depression in individuals of European descent. *Nat. Genet.*, **48**, 1031–1036.

Jha,S. *et al.* (2008) Towards practical privacy for genomic computation. In: *2008 IEEE Symposium on Security and Privacy (S&P 2008), 18–21 May 2008*, Oakland, California, USA, pp. 216–230.

Kamara,S. and Raykova,M. (2011) Secure outsourced computation in a multi-tenant cloud. In: *IBM Workshop on Cryptography and Security in Clouds, Zurich, Switzerland*, pp. 15–16.

Khan,R. and Mittelman,D. (2018) Consumer genomics will change your life, whether you get tested or not. *Genome Biol.*, **19**, 1–4.

Liu,C. *et al.* (2015) Oblivm: a programming framework for secure computation. In: *2015 IEEE Symposium on Security and Privacy*, *SP 2015*, *May 17–21*, *2015*, San Jose, CA, USA, pp. 359–376.

Malkhi,D. *et al.* (2004) Fairplay – secure two-party computation system. In: *Proceedings of the 13th USENIX Security Symposium, August 9–13, 2004*, San Diego, CA, USA, pp. 287–302.

Mittos,A. *et al.* (2019) Systematizing genome privacy research: a privacy-enhancing technologies perspective. *PoPETs*, **2019**, 87–107.

Naveed,M. *et al.* (2015) Privacy in the genomic era. *ACM Comput. Surv.*, **486**, 1–44.

Raisaro,J.L. *et al.* (2017) Addressing beacon re-identification attacks: quantification and mitigation of privacy risks. *JAMIA*, **24**, 799–805.

Salem,A. *et al.* (2019) Privacy-preserving similar patient queries for combined biomedical data. *PoPETs*, **2019**, 47–67.

Shringarpure,S.S., and Bustamante,C.D. (2015) Privacy risks from genomic data-sharing beacons. *Am. J. Hum. Genet.*, **97**, 631–646.

Sousa,J.S. *et al.* (2017) Efficient and secure outsourcing of genomic data storage. *BMC Med. Genomics*, **10**, 46.

Sürün,B. *et al.* (2020) Clinvap: a reporting strategy from variants to therapeutic options. *Bioinformatics*, **36**, 2316–2317.

von Thenen,N. *et al.* (2019) Re-identification of individuals in genomic data-sharing beacons via allele inference. *Bioinformatics*, **35**, 365–371.

Wang,X.S. *et al.* (2015) Efficient genome-wide, privacy-preserving similar patient query based on private edit distance. In: *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security, October 12–16, 2015*, Denver, CO, USA, pp. 492–503.

Yao,A.C.-C. (1986). How to generate and exchange secrets. In: *Proceedings of the 27th Annual Symposium on Foundations of Computer Science, SFCS '86*. IEEE Computer Society, Washington, DC, USA, pp. 162–167.