A Framework for Secure Computations With Two Non-Colluding Servers and Multiple Clients, Applied to Recommendations

Thijs Veugen, Robbert de Haan, Ronald Cramer, and Frank Muller

Abstract—We provide a generic framework that, with the help of a preprocessing phase that is independent of the inputs of the users, allows an arbitrary number of users to securely outsource a computation to two non-colluding external servers. Our approach is shown to be provably secure in an adversarial model where one of the servers may arbitrarily deviate from the protocol specification, as we can employ an arbitrary number of dummy users. We use these techniques to implement a secure recommender system based on collaborative filtering that becomes more secure, and significantly more efficient than previously known implementations of such systems, when the preprocessing efforts are excluded. We suggest different alternatives for preprocessing, and discuss their merits and demerits.

Index Terms—Secure multi-party computation, malicious model, secret sharing, client-server systems, preprocessing, recommender systems.

I. INTRODUCTION

RECOMMENDATION systems consist of a processor together with a multitude of users, where the processor provides recommendations to requesting users, which are deduced from personal ratings that were initially submitted by all users. It is easy to see that, in a non-cryptographic setup of such a system, the processor is both able to learn all data submitted by the users, and spoof arbitrary, incorrect recommendations.

In this work we replace the recommendation processor by a general two-server processor in such a way that, as long as one of the two servers is not controlled by an adversary and behaves correctly,

1) the privacy of the ratings and recommendations of the users is maintained to the fullest extent possible, and
2) a server that is under adversarial control is unable to disrupt the recommendation process in such a way that an incorrectly computed recommendation will not be detected by the requesting user.

Our result uses a modified version of the standard model for secure multi-party computation, which is a cryptologic paradigm in which the players (recommenders) jointly perform a single secure computation and then abort. In our model the computation is ongoing (recommendations are repeatedly requested) and outsourced to two external servers that do not collude. This approach allows for the involvement of many users that need only be online for very short periods of time in order to provide input data to, or request output data from, the servers. In practice, one of the two servers could be the service provider that wishes to recommend particular services to users, and the other server could be a governmental organisation guarding the privacy protection of users. The role of the second server could also be commercially exploited by a privacy service provider, supporting service providers in protecting the privacy of their customers.

While this work focuses on the application of secure recommendation systems, we point out that our underlying framework is sufficiently generic for use in other, similar applications, and also easily extends to model variations involving more than two servers. In general, any configuration is foreseen consisting of multiple servers that have a joint goal of securely delivering a service to a multitude of users. The online phase will be computationally secure as long as one of the servers is honest [1]. However, the application should allow for a significant amount of preprocessing, which is independent of the data, and could be performed during inactive time.

Although most related work (see Subsection I-A) is only secure in the semi-honest model, we provide security in the malicious model (see Subsection III-B). There are several reasons why security in the malicious model is important. First of all, the main goal of securing the system is that we do not want the servers to learn the personal data of users. We therefore should not just trust them to follow the rules of the protocol. Second, in a malicious model the correctness of the user outputs is better preserved, because outputs cannot be corrupted by one (malicious) server on his own. Third, a malicious server might introduce a couple of new, so called
dummy users, which he controls. These dummy users might help him deduce more personal data than is available through the protocol outputs. We show that in our malicious model this is not possible.

A. Related Work

Most related work on privacy preserving recommendations is secure in the semi-honest model, so parties are assumed to follow the rules of the protocol. As mentioned by Lagendijk et al. [2], “Achieving security against malicious adversaries is a hard problem that has not yet been studied widely in the context of privacy-protected signal processing.”

Erkin et al. [3] securely computed recommendations based on collaborative filtering. They used homomorphic encryption within a semi-honest security model just like Bunn and Ostrovsky [4]. Goethals et al. [5] stated that although such techniques can be made secure in the malicious model, that will make them unsuitable for real life applications because of the increased computational and communication costs.

Polat and Du [6] used a more lightweight approach by statistically hiding personal data, which unfortunately has been proven insecure by Zhang et al. [7]. Atallah et al. [8] used a threshold secret-sharing approach for secure collaborative forecasting with multiple parties.

Nikolaenko et al. [9] securely computed collaborative filtering by means of matrix factorization. They used both homomorphic encryption and garbled circuits in a semi-honest security model. In another paper [10], these authors use similar techniques to securely implement Ridge regression, a different approach of collaborative filtering.

Catrina and de Hoogh [11] developed an efficient framework for secure computations in the semi-honest model, based on secret sharing and statistical security, which could also be used for a recommender system. Peter et al. [12] considered a model where users can outsource computations to two non-colluding servers. They use homomorphic encryption, each user having its own key, but require the servers to follow the rules of the protocol.

Canny’s approach [13] on private collaborative filtering is able to cope with malicious behaviour. A detailed comparison with our approach can be found in Subsection V-E.

In the last several years, a couple of computation protocols have been developed, which are both practical and secure in the malicious model. The idea is to use public-key techniques in a data-independent pre-processing phase, such that cheap information-theoretic primitives can be exploited in the online phase, which makes the online phase efficient. In 2011, Bendlin et al. [1] presented such a framework with a somewhat homomorphic encryption scheme for implementing the pre-processing phase. This has been improved lately by Damgård et al. [14], which has become known as SPDZ (pronounced “Speedz”). Last year, Damgård et al. [15] showed how to further reduce the precomputation effort.

B. Our Contribution

We used the SPDZ framework, as introduced in the previous subsection, which enables secure multi-party computations in the malicious model, extended it to the client-server model, and worked out a secure recommendation system within this setting. Not only did this lead to a recommendation system that is secure in the malicious model, but also the online phase became very efficient in terms of computation and communication complexity.

To extend SPDZ to the client-server model, we developed secure protocols that enable users (clients) to upload their data to the servers, and afterwards obtain the computed outputs from the servers. This required a subprotocol for generating duplicate sharings, as described in Appendix A.

To securely compute a recommendation within SPDZ, we had to develop a secure comparison protocol and a secure integer division protocol. For the comparison protocol, we combined ideas from Nishide and Ohta [16], and Schoenmakers and Tuyls [17]. In fact, we finetuned the bitwise comparison protocol from [17] to a linear-round secret-sharing setting, and replaced the constant-round solution of [16]. For the integer division, which forms the bottleneck of our application, we took a secure solution from Bunn and Ostrovsky [4], translated it to a secret-sharing setting, and put in our secure comparison protocol. This yielded an efficient secure integer division, because its output was relatively small in our application.

Finally, we took the effort for implementing and testing our recommender system within the SPDZ framework, and comparing it with the current state-of-the-art.

C. Organization of the Paper

We first explain the basics of collaborative filtering and authenticated secret sharing, which are necessary for our paper. In Section III we describe how we fit secure multi-party computation in the client-server model, and the resulting security model. Section IV focusses on the secure implementation of our recommender system, the various options for preprocessing, and the more complicated subprotocols that are needed. The complexity of this implementation is analysed in Section V, and compared with related work. Finally, we stress the wide applicability of our secure framework.

II. Preliminaries

After explaining collaborative filtering, we introduce the concepts of SPDZ: tagged secret sharing, and the basic protocols of the computation phase.

A. User-Based Collaborative Filtering

We provide as an example application, taken from [3], a basic implementation for generating recommendations, more specifically a collaborative filtering method with user-neighborhood-based rating prediction [18]. We do not pretend to present the best recommender system, but merely want to introduce some basic components, and show how they can be implemented securely.

There is one processor \( \mathcal{R} \), with \( N \) users, and \( M \) different predefined items. A small subset of size \( S \) \((1 \leq S < M)\) of these items is assumed to have been rated by each user,
reflecting his or her personal taste. The remaining \( M - S \) items have only been rated by a small subset of users that have experienced the particular item before. A user that is looking for new, unrated items, can ask the processor to produce estimated ratings for (a subset of) the \( M - S \) items. The number \( N \) of users can be large, say 1 million, \( M \) is in the order of hundreds, and \( S \) usually is a few tens [18].

During initialisation, each user \( n \) uploads to the processor a list of at most \( M \) ratings of items, where each rating \( V_{(n,m)} \) is represented by a value within a pre-specified interval. Users can update their ratings at any time during the lifetime of the system. A user can, at any time after the initialisation, request a recommendation from the processor. When the processor receives such a request from a user, it computes a recommendation for this user as follows. First, it uses the initial \( S \) ratings in each list to determine which other users are considered to be similar to the requesting user, i.e. have similarly rated the first \( S \) items. The remaining \( M - S \) entries in the lists are then used to compute and return a recommendation for the requesting user, consisting of \( M - S \) ratings averaged over all similar users.

To get an idea of the required computation we describe the required computational steps. Let \( U_m \) be the set of users that have rated item \( m \), \( S < m \leq M \).

1) Each user uploads his (between \( S \) and \( M \)) ratings to the processor. Like in [3], we assume the first \( S \) (similarity) ratings have been normalized and scaled beforehand. A rating is normalized by dividing it by the length (Euclidean norm) of the vector \((V_{(n,1)}, \ldots, V_{(n,S)})\), yielding a real number between 0 and 1. Next, this real number is scaled and rounded to a positive integer consisting of a few (e.g. four) bits. The remaining ratings should only be scaled and rounded to an integer with the same maximal number of bits.

2) When user \( A \) asks for a recommendation, the processor computes \( M - S \) estimated ratings for \( A \). First, the similarities \( Sim_{A,n} = \sum_{m=1}^{S} V_{(n,m)} \cdot V_{(A,m)} \) are computed for each user \( n \).

3) Each similarity value is compared with a public threshold \( t \in \mathbb{N}^+ \), and its outcome is presented by the bit \( \delta_n = (t < Sim_{A,n}) \).

4) The recommendation for user \( A \) consists of \( M - S \) estimated ratings, the estimated rating for item \( m \), \( S < m \leq M \), simply being an average of the ratings of the similar users: \( \text{Rec}_m = (\sum_{n \in U_m} \delta_n \cdot V_{(n,m)}) \div (\sum_{n \in U_m} \delta_n) \), where \( \div \) denotes integer division.

5) The processor sends back the recommendation \( \text{Rec}_{S+1} \ldots \text{Rec}_M \) to user \( A \).

Here, we used the notation \((x < y)\) to denote the binary result of the comparison of two numbers \( x \) and \( y \), which equals 1 when the comparison is true, and 0 otherwise.

B. Tagged Secret Sharing

Let \( p \) be a prime consisting of \( \ell \) bits. We denote \( \mathbb{F}_p \) the field of integers modulo \( p \) with ordinary addition and multiplication, and use the elements as if they were integers of the set \( \{0, 1, \ldots, p-1\} \). We use + for addition and \( \cdot \) for multiplication, both in the field \( \mathbb{F}_p \), and therefore omit the reduction modulo \( p \). All secret values, which are elements of \( \mathbb{F}_p \), are additively shared between the servers. The framework of tagged secret sharing used here is from Bendlin et al. [1].

1) Authenticated Secret Sharing: An external dealer distributes shares of a secret value \( x \in \mathbb{F}_p \) to the two servers as follows:

1) The dealer selects a value \( r \in \mathbb{F}_p \) uniformly at random.
2) The dealer sends the value \( r \) to server 1 and the value \( x - r \) to server 2.

The values \( x_1 = r \) and \( x_2 = x - r \) are considered to be the share of server 1 and the share of server 2, respectively. It should be clear from the description above that the shares \( x_1 \) and \( x_2 \) are both individually statistically independent of the secret \( x \), while they together allow to determine the value of \( x \), by simply adding these shares together.

In addition to the distribution of the shares, the dealer distributes authentication tags on the shares with respect to the authentication code \( C : \mathbb{F}_p \times \mathbb{F}_p^2 \to \mathbb{F}_p \), defined as \( C(x, (\alpha, \beta)) = \alpha \cdot x + \beta \). Here the value \((\alpha, \beta)\) is called the authentication key and the value \(\alpha \cdot x + \beta\) the authentication tag for the share \( x \).

For every share \( x_1 \) for server 1, the dealer generates a random authentication key \((\alpha_2, \beta_2)\), computes the corresponding authentication tag \( m_1 = \alpha_2 \cdot x_1 + \beta_2 \) and sends the key \((\alpha_2, \beta_2)\) to server 2, and the share \( x_1 \) and tag \( m_1 \) to server 1. Symmetrically, this is done for the share \( x_2 \) of server 2. We use notation \((x)\) to denote such a randomized share distribution with corresponding authentication keys and tags for the value \( x \), and call \((x)\) an (authenticated) secret sharing. Equivalently, we say that the value \( x \) is secret-shared when a secret sharing \((x)\) has been constructed.

Suppose that for a secret sharing \((x)\) the value \( x \) is to be revealed to server 1. This proceeds by server 2 sending the share \( x_2 \) and the tag \( m_2 \) to server 1. Server 1 then verifies whether \( m_2 = m_1 = \alpha_2 \cdot x_1 + \beta_2 \) and recovers \( x = x_1 + x_2 \). If the tag values are incorrect, the protocol is aborted. This is referred to as opening \((x)\) to server 1, and works symmetrically for server 2. A secret sharing can also be opened to a party that is external to servers 1 and 2, by sending the party all relevant shares, authentication tags and authentication keys, and letting this party do the corresponding verifications.

It is easy to see that any attempt by, for example, server 1 to correctly adjust a tag \( m_1 \) with key \((\alpha_2, \beta_2)\) during a malicious share modification from \( x_1 \) to the value \( x_1' \neq x_1 \), boils down to guessing the value \( \alpha_2 \cdot (x_1' - x_1) \), which only succeeds with probability \( 1/p \) if the value of \( \alpha_2 \) is unknown.

Although we only focus on two servers, the framework can cope with multiple servers. When \( n \) servers are available, the basic idea is that each server is given authentication keys for each of the \( n \) shares [1]. Intuitively, this allows an honest server to detect malicious behaviour of the remaining servers, so in SPDZ one honest server is sufficient for providing security. The quadratic amount of authentication keys has been improved lately by Damgård et al. [14], where tag authentication is linear in the number of players.

2) Structured-Authentication: As we demonstrate in the next section, it is possible to perform basic linear operations on secret-shared values without interaction. However, in order to
make it possible to non-interactively update the corresponding authentication keys and tags, we require both servers to fix their respective α-components of their authentication keys. Henceforth, each server $i$ maintains a fixed private value $α_i \in \mathbb{F}_p$ that is used in all of its authentication keys, and we use notation $[x]$ for a secret sharing of $x$ based on such α-priori fixed α-components.

Having an incorruptible external dealer, the servers can easily create such a structured distribution $[x]$ using the standard distribution $(x)$ obtained from the dealer. We describe the necessary steps for server 1; they are similar for server 2. All that is required is that for every $α_i$ received from the dealer, server 1 sends the difference $δ_i = α_i - α_{i-1}$ to server 2, after which server 2 updates his tag $m_2$ to $m_2 + δ_i \cdot x_2$. Since the values $α_1$ and $α_2$ are only known to server 1, $δ_i$ reveals no information about $α_1$ to server 2 [1].

C. Core Protocols for the Computation Phase

In this section we describe the core protocols that are needed for our recommender system. Suppose that servers 1 and 2 hold fixed partial authentication keys $α_1$, $α_2 \in \mathbb{F}_p$ as described in Subsection II-B.2.

1) Linear Operations: Let $[x]$ and $[y]$ be secret sharings of arbitrary values $x, y \in \mathbb{F}_p$, and let $c \in \mathbb{F}_p$ be a public constant. We will show how to non-interactively compute secret sharings for $[x+y], [cx]$ and $[x+c]$ respectively.

An authenticated secret sharing of the sum $z = x + y$ is computed by locally adding the shares, keys, and tags of $x$ and $y$. Let $z_i = x_i + y_i, β_i^x = β_i^x + β_i^y$ and $m_i = m_i^x + m_i^y$, for $i = 1, 2$, where $z_i, m_i^x, m_i^y$ are held by server $i$. Then indeed $z = z_1 + z_2 = x + y$, and the authentication tags $m_i^z$ of the shares of $z$ are correct, with authentication keys $(α_i, β_i^z)$.

From a sharing $[x]$, an authenticated secret sharing of $cx$ is computed by local multiplication of the shares, keys, and tags of $x$. Let $z_i = c \cdot x_i, β_i^z = c \cdot β_i^x$, and $m_i = c \cdot m_i^x$, for $i = 1, 2$. Then indeed $z = z_1 + z_2 = cx$, and the authentication tags $m_i^z$ of the shares of $z$ are correct.

To add a public constant to a secret sharing $[x]$, one party adds the constant to its share, and the other party adjusts its authentication key. Let $z_i = c + x_i, z_2 = x_2, β_i^z = β_i^x, β_i^z = β_i^z - c \cdot α_2, m_i^z = m_i^z + m_i^z$, and $m_i^z = m_i^z$. Then indeed $z = z_1 + z_2 = c + x$, and the authentication tags remain correct.

2) Multiplication: We have seen in the previous subsection that linear operations on secret-shared values can be handled non-interactively, including the updates of the corresponding authentication keys and tags. In this section we describe how to turn the multiplication of two secret-shared values into an operation that is essentially linear, using a precomputed multiplication triplet. Since the multiplication protocol below requires some secret sharings to be opened, the multiplication of two shared values requires interaction between the two servers. For clarity, we omit the updating of authentication keys and tags, and the details of opening a secret shared value, which have been described earlier.

Let $[x]$ and $[y]$ denote secret sharings of arbitrary values $x, y \in \mathbb{F}_p$, and let $([a], [b], [c])$ be a given multiplication triplet such that $c = ab$. The following sequence of local linear operations and interactions is used to compute a sharing $[z]$ where $z = xy$, making use of the precomputed multiplication triplet:

1) The servers locally compute the secret sharing $[v] = [x-a]$ from $[x]$ and $[a]$, and then open it towards each other.
2) The servers locally compute the secret sharing $[w] = [y-b]$ from $[y]$ and $[b]$, and then open it towards each other.
3) The servers locally compute the secret sharing $[z] = [xy] = w[a] + v[b] + [c] + vw$.

The correctness of the outcome of the multiplication protocol is easily verified by working out $xy = (v + a)(w + b)$.

III. THE SECURE CLIENT-SERVER MODEL

A. Overview

Our secure framework relies heavily on techniques from the cryptographic area of secure multi-party computation [19], [20]. The problem of secure multi-party computation considers a fully connected communication network of $n$ parties, where the parties wish to compute the outcome of a given function $f$ on their respective inputs. However, apart from the output, no information on the inputs should leak during the course of the computation. In this work we consider functions $f$ that correspond with the computation of a recommendation for a user.

The ideal solution to the problem of secure multi-party computation involves an independent, incorruptible mediator that privately takes all the required inputs, computes and reveals the desired output to the appropriate parties, and then forgets everything. The research area of secure multi-party computation studies techniques that allow the parties to simulate the behavior of such a mediator.

These simulations have the following structure. First, there is an input phase that enables the parties to encrypt their respective inputs for use in the secure computation. Then a computation phase takes place during which an encrypted output of the function $f$ is computed from the encrypted inputs. Finally, an output phase takes place where the output is decrypted, and sent to the appropriate parties.

We consider secure multi-party computation in the preprocessing model [21], where at some point in time prior to the selection of the inputs, a preprocessing phase takes place that establishes the distribution of an arbitrary amount of correlated data between the parties involved in the computation. Consequently, this data is completely independent of the input data of the parties. The goal of the preprocessing is to remove as much of the complexity and interaction from the actual computation as possible, which as a result makes this computation extremely efficient. We discuss possible ways to implement such a preprocessing phase in Section IV-A.

In theory, using techniques from secure multi-party computation, it is possible for the users of the recommender system to securely and jointly compute the recommendations themselves. However, for a large number of users that approach
quickly becomes impractical. In this work we instead let the users outsource the problem of multi-party computation to two dedicated servers that execute a number of two-party computations with preprocessing. Unlike in ordinary secure multi-party computation, the inputs are here provided by some external parties, and the outputs are also returned to external parties. We therefore require special techniques to correctly integrate these operations into the computation. Moreover, we design the system in such a manner that the users need only provide every input (and possible update) once, so they need not retransmit their current inputs for the computation of every individual recommendation.

B. Adversarial Behavior and Security

We consider an adversary that is able to take full control of one of the two servers involved in the two-party computation with preprocessing. Moreover, the adversary can initially introduce an arbitrary number of so-called dummy users into the system, which are also under his complete control. Dummy users can in particular, like ordinary users, input or update their ratings, and request recommendations. The adversary can read all data available to the entities that are under his control, and can make them deviate from the cryptographic protocol specification arbitrarily.

The goal is to prove that the actions of the adversary have essentially no impact on the outcome or the security of the protocol. In order to describe the security level of our secure recommendation system we make use of the real/ideal world paradigm [22].

This paradigm is based on the comparison between two scenarios. In the first scenario, the real world, the adversary participates in a normal protocol execution. In the second scenario, the ideal world, all participants in the protocol have black-box access to the functionality that the protocol in the real world attempts to emulate. Moreover, there exists a simulator that creates a virtual environment around the adversary, and interacts with the adversary in a special fashion.

The goal of the simulator in the ideal world is to simulate the expected world-view of the adversary during a real protocol execution. The security analysis now consists of showing that the adversary cannot distinguish whether it is acting during a protocol execution in the real world, or in the ideal world. Since no effective attack is possible in the ideal world, it then follows that no effective attack is possible in the real world either.

Our description of the real/ideal world paradigm so far has been generic. We now provide additional details for the real/ideal world setup for our specific setting of secure recommender systems. The real world corresponds with the model outlined in Section III-A, except that we additionally have an entity called the environment that fully controls the actions of the adversary and additionally chooses the inputs of the non-adversarial users.

The ideal world describes how we would ideally expect the recommendation processor $R$ to behave. The users in the ideal world have access to an independent incorruptible recommendation processor $T$ that privately takes all the required inputs and updates from the users and, upon request, privately returns the correct recommendations to the users. The two servers that are used to implement the processor $R$ in the real world do not exist in the ideal world.

Just like in the real world, the environment in the ideal world fully controls the actions of the adversary, and provides inputs to the non-adversarial users, except that all communication between the adversary and the non-adversarial entities (i.e., the non-dummy users and the non-existent second server of the real-world setting) is intercepted by the simulator. Specifically, the goal of the simulator is to simulate all communication between the two servers, and all communication from the non-adversarial server to the dummy users, as it would occur during a protocol execution in the real world.

To help with this simulation, the simulator has access to all data generated in the preprocessing phase for the two servers, and it may in fact even be assumed without loss of generality that it actually generates this data. Moreover, since in the ideal world non-adversarial users communicate directly with the ideal processor, this processor notifies the simulator whenever a non-adversarial user either provides it with an input, or requests a recommendation.

The level of security that our secure recommendation system achieves can now be described as follows. It is possible to design a simulator for our secure recommendation system in such a manner that regardless of what the environment (and therefore the adversary) does, it cannot significantly distinguish whether it is acting in the real world or in the ideal world. Since no meaningful attack is possible in the ideal world, the system is therefore secure in the real world.

Because a server is able to change his share of a particular secret-shared value, and correctly guess the updated tag value with probability $1/p$, perfect security is not achievable. The best security level is attained by having a trusted dealer perform the preprocessing phase, in which case the real world and the ideal world are statistically indistinguishable by the simulator, and our recommender system is statistically secure. When the preprocessing has been performed by public-key cryptography between the two servers, this will reduce to computationally indistinguishable worlds, and eventually a computationally secure recommender system.

Since our core protocols, as described in Section II-C, fall within the general framework of [1], we for these protocols refer to their proof of security in the malicious model. A key point in which our model differs is that all inputs originate from, and all output is returned to, external parties. We therefore include appropriate ideal world simulation details to our input (see Section IV-D) and output (see Section IV-E) subprotocols.

C. Our Approach

Every computation corresponds with a function $f$, which can be represented via an arithmetic circuit consisting of basic operations like addition and multiplication. For the recommender application we consider here, it suffices to consider these basic operations together with the more complex operations of comparison and integer division, which are composed of basic operations.
As mentioned earlier, as a result of the ‘outsourcing aspect’ of the secure computation, the input phase is non-standard in the sense that the inputs are not provided by the two servers. We nevertheless assume for now (and provide details on how to accomplish this in Section IV-D) that the input phase results in ‘encryptions’ of the inputs that are suitable for the computation phase of the two-party computation with preprocessing.

The encryption we use for the inputs is linear secret-sharing with authentication tags. To be more precise, for every encrypted input, both servers end up with a random value as their share. This is done in such a way that the shares together determine the value of the input, while they are individually statistically independent of the value of the input. Moreover, each of these shares is accompanied by an authentication tag that together with the share is linked to an authentication key that is held by the other server. As a result, the servers are committed to the values of their shares.

Although the users providing the inputs could in principle take care of the share distribution, these users cannot be trusted to provide the authentication keys and tags, since they might be under control of one of the servers. Moreover, we want to introduce additional structure to the authentication keys in order to enable our secure two-party computation approach, as explained in Subsection II-B2.

Once the encryption of the inputs has been established, the computation recursively handles the operations in the circuit while maintaining the encryption structure as an invariant. In other words, every operation in the circuit is initiated with two encrypted inputs, and produces an encrypted output without leaking any information on the encrypted values. Therefore, at the end of the circuit evaluation, an encryption of the final output becomes available.

The output phase is also non-standard, as the output needs to be revealed to an external party. Here the idea is that all data related to the encryption of the output is sent back to the relevant user, so that this user can verify the correctness of the shares using the authentication keys and tags, and then decrypt the output using the shares.

However, we need to recycle the fixed $a$-keys in order to prevent the users from having to resubmit their inputs for every recommendation computation. Therefore, the used authentication keys may never be sent to a (dummy) user, as this would break security for future recommendation computations. We provide our solution to this problem in Section IV-E.

IV. IMPLEMENTATION

In this section we describe how the recommendation application has been implemented, using the Bendlin et al. [1] framework. We mention the required secret sharing operations, and refer to Section II-B for the details on updating of authentication keys and tags.

A. Remarks on the Preprocessing Phase

In the preprocessing phase we expect the following resources to be computed:

1) “Multiplication triplets”, i.e., \( ([a], [b], [c]) \) with \( a, b, c \in \mathbb{F}_p \) selected uniformly at random and \( c = ab \).
2) Bitwise secret sharings \( \{r\}_B \) of random elements \( r \in \mathbb{F}_p \), see Subsection IV-B.
3) “Duplicate” secret sharings \( \{r\} \) and \( \langle r \rangle \) for random \( r \in \mathbb{F}_p \), see Appendix A.

Since each secure multiplication requires a precomputed multiplication triplet, the computation of these triplets takes the main effort of the preprocessing phase.

For all of these sharings it is assumed that the distribution is as if all shares, keys, and tags were first generated and distributed directly by an incorruptible external dealer, and then (where required) updated via server interaction as described at the end of Subsection II-B2.

There are many ways to implement preprocessing. In this section we discuss some methods that can be used to generate the random sharings and multiplication triplets. While some of these methods can also be used to generate the bitwise sharings and ‘duplicate’ sharings of random values, we describe in the Appendix a generic solution that allows creating the latter sharings using the former sharings.

The easiest way to implement preprocessing is to outsource the generation of triplets and random values to a trusted third party, who then computes the proper values and sends each server the corresponding shares and authentication tags. Although such a third party never sees any of the user data or other data related to the computation, it is still crucial for the security of the recommendation system that this party does not maliciously collaborate with any of servers. Although in theory also the online phase could be outsourced to this trusted third party, this would not only require the third party to be online continuously, but also give insight into sensitive user data, and increase the risk of data leakage.

Similarly, the preprocessing could be outsourced to multiple external parties for which more than two-thirds are believed to be honest. These parties can then perform information-theoretically secure multi-party computation [19], [20] to generate encryptions of the shares, authentication keys and tags for the required triplets and random values, and open them to the respective servers.

When no suitable external parties are available, the preprocessing can instead be taken care of by the two servers themselves using cryptography. This could for instance be implemented using Yao’s garbled circuit approach [23], semi-homomorphic encryption [1], or fully-homomorphic encryption [24]. Standard Yao’s garbled circuits are only secure in the semi-honest model, but recent cut-and-choose protocols [25], [26] enable implementing the preprocessing by garbled circuits in the malicious model with two parties.

Although jointly precomputing triplets requires a substantial amount of work due to the high computational costs involved with the use of heavy cryptographic techniques, the computations can easily be done in parallel by dedicated hardware. Recently, the precomputation effort from the original SPDZ framework [1] has been reduced by Damgård et al. [15], lowering the time needed for generating a multiplication triplet.
from 2.5 seconds to 0.02 seconds. When covert security is sufficient, this can be improved further to 0.0014 seconds.

Regardless of the approach that is chosen, the nature of the application allows for many triplets to be computed during inactive time, which can then be used to generate the recommendations efficiently in active time.

B. Secure Comparison

In this section we describe how to securely compute the outcome of the comparison \( x < y \), for any two elements \( x, y \in \mathbb{F}_p \), based on a solution by Nishide and Ohta [16]. This is essentially based on the following two observations.

1) Interval Comparison: The first observation is that it is sufficient to have a protocol that compares the secret-shared value \( z = (x - y) \mod p \) with \( \frac{2}{3} \), i.e., that determines \( \lfloor (z < \frac{2}{3}) \rfloor \) from \( z \). Although the original protocol by Nishide and Ohta also requires computing \( \lfloor (x < \frac{2}{3}) \rfloor \) and \( \lfloor (y < \frac{2}{3}) \rfloor \), we have chosen \( p \) in our implementation large enough such that we are assured that in every execution of the secure comparison protocol the inputs do not exceed \( \frac{2}{3} \), which reduces the execution time of the secure comparison protocol roughly by a factor three.

Given the shared interval comparison bit \( \lfloor (z < \frac{2}{3}) \rfloor \), the shared comparison bit \( \lfloor (x < y) \rfloor \) is easily derived through \( (x < y) = 1 - (z < \frac{2}{3}) \) [16].

2) Least Significant Bit Protocol: In this paragraph we frequently switch between integers (which are not reduced modulo \( p \)) and field elements, and use + for both types of additions.

Let \( \zeta \) be the integer \( 2^z \). The second observation is that if \( z > \frac{2}{3} \), then the integer \( \zeta \) will be larger than \( p \), so the field element \( 2^z \in \mathbb{F}_p \) will equal \( z - p \) and will therefore be odd. On the other hand, if \( z < \frac{2}{3} \), then the field element \( 2^z \in \mathbb{F}_p \) will equal the integer \( \zeta \), and will be even. In order to determine \( \lfloor (z < \frac{2}{3}) \rfloor \), it is therefore sufficient to determine the value of the least significant bit of the integer \( \zeta \).

In order to securely compute the least significant bit \( \zeta_0 \) of \( \zeta \), a precomputed bitwise secret sharing \( [r/s] \) is used to compute \( [s] = [\zeta + r] \), which is then opened towards both servers [16].

If the integer \( \zeta + r \) is smaller than \( p \), then \( \zeta_0 = s_0 \oplus r_0 \), where \( s_0 \) and \( r_0 \) are the least significant bits of \( s \) and \( r \), respectively. On the other hand, if the integer \( \zeta + r \) is at least \( p \), then \( \zeta_0 = 1 - (s_0 \oplus r_0) \). Since \( s \) is known to both parties, \( s_0 \oplus r_0 \) is easily computed as \( r_0 \), if \( s_0 = 0 \), and \( 1 - r_0 \), otherwise.

The remaining problem is therefore to determine whether the integer \( \zeta + r \) is smaller than \( p \). This is equivalent to determining the value of \( (s < r) \), because the field element \( s \) is smaller than the field element \( r \) exactly when the integer \( \zeta + r \) is smaller than \( p \).

The comparison of \( s \) and \( r \) can be completed by going through the bits of field elements \( s \) and \( r \), since the (secret-shared) bits of \( r \) are available. Let \( r_i, 0 \leq i < \ell \) be the \( \ell \) bits of \( r \). Then the following procedure can be used to securely compute \( [\delta] = [s < r] \):

1) If \( s_0 = 0 \) then \( [\delta] := [r_0] \) else \( [\delta] := [0] \).

The premise \( \delta = (s_0 < r_0) \) is now true.

2) For \( i = 1 \) to \( \ell - 1 \) do

If \( s_i = 0 \), then \( [\delta] := [r_i] + [1 - r_i] \cdot [\delta] \).

Otherwise, \( [\delta] := [r_i] \cdot [\delta] \).

The invariant condition \( \delta = (s_1 \ldots s_0 < r_1 \ldots r_0) \) is now true.

Given \( [\delta] \), as explained above \( \lfloor (z < \frac{2}{3}) \rfloor \) is easily derived as \( [\delta \oplus s_0 \oplus r_0] = [\delta] - [s_0 \oplus r_0] - 2[\delta] \cdot [s_0 \oplus r_0] \). Therefore, one secure comparison takes \( \ell \) rounds and \( \ell \) secure multiplications in total.

This linear round bitwise comparison protocol, similar to the circuit of Schoenmakers and Tuyls [17], computes \( [\delta] = ([s < r]) \) within \( \ell - 1 \) rounds, requiring only one multiplication per round. Constant round solutions by Damgård et al. [27], or Nishide and Ohta [16] offer only slight round improvements, requiring 19 and 15 rounds respectively, and on the other hand will increase drastically the number of secure multiplications (e.g., 2795 + 5 for [16]), leading to more communication and computations.

C. Integer Division

In our implementation of secure recommendations in Subsection IV-F we need a protocol for integer division between two secret-shared values. In a recent paper, Veugen et al. [28] demonstrated integer division in a setting with additive homomorphic encryption, and public or privately held divisors. Constant-round protocols based on secret sharing for integer division are also available [27]. However, since these make use of a large amount of precomputed data, and since the prime \( p \) remains relatively small in our application, we instead make use of a simple protocol based on long division.

Let \( x, y, z \in \mathbb{Z}_p \) be such that \( z = y \div x \), where the symbol \( \div \) is used to denote integer division without remainder. Moreover, suppose that \( z_{n-1}, \ldots, z_0 \) are the \( n \) bits of \( z \) for some \( n \), \( 0 < n < \ell \). The following algorithm, inspired by [4], will compute the values \( z_{n-1}, \ldots, z_0 \) one by one.

1) \( \psi := y; \ z := 0; \)
2) For \( i = 0 \) downto \( 0 \) do
   a) \( \zeta := 2^i \cdot x; \)
   b) \( z_i := ([\zeta] \leq \psi) \);
   c) \( z := z_i + 2^i \cdot z; \)

   The invariant \( z = \sum_{j=0}^{i-1} z_j 2^j \) is now true.

   d) \( \psi := \psi - z_i \cdot \zeta; \)

   The invariant conditions \( y = \psi + x \cdot z \) and \( \psi < 2^i x \) are now true.

Because of the invariant conditions, we end up with \( y < x \) and \( y = \psi + x \cdot z \), so \( z \) will be the result of the integer division and \( \psi \) will equal the remainder. Although the algorithm for integer division is described on integers here, all listed operations can equivalently be performed on values that are secret-shared modulo \( p \). The algorithm requires \( n \) rounds with one secure comparison (and one additional secure multiplication) per round, but in our application \( z \) represents a rating, so \( n \) will be small.

From a complexity point of view, the integer division protocol will form the bottleneck of our implementation. As a sidenote, observe that \( z = 1 \ldots 1 = 2^n - 1 \) whenever \( x = 0 \), so we could even cope with the case of zero similar users, if we require ratings to be limited to \( 2^n - 2 \).
D. The Input Phase

Although our framework allows clients to upload any values to the processor, we focus on our recommender application (see Subsection II-A). Initially, each user \( n \) \((1 \leq n \leq N)\) will have to upload his ratings \( V_{(n,m)} \) \((1 \leq m \leq M)\) once, before recommendations can be requested. A user could easily act as a dealer by splitting his rating into two shares, and sending each server a share accompanied by proper authentication tags. However, since some users may collude with one of the servers, it is easy to see that any solution along these lines will leak the private authentication keys \( a_1 \) and \( a_2 \) to the adversary.

Therefore, a small additional protocol is needed for uploading user ratings. We assume the servers have a precomputed pair of random secret sharings \([r]\) and \([r]\), as described in Appendix A, leading to the following protocol for uploading rating \( V_{(n,m)} \).

1) The servers open \([r]\) towards user \( n \) by sending him the corresponding shares, message authentication tags, and message authentication keys.

2) After verifying the correctness of the tags, user \( n \) sends the value \( (V_{(n,m)} - r) \in \mathbb{F}_p \) to both servers.

3) After verifying that they both received the same value, the servers compute \( [V_{(n,m)}] = [r] + (V_{(n,m)} - r) \) as described in Subsection II-C1.

Only user \( n \) will learn \( r \), which he uses to blind his rating. By adding the blinded value to \([r]\), the servers obtain the required sharing with their regular authentication keys. The user will only learn the random keys \((\alpha', \beta)\), so the private keys \( a_i \) remain secure.

1) Ideal World Simulation: In this section we describe the behavior of the simulator during the input phase of the ideal world. First we point out that, due to knowledge of all preprocessed data, the simulator has access to all shares and authentication keys that are held, or have been computed, by the adversarial server.

Let \([r]\) and \([r]\) again be a precomputed pair as is used in the real world input protocol. We need to distinguish between two different scenarios, namely one where an honest (non-adversarial) user provides an input, and one where a dummy user provides an input. First, suppose an honest user provides the input. In that case the input is delivered straight to the ideal processor \( I \), which implies that the simulator has no knowledge of the value of this input. Upon being notified by the processor of such an input event, the simulator selects a random (or default) value \( x \in \mathbb{F}_p \), and then acts on behalf of the honest user and the non-existent honest server towards the adversarial server, via the following steps.

1) The adversarial server sends the simulator its shares, message authentication tags, and message authentication keys corresponding to \([r]\).

2) The simulator verifies that the three values sent by the adversarial server are correct, and aborts otherwise.

3) The dishonest server performs the verification of the value \( x - r \) with the simulator (which again emulates the honest server) as it would in the real world, and the servers compute \([x] = [r] + [x - r]\).

Now suppose that the input is provided by a dummy user. The simulator then acts as follows.

1) The adversarial server and the simulator open \([r]\) towards the dummy user.

2) The dummy user sends a value \( y \in \mathbb{F}_p \) to the dishonest server and the simulator, who verify that they have received the same value, using the same method as is used in the real world.

3) If the adversarial server and the simulator received the same value, they compute the sharing \([r + y]\).

4) Using knowledge of the shares of the adversarial server, the simulator now determines the input \( x = r + y \) of the dummy user. The simulator then inputs the value \( x \) to the ideal processor \( I \) on behalf of the dummy user.

It is straightforward to verify that the view of the adversarial server in the ideal world now corresponds with that in the real world.

2) Input Checking: To further reduce the possibilities for malicious adversaries, we could easily check the size of user inputs. Given the shared uploaded value \([x]\), we could securely compute the secret-shared bit \([(x < 2^q)]\), and open it. The secure comparison protocol, as described in Subsection IV-B could be used for this end, but since we are not assured that \( x < p/2 \), we would have to run the ‘least significant bit’ protocol twice, namely for \( x \) and \( z = (x - 2^q) \mod p \), and combine the results as shown by Nishide and Ohta [16].

E. The Output Phase

Although our framework allows clients to download any value from the processor, we focus on our recommender application (see Subsection II-A). After a recommendation has been computed, the outputs \( \text{Rec}_{m} \in \mathbb{F}_q \) \((S < m \leq M)\) have to be sent to the requesting user. Since we want the user to be able to verify the correctness of the output, but we do not want to reveal the private authentication keys \( a_i \) to him, we need an additional protocol.

Assuming the servers have access to a precomputed pair of random sharings \([r]\) and \([r]\) (see Appendix A), we have the following protocol for downloading the estimated rating \( \text{Rec}_{m} \).

1) The servers compute \([\text{Rec}_{m} - r]\) = \([\text{Rec}_{m}] - [r]\), and open the resulting secret sharing towards each other by revealing the shares and authentication tags.

2) After checking the correctness of the tags, the servers compute \( (\text{Rec}_{m} = (r) + (\text{Rec}_{m} - r) \). This is an addition of a public constant to a sharing, but using completely random authentication keys \((\alpha', \beta)\).

3) The servers open \( (\text{Rec}_{m}) \) to the user. They also send him the corresponding tags and \((\alpha', \beta)\) keys, so he can check the correctness of his output.

Again, the user will only learn the random keys \((\alpha', \beta)\), so the private keys \( a_i \) remain secure.

1) Ideal World Simulation: In this section we describe the behavior of the simulator during the output phase in the ideal world. We remind the reader that the simulator has access to all shares and authentication keys that are held or have been
computed by the adversarial server, and all inputs provided by the dummy users. However, the simulator has no knowledge of the inputs of the honest users since these are directly submitted to the ideal processor \( I \).

We again distinguish between the request of an honest user and a dummy user. Suppose first that the request comes from an honest user. The simulator then acts as follows.

1) It first emulates the first two protocol steps in the real world towards the adversarial server. This results in a sharing \( \langle x \rangle \).
2) The adversarial server then sends the simulator its share, key and tag corresponding to \( \langle x \rangle \). If the adversarial server sends an incorrect value, the simulator aborts.

Now suppose that the output request comes from a dummy user. The simulator then needs to provide the dummy user with the correct recommendation output. This is done via the following steps.

1) The simulator emulates the first two protocol steps in the real world towards the adversarial server. This results in a sharing \( \langle x \rangle \).
2) The simulator now requests the correct estimated rating \( y \) from the ideal processor \( I \) on behalf of the dummy user.
3) The simulator uses the knowledge of the shares and authentication keys of the dishonest server to locally modify its share and tag in the sharing \( \langle x \rangle \), thus transforming it into a sharing \( \langle y \rangle \).
4) The adversarial server and the simulator now open the sharing \( \langle y \rangle \) towards the dummy user as they would in the real world.

It is again straightforward to verify that the views of the adversarial server and the dummy user in the ideal world correspond with that in the real world.

F. Application: Secure Recommendations

Using the approach described in Subsection III-C, and the basic operations described in Subsection II-C, we can securely compute the example application detailed in Subsection II-A. Based on the size and number of the ratings and the desired security level, we have decided to use a 20-bit prime number \( p \) for the implementation. This for instance implies that authentication tags can only be forged with probability \( 2^{-20} \). The parameters for the number of items were set to \( M = 1000 \) and \( S = 20 \), just as in [3], and we used ratings consisting of \( v = 4 \) bits. Consequently, the maximal size of a similarity value is \( 2^{20} - 1 \approx 4500 \) which will never exceed \( p/2 \), justifying the input requirement of the comparison protocol (see Subsection IV-B). We assume that on average 3% of the users rated a particular item \( m \), \( S < m \leq M \), which is similar to the EachMovie dataset used in [13].

Assuming all required values have been precomputed (see Section IV-A), the on-line phase of such a secure recommendation computation will consist of the following five steps, which are the secured versions of the steps described in Subsection II-A.

1) Initially, each user will have to upload his (at most) \( M \) ratings, so both servers will obtain authenticated secret sharings \( \{V_{(n,m)}\} \) (see Section IV-D). After the uploading, the servers will know \( U_m \) for each \( m \), \( S < m \leq M \), specifying the users that rated item \( m \).
2) When user A asks for a recommendation, the servers will compute similarities \( \{\text{Sim}_{A,n}\} \) for each (other) user \( n \). The computation of each similarity value takes \( S \) multiplications, and \( S - 1 \) local additions.
3) Next, each similarity value is compared with a public threshold \( t \), leading to \( N - 1 \) shared bits \( \{\delta_n\} \) (see Subsection IV-B). Each comparison will cost \( \ell \) secure multiplications and some local additions.
4) The shared comparison bits can be used to compute the actual estimated rating for each item \( m \), \( S < m \leq M \). For item \( m \), the computation of the cumulative sum of ratings of similar users requires \( |U_m| - 1 \) multiplications and \( |U_m| - 2 \) additions. The computation of the number of similar users requires \( |U_m| - 2 \) additions. As a final step, for each item these two have to be divided as shown in Section IV-C. This integer division protocol requires \( v \) secure comparisons, and a total per item of \( v(\ell + 1) \) multiplications, and some additions.
5) Once the estimated ratings \( \{\text{Rec}_n\} \), \( S < m \leq M \), have been computed, they have to be downloaded to user A (see Section IV-E). We need \( \sum_{\delta \in U_m} \delta_n \cdot V_{(n,m)} < \frac{p}{2} \) to correctly perform the integer division, but in practice only a small portion of users will be similar to user A, and the user-item matrix will be sparsely filled.

V. COMPLEXITY ANALYSIS

We analyse the computation, communication, storage, and precomputation complexity of computing recommendations in our framework. We did not add extra time due to network latency during the protocol simulation, just as in [3], but we counted the number of communication rounds, which is a good measure for network latency. We distinguished between the initial step in which the ratings are uploaded, and the on-line recommendation: steps 2, 3, and 4.

A. Computation Complexity

The application was implemented in C, using Visual Studio 2012 on a 64-bits Windows 7 laptop with a 2.5 GHz processor and 4 GB of memory. Figure 1 shows the computation time in seconds for a varying number of users. We did not use parallel computing, so in practice the execution time for two servers will be approximately half as much. The time for uploading the user ratings could even be parallelized amongst all users which would reduce the uploading time by a factor \( \frac{3}{N} \).

For 10,000 users, we need 0.34 seconds to compute one recommendation, excluding precomputation time.

B. Communication Complexity

Figure 2 shows the amount of numbers of size \( p \) that have to be sent during the on-line phase. For 10,000 users and a 20 bit
Fig. 1. Amount of on-line computation.

Fig. 2. Amount of on-line communication.

TABLE I

<table>
<thead>
<tr>
<th>Protocol step</th>
<th>number of comm. rounds</th>
</tr>
</thead>
<tbody>
<tr>
<td>1) Uploading the ratings</td>
<td>1</td>
</tr>
<tr>
<td>2) Computing the similarities</td>
<td>1</td>
</tr>
<tr>
<td>3) Secure threshold comparison</td>
<td>$v \cdot \ell + 1$</td>
</tr>
<tr>
<td>4) Integer division</td>
<td>2</td>
</tr>
<tr>
<td>5) Output to user</td>
<td></td>
</tr>
</tbody>
</table>

size $p$ this leads to a total of 35 MB of communication. The initial uploading of the ratings takes a relatively large amount of communication.

Besides the amount of communicated information, the number of communication rounds is also important for determining the time needed for communication, because each round introduces an extra delay depending on the communication technology used. Table I depicts the number of communication rounds per protocol step.

Given that $p$ consists of $\ell = 20$ bits, and each rating consists of $v = 4$ bits, the total number of communication rounds is 105. Although many computations can be parallelized, the integer division protocol, used for averaging the rating of a recommended item over all similar users, clearly requires the highest number of rounds.

C. Storage Complexity

The servers mainly need to store the initial ratings of all users, and the numbers precomputed for performing all calculations, for which we count the number of required triplets and bitwise shared random values. Given that each secure comparison needs $\ell$ secure multiplications, we derive the number of secure multiplications (see Table II), and consequently the number of required triplets, for our entire protocol. We used here that on average 3% of the users rated each item.

Given that $\ell = 20$, $M = 1000$, $S = 20$, and $v = 4$, our protocol requires $69.4(N - 1) + 82,320$ triples for one recommendation. To store one triple, each server needs to store 3 shares, 3 tags and 3 keys, so $9\ell$ bits in total.

For each secure comparison, the servers additionally need one bitwise shared random value. Each bitwise shared random value consists for each server of $\ell$ shares, $\ell$ tags and $\ell$ keys. The protocol consists of one secure comparison per user, and $v(M - S)$ for the final integer division.

Thus, for computing one recommendation, each server needs to store $(69.4(N - 1) + 82,320) \cdot 9\ell + (N - 1 + v(M - S)) \cdot 3\ell^2$ precomputed bits, and additionally $N \cdot (S + 0.03(M - S)) \cdot 3\ell^2$ bits to store the initial ratings. For 10,000 users this amounts to 22 MB per server.

D. Precomputation Complexity

In practice the precomputation effort is dominated by the generation of multiplication triplets. As described in Section IV-A, the easiest way to obtain all precomputed data is to involve an honest third party. This minimizes the required amount of communication and computation during the offline precomputation phase to a level that is at most that of the online phase.

We assume such a third party is not available, so the servers themselves jointly precompute all required triplets and sharings. In their implementation Bendlin et al. [1] require 2 to 3 seconds to precompute one triple, in a way that is provably secure in the malicious model. This precomputation effort has recently been reduced by Damgård et al. [15] to 0.02 seconds per triple (with a 32-bits prime). Based on their estimation, the precomputation time for our protocol would be roughly $69.4 \cdot 0.02 = 1.388$ seconds per user. However, for commercial purposes, dedicated and highly parallelized hardware can be acquired to substantially reduce the time needed.

Even on standard hardware, $69.4 \cdot 0.0014 = 0.097$ seconds is
achievable, when covert security is sufficient, leading to only 16 minutes for 10,000 users.

E. Comparison With Related Work

The only work on recommender systems, the authors know is provably secure in the malicious model, is Canny’s paper [13]. Because they use a different matrix-based type of collaborative filtering, within a peer-to-peer environment, a fair overall comparison is difficult, but we mention the main differences:

- Although both use collaborative filtering, they use an iterative Singular Value Decomposition algorithm, while we use user-neighborhood-based rating prediction, which in general is better able to cope with dynamic users, but provides less accurate recommendations [18].
- They finetuned their algorithm to the encrypted domain, such that only vector summations of encrypted data are needed, while we also allow more complicated operations like secure comparison and integer division.
- Their computations are performed by the users, half of which have to be honest, and only static adversaries are allowed. In our system, the computations are performed by the servers, where only one needs to be honest, and adversaries can be dynamic.
- Having users jointly perform the computations, and having the private key secret-shared among all clients, makes their solution less scalable. On the other hand, we have to find a second independent server that is assured not to collude with the first server.
- They estimated the time for computing one recommendation in a setting with 74,442 users at 15 hours. This number is based on a benchmark from 2002 on 1024-bit ElGamal. Although computer power has increased since then, a key length of 2048 would be more appropriate now, yielding a estimated running time of 3 hours in 2014. Time for generating keys and random coins was not included. Our time for producing one recommendation with 10,000 users is 0.34 seconds, excluding four hours of precomputing.
- Although users can be untrusted, they assume a secure blackboard, where users can upload values, which can be read by other users but not changed. We assume the existence of at least one honest server.
- They assume the existence of a trusted source that can produce many random coin tosses, and optionally also generate the keys, both of which are needed in the cryptographic protocols. We assume multiplication triplets and bitwise-shared random numbers have been precomputed, but we do not need an additional trusted source for this end.
- They allow some amount of information leakage. An aggregated model matrix $A$ is published, and used by the users in their computations. We do not allow information leakage.
- They need 4 GB, of data communication per client, where we need 35 MB in total for 10,000 users, which is considerably less.
- They estimate a 10-50 MB storage per client, where we require 44 MB in total for 10,000 users to be stored by the two servers, which is considerably less.

Although the security model is different, our recommender system is more similar to Erkin et al.’s [3], and we also compare our experimental results with theirs. They used the GMP library on a single computer with a 2.33 GHz processor and 16 GB memory. Because our numbers are smaller, we did not need GMP, and we had only 4GB of memory available with a 2.5 GHz processor. They did not cope with unrated items while computing the number of similar users, leading to less accurate rate predictions, where we allowed the servers to learn the sets $U_m$. It took us 0.34 seconds to compute a recommendation for 10,000 users, excluding four hours of precomputation (or 16 minutes with covert security), while they needed 50 minutes. They needed a total communication amount of 147 MB, somewhat more than our 35 MB. Also, we needed only 105 communication rounds while their implementation required at least 1024 communication rounds due to packing. In our setting, each server has to store 22 MB for 10,000 users, and their service provider needs 112 MB for storing all user data, and an additional 35 MB for computing a recommendation.

The differences with respect to security are more significant. In [3], a user requesting a recommendation will learn the amount of similar users. The reason is that this approach avoids having to perform a secure integer division protocol which is quite expensive. We mitigated that threat by implementing a secure integer division protocol. The most important difference is that their protocol is secure in the semi-honest model, while ours is secure in the malicious model. In our recommender system, the user is moreover assured that his outputs, the predicted ratings, are correctly computed.

Although [3] also requires some form of precomputation, precomputation is an essential part of our solution, and takes a substantial amount of time. On the other hand, in the covert security model, which is still an improvement w.r.t. the semi-honest model, our total execution time becomes favorable.

VI. Conclusions

We provide a general framework for outsourcing ongoing computations, which is suitable for dealing with a large number of users, and is provably secure in the malicious model. This framework is then applied to the problem of secure recommendation and, given a sufficient amount of precomputed data, leads to extremely efficient implementations.

The approach is very generic and easily allows for variations. Although we only consider the case of two servers, our model described in Section III is easily extended to an arbitrary number of servers. As a consequence, many other secure applications are possible using our framework.

APPENDIX

A. Generating Duplicate Sharings of a Random Secret

For the uploading of user ratings, and downloading of estimated ratings, we need to precompute “duplicate” sharings $\langle r \rangle$ and $\langle r \rangle$ for random elements $r \in \mathbb{F}_p$. To avoid leakage of
the private $\alpha$’s, for each sharing $\langle r \rangle$ the servers should use fresh random $\alpha$-values in their authentication keys that are independent of their private fixed values for $\alpha_1$ and $\alpha_2$.

Generating a sharing $\langle r \rangle$ of a fresh random number $r \in \mathbb{F}_p$ is described in [1], resulting in two random shares $r_1$ and $r_2$, one for each server. A new sharing $\langle r \rangle$ of the same random number $r$, but with different authentication keys, is created by first converting the two shares of $r$ into shared secrets, and then computing the new tags from fresh random authentication keys.

1) The servers generate a fresh random sharing $\langle s \rangle$.
2) Server 1 transmits $r_1 - s$ and its tag $m_1' - m_1$ to server 2. Simultaneously, server 2 sends $r_2$ and $m_2'$ to server 1.
3) The servers verify the tags and, if these are correct, both determine $r_1 - s = (r_1 - s) - s_2$.
4) The servers obtain a sharing $\langle r_1 \rangle$ of the secret share $r_1$ via a (local) computation of $\langle s \rangle + (r_1 - s)$.
5) By repeating the first four steps, the servers similarly obtain a sharing $\langle r_2 \rangle$ of the secret share $r_2$.
6) The servers generate fresh random sharings $[\alpha'_1], [\alpha'_2], [\beta'_1]$ and $[\beta'_2]$ that will serve as new authentication keys.
7) The servers securely compute $m_1'' = [r_1 \cdot [\alpha'_2 + \beta'_2]$.
8) The servers open $\alpha'_2$ and $\beta'_2$ to server 2, and open $m_1'$ to server 1. By verifying the tags, the correctness of the opened values is checked.
9) By repeating steps 7 and 8, server 1 will obtain $\alpha'_1$ and $\beta'_1$, and server 2 $m_2''$.

It follows that $m_1 = r_1 \cdot \alpha'_2 + \beta'_2$ and $m_2 = r_2 \cdot \alpha'_1 + \beta'_1$, so indeed we created different authentication keys for the same shared value $r = r_1 + r_2$, and thus a second sharing $\langle r \rangle$ with independent authentication keys.

REFERENCES


Thijs Veugen received two M.Sc. degrees in mathematics and computer science (both cum laude) and the Ph.D. degree in information theory from the Eindhoven University of Technology Eindhoven, The Netherlands. He was a Scientific Software Engineer with Statistics Netherlands, Heerlen, The Netherlands. Since 1999, he has been a Senior Scientist with the Information Security Research Group, Department of Technical Sciences, TNO, Delft, The Netherlands. He is currently a Senior Researcher with the Cyber Security Group, Delft University of Technology, Delft, and has specialized in applications of cryptography. He has written many scientific papers on computing with encrypted data, and serves frequently as a member of the program committee board of information security related conferences, and holds numerous related patents in various countries.
Robbert de Haan received M.S. degrees in computer science and mathematics from the University of Amsterdam, Amsterdam, The Netherlands, in 2003 and 2004, respectively, and the Ph.D. degree in mathematics from Leiden University, Leiden, The Netherlands, in 2009, based on cryptologic research done at the Centrum Wiskunde and Informatica, Amsterdam. His research interests include cryptology and information security.

Ronald Cramer received the Ph.D. degree in cryptology from the University of Amsterdam, Amsterdam, The Netherlands, in 1997, and the M.Sc. degree in pure mathematics from Leiden University, Leiden, The Netherlands, in 1992. Since 2004, he has been the head and founder of the Cryptology Group with Centrum Wiskunde and Informatica, Amsterdam, The Netherlands, which is the Dutch National Research Center for Mathematics and Computer Science. Since 2004, he has been a Full Professor with the Mathematical Institute, Leiden University, Leiden, The Netherlands. Prior to that, he held research positions with the Swiss Federal Institute of Technology in Zurich, Switzerland, and Aarhus University, Aarhus, Denmark. His research areas include cryptology and applications of algebraic number theory and algebraic geometry. He is Member of the Royal Netherlands Academy of Arts and Sciences and Fellow of the International Association for Cryptologic Research.

Frank Muller received the M.Sc. degree in electrical engineering from Delft University of Technology, Delft, The Netherlands. He is currently a Senior Scientist with TNO, Delft, The Netherlands. His research interests include cryptographic protocols, payment systems, and mobile network security.