Int. J. Inf. Secur.
DOI 10.1007/s10207-016-0350-0

REGULAR CONTRIBUTION

Secure computation of hidden Markov models and secure floating-point arithmetic in the malicious model

Mehrdad Aliasgari[1] · Marina Blanton[2] · Fattaneh Bayatbabolghani[2]

Abstract Hidden Markov model (HMM) is a popular statistical tool with a large number of applications in pattern recognition. In some of these applications, such as speaker recognition, the computation involves personal data that can identify individuals and must be protected. We thus treat the problem of designing privacy-preserving techniques for HMM and companion Gaussian mixture model computation suitable for use in speaker recognition and other applications. We provide secure solutions for both two-party and multi-party computation models and both semi-honest and malicious settings. In the two-party setting, the server does not have access in the clear to either the user-based HMM or user input (i.e., current observations) and thus the computation is based on threshold homomorphic encryption, while the multi-party setting uses threshold linear secret sharing as the underlying data protection mechanism. All solutions use floating-point arithmetic, which allows us to achieve high accuracy and provable security guarantees, while maintaining reasonable performance. A substantial part of this work is dedicated to building secure protocols for floating-point operations in the two-party setting, which are of independent interest.

✉ Marina Blanton
mblanton@nd.edu

Mehrdad Aliasgari
mehrdad.aliasgari@csulb.edu

Fattaneh Bayatbabolghani
fbayatba@nd.edu

[1] Department of Computer Engineering and Computer Science, California State University, Long Beach, CA, USA

[2] Department of Computer Science and Engineering, University of Notre Dame, Notre Dame, IN, USA

Keywords Secure computation · Floating point · Hidden Markov models · Gaussian mixture models

1 Introduction

1.1 Motivation

Hidden Markov models (HMMs) have been an invaluable and widely used tool in the area of pattern recognition. They have applications in bioinformatics, credit card fraud detection, intrusion detection, communication networks, machine translation, cryptanalysis, robotics, and many other areas. An HMM is a powerful statistical tool for modeling sequences that can be characterized by an underlying Markov process with unobserved (or hidden) states, but visible outcomes. One important application of HMMs is voice recognition, which includes both speech and speaker recognition. For both, HMMs are the most common and accurate approach, and we use this application as a running example that guides the computation and security model for this work.

When an HMM is used for the purpose of speaker recognition, usually one party supplies a voice sample and the other party holds a description of an HMM that represents how a particular individual speaks and processes the voice sample using its model and the corresponding HMM algorithms. Security issues arise in this context because one's voice sample and HMMs are valuable personal information that must be protected. In particular, a server that stores hidden Markov models for users is in possession of sensitive biometric data, which, once leaked to insiders or outsiders, can be used to impersonate the users. For that reason, it is desirable to minimize exposure of voice samples and HMMs corresponding to individuals when such data are being used for authentication or other purposes. To this end, in this work we design solutions for securely performing computation on HMMs in such a way that no information about private data is revealed as a result of execution other than the agreed upon output. This immediately implies privacy-preserving techniques for speaker recognition as well as other applications of HMMs.

In more detail, in the speaker recognition application the overall process consists of two phases: (i) feature extraction in the form of creating an HMM and (ii) speaker authentication in the form of evaluating an HMM. The same two phases would need to be executed in other applications as well. Feature extraction constitutes a one-time enrollment process, during which information about how user $\mathcal{U}$ speaks is extracted and privately stored at a server or servers that will later authenticate the user (i.e., the HMM is not available to the servers in the clear and prevents leakage of user information and consequently user impersonation by unauthorized parties). At the time of user authentication, an individual $\mathcal{U}'$ wanting to gain access to the system as user $\mathcal{U}$ engages in privacy-preserving user authentication by evaluating a voice sample that $\mathcal{U}'$ supplies on $\mathcal{U}$'s HMM that the server stores. This takes the form of a secure protocol run between the client $\mathcal{U}'$ and the server. Note that user authentication can never be performed locally by the client because $\mathcal{U}'$ can always return the desired value as the final outcome to the server.

1.2 Our contributions

There are three different types of problems and corresponding algorithms for HMM computation: the forward algorithm, the Viterbi algorithm, and the expectation maximization (EM) algorithm. Because the Viterbi algorithm is most commonly used in voice recognition, we provide a privacy-preserving solution for that algorithm, but the techniques can be used to securely execute other HMM algorithms as well. Furthermore, to ensure that Gaussian mixture models (GMMs), which are commonly used in HMM computation, can be part of secure computation as well, we integrate GMM computation in our privacy-preserving solutions.

One significant difference between our and prior work on secure HMM computation is that we develop techniques for computation on floating-point numbers, which provide adequate precision and are most appropriate for HMM computation. We also do not compromise on security, and all of the techniques we develop are provably secure under standard and rigorous security models, while at the same time providing reasonable performance (we implement the techniques and experimentally show performance in the semi-honest setting).

To cover as wide of a range of application scenarios as possible, we consider multiple settings: (i) the two-party setting in which a client interacts with a server and (ii) the multi-party setting in which the computation is carried out by $n > 2$ parties, which is suitable for collaborative computation by several participants as well as secure outsourcing of HMM computation to multiple servers by one or more computationally limited clients. In the two-party setting, the server should have no access in the clear to either the user-based (private) HMM or user input (i.e., current observations) and thus the server stores the encrypted HMM and computation proceeds on encrypted data (see Sect. 4.1 for justification of this setup). In the multi-party setting, on the other hand, threshold linear secret sharing is employed as the underlying mechanism for privacy-preserving computation.

We provide techniques for both semi-honest (also known as honest-but-curious or passive) and malicious (also known as active) security models using secure floating-point operations from [2]. Because [2] treats only the semi-honest setting, equivalent solution secure in the stronger malicious model are not available for the two-party case. We thus develop necessary protocols to support general secure floating-point operations in the two-party computation setting based on homomorphic encryption. These protocols have applicability well beyond the HMM domain treated in this work. Their rigorous simulation-based proofs of security are the most challenging part of this work and its distinct and substantial contribution. Note that such proofs were not provided for the equivalent building blocks secure in the semi-honest setting and had to be constructed from scratch.

To summarize, our contributions consist of developing provably secure HMM and GMM computation techniques based on Viterbi algorithm using floating-point arithmetic. Our techniques are suitable for two-party and multi-party computation in a variety of settings and are designed with their efficiency in mind, which we evaluate through experimental results of an implementation. A significant part of this work is dedicated to secure floating-point operations in the malicious model in the two-party setting to support HMM computation (and a large number of other applications that use floating-point arithmetic) in the malicious model. We rigorously prove security of the floating-point operations using simulation-based proofs, and these protocols should be treated as a major contribution of this work.

Preliminary version of this work appeared in [1]. Some aspects of [1] (such as description of integer building blocks) and some results from [1] (such as improved floating-point product computation) are omitted from this article because of space considerations. Most of the content of this publication, on the other hand, is new and has not appeared in [1].

1.3 Paper organization

The rest of this work is organized as follows: We first review related literature in Sect. 2 and provide background information regarding HMMs and GMMs in Sect. 3. In Sect. 4, we describe our framework covering both two-party and multi-party settings and the security model. We then present the building blocks in Sect. 5 and describe our overall solution in the semi-honest model in Sect. 6. Section 7 reports on the results of our implementation, and in Sect. 8 we present new techniques to enable secure execution of our solution in the malicious setting. Lastly, Sect. 9 concludes this work.

2 Related work

To the best of our knowledge, privacy-preserving HMM computation was first considered in [40], which provides a secure two-party solution for speech recognition using homomorphic encryption and integer representation of values. In general, integer representation is not sufficient for HMM computation because it involves various operations on probability values, which occupy a large range of real numbers and demand high precision. In particular, probabilities need to be repeatedly multiplied during HMM computation, and the resulting product can quickly diminish with each multiplication, leading to inability to maintain precision using integer or fixed-point representation. Smaragdis and Shashanka [40] compute this product using logarithms of the values, which becomes the sum of logarithms (called logsum in [40]). This allows the solution to retain some precision even with (scaled) integer representation, but the computation was nevertheless not shown to be computationally stable and the error was not quantified. Also, as was mentioned in [17], one of the building blocks in [40] is not secure.

The techniques of [40] were later used as is in [39] for Gaussian mixture models. The same idea was used in [30,33] to develop privacy-preserving speaker verification for joint two-party computation, where the HMM parameters were stored in an encrypted domain. Also, [32] treats speaker authentication and identification and speech recognition in the same setting. Similar to [31,39] aimed at providing secure two-party GMM computation using the same high-level idea, but with implementation differences. The solution of [31], however, has security weaknesses. In particular, the protocol reveals a non-trivial amount of information about the private inputs, which, in combination with other computation or outside knowledge, may allow for full recovery of the inputs (we provide additional detail about this security weakness in [1]). Some of the above techniques were also used in privacy-preserving network analysis and anomaly detection in two-party or multi-party computation [26,27].

Another work [35] builds a privacy-preserving protocol for HMM computation in the two-party setting using a third-party commodity server to aid the computation. In [35], one participant owns the model and the other holds observations. We build a more general solution that can be applied to both two-party (without an additional server) and multi-party settings, uses high-precision floating-point arithmetic, and is secure in a stronger security setting (in the presence of malicious participants).

All of the above work uses integer-based representations, where in many cases multiplications were replaced with additions of logarithms, as originated in [40]. With the exception of [32], these publications did not quantify the error, while using integer (or fixed-point) representation demands substantially larger bit length representation than could be used otherwise and the error can accumulate and introduce fatal inaccuracies. Pathak et al. [32] evaluated the error and report that it amounted to 0.52 % for their specific set of parameters.

The need to use non-integer representation for HMM computation was recognized in [18] and the authors proposed solutions for secure HMM forward algorithm computation in the two-party setting using logarithmic representation of real numbers. The solution that uses logarithmic representation was shown to be accurate for HMM computation used in bioinformatics (see [17]), but it still has its limitations. In particular, the look-up tables used in [18] to implement certain operations in logarithmic representations grow exponentially in the bitlength of the operands. This means that the approach might not be suitable for some HMM applications or a set of parameters. The use of floating-point numbers, on the other hand, allows one to avoid the difficulties mentioned above and provides a universal solution that works for any application with a bounded (and controlled) error. Thus, in this work we address the need to develop secure computation techniques for HMMs on standard real number representations and provide the first provably secure floating-point solution for HMM algorithms, which initially appeared in [1].

As mentioned earlier, a substantial new component of this work deals with secure floating-point arithmetic in the malicious model. We are not aware of any work that provides techniques for floating-point operations in the security model with fully malicious participants. We show that such solutions can be built in the multi-party case using existing techniques, while in the two-party setting they require new tools which we put forward in this work.

3 Hidden Markov models and Gaussian mixture models

A HMM is a statistical model that follows the Markov property (where the transition at each step depends only on the previous transition) with hidden states, but visible outcomes. The inputs are a sequence of observations, and for each sequence of observations (or outcomes), the computation consists of determining a path of state transitions which is the likeliest among all paths that could produce the given observations. More formally, an HMM consists of:

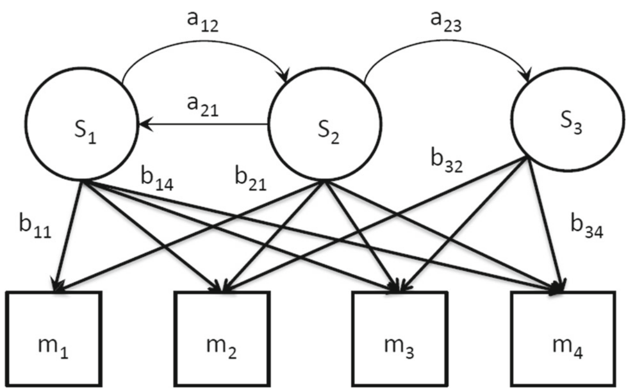

Fig. 1 An example of a hidden Markov model with s_i's representing states and m_i's representing outcomes

- N states $S_1, \ldots, S_N$;
- M possible outcomes $m_1, \ldots, m_M$;
- a vector $\pi = \langle \pi_1, \ldots, \pi_N \rangle$ that contains the initial state probability distribution, i.e., $\pi_i = \Pr[q_1 = S_i]$, where q is a random variable over the set of states indexed by the transition number;
- a matrix A of size $N \times N$ that contains state transition probabilities, i.e., a cell a_{ij} of A at row i and column j contains the probability of the transition from state S_i to state $S_j a_{ij} = \Pr[q_{k+1} = S_j | q_k = S_i]$;
- a matrix B of size $N \times M$ that contains output probabilities, i.e., a cell b_{ij} of B at row i and column j contains the probability of state S_i outputting outcome $m_j b_{ij} = \Pr[q_k = S_i | X_k = m_j]$.

In the above, observations $X_1, \ldots, X_T$ form HMM's input, to which we collectively refer as X. The above parameters define an HMM. In our running application of speaker recognition, an HMM is a model that represents how a particular person speaks and an input corresponds to the captured voice sample of a client. Figure 1 shows an example of an HMM.

In most cases, matrix B is computed based on observations. Usually this is done by evaluation of the observed value on probability distributions of states' outcomes. Then based on the observations, certain elements of B are used to form a $N \times T$ matrix. For clarity of exposition, we refer to the elements of B chosen based on, or computed from, the current observations as $N \times T$ matrix β.

One very common distribution model used to compute β is a GMM. GMMs are mixtures of Gaussian distributions that represent the overall distribution of observations. Namely, an observation is evaluated on a number of Gaussian distributions with different parameters and the evaluations are combined together to produce the final probability of a random variable acquiring that particular observation. In the case of HMMs, we use a GMM to compute the output probability of state S_j producing an observation at time k as follows:

$$\beta_{jk} = \sum_{i=1}^{\alpha} w_i \mathrm{e}^{-\frac{1}{2}(X_k - \mu_i)^T \Sigma_i^{-1} (X_k - \mu_i)} \tag{1}$$

In the above, X_k is a vector of size f that represents the random variable corresponding to the observation at time k. In voice applications, X_k usually contains the Mel-frequency cepstra coefficients (MFCCs). The parameter α is the total number of mixture components (here, Gaussian distributions). The ith component has a mean vector μ_i of size f and a covariance matrix Σ_i of size $f \times f$. The components are added together, each weighted by a mixture weight w_i, to produce the probability distribution of state S_j when the observed random variable is X_k. We use notation μ, Σ, and w to refer to the sequence of μ_i, Σ_i, and w_i, respectively, for $i = 1, \ldots, \alpha$.

There are three different types of problems and respective dynamic programming algorithms for HMM computation: the forward algorithm, the Viterbi algorithm, and the expectation maximization (EM) algorithm [36]. In the forward algorithm, the goal is to compute the probability of each state for each transition given a particular sequence of observations. Namely, in this algorithm, we compute $\Pr[q_k = S_i | X_1 \ldots X_T]$. In the Viterbi algorithm, after observing the outcomes, the goal is to construct the path of states which is the most likely among all possible paths that can produce the observations, as well as the probability of the most likely path. In other words, for any given sequence of observations, each path has a certain probability of producing that sequence of observations. The output of the Viterbi algorithm is the path with the highest probability and the value of the probability. The computation performed in the Viterbi algorithm uses the forward algorithm. In the EM algorithm, the goal is to learn the HMM. Namely, given a sequence of observations, this algorithm computes the parameters of the most likely HMM that could have produced this sequence of observations. All of these three algorithms use dynamic programming techniques and have complexity of $O(TN^2)$.

Because in this work we use speaker recognition to demonstrate secure techniques for HMM computation, we focus on the Viterbi algorithm used in speaker recognition. The techniques developed in this work, however, can also be used to construct secure solutions for the other two algorithm. In what follows, we provide a brief description of the Viterbi algorithm and refer the reader to online materials for the forward and EM algorithms. In the algorithm below, P^* is the probability of the most likely path for a given sequence of observations and $q^* = \langle q_1^*, \ldots, q_T^* \rangle$ denotes the most likely path itself. The computation uses dynamic programming to store intermediate probabilities in δ, after which the path of the maximum likelihood is computed and placed in q^*.

```
⟨P*, q*⟩ ← Viterbi(λ = ⟨N, T, π, A, β⟩)
1. InitializationStep: for $i = 1$ to $N$ do
    (a) $\delta_1(i) = \pi_i \beta_{i1}$
    (b) $\psi_1(i) = 0$
2. RecursionStep: for $k = 2$ to $T$ and $j = 1$ to $N$ do
    (a) $\delta_k(j) = \left(\max_{1 \le i \le N} [\delta_{k-1}(i) a_{ij}]\right) \beta_{jk}$
    (b) $\psi_k(j) = \arg\max_{1 \le i \le N} [\delta_{k-1}(i) a_{ij}]$
3. TerminationStep:
    (a) $P^* = \max_{1 \le i \le N} [\delta_T(i)]$
    (b) $q_T^* = \arg\max_{1 \le i \le N} \delta_T(i)$
    (c) for $k = T - 1$ to 1 do $q_k^* = \psi_{k+1}(q_{k+1}^*)$
4. Return $\langle P^*, q^* \rangle$
```

In speaker recognition, we apply the Viterbi algorithm to extracted voice features and an HMM that was created using a GMM and training voice features. The overall computation then consists of forming an HMM using GMM computation and executing the Viterbi algorithm, as given next.

```
⟨P*, q*⟩ ← HMM(N, T, π, A, α, w, μ, Σ, X)
1. For $j = 1$ to $N$ and $k = 1$ to $T$, compute $\beta_{jk}$ as in equation 1 using $\alpha$, $w_i$'s, $\mu_i$'s, $\Sigma_i$'s, and $X_k$.
2. Set $\lambda = \langle N, T, \pi, A, \beta \rangle$.
3. Execute $\langle P^*, q^* \rangle = \mathsf{Viterbi}(\lambda)$.
4. Return $\langle P^*, q^* \rangle$.
```

4 Framework

In this section, we introduce two categories of secure computation that we consider in this work (two- and multi-party), precisely define the computation to be carried out, and formalize two security models for secure computation.

4.1 Two-party computation

The first category of secure computation that we consider is secure *two-party* computation. Without loss of generality, we will refer to the participants as the client and the server. Using speaker recognition as the example application, the setting can be described as follows: The client possesses a voice sample, the server stores a model that represents how a registered user speaks, and user authentication is performed by conducting HMM computation on the client's and server's inputs. Therefore, for the purposes of this work, we assume that the client owns the observations to an HMM, i.e., $X_1, \ldots, X_T$, and the server holds the parameters of the HMM and GMM, i.e., N, vector π, matrix A, α, mixture weights w, vectors μ, and matrices Σ. Because even the parameters of HMM might reveal information about the possible input observations, to build a fully privacy-preserving solution in which the server does not learn information about user biometrics, the server should not have access to the HMM parameters in the clear. For that reason, we assume that the server holds the parameters π, A, B, w, μ, and Σ in an encrypted form and computation proceeds on encrypted data. While there are other underlying techniques for secure two-party computation (such as garbled circuit evaluation), we view storing HMM data encrypted at the server and evaluating the function on encrypted data as the best option, despite high computational overhead associated with this approach. If encryption is not used, the HMM values will need to be split into random shares, with one share of each value stored by the client and the other share stored by the server. This creates multiple issues, one of which is that the client's state is large and the shares of the HMM must be present on each device from which the client wants to authenticate. The second issue is that a malicious user will need to be forced to enter the original HMM data into each authentication session to avoid tampering with the authentication process, which is generally not known how to do.

To permit the computation to take place on encrypted data, we resort to an encryption scheme with special properties, namely semantically secure additively homomorphic public-key encryption scheme (defined below). Furthermore, to ensure that neither the server can decrypt the data it stores, nor the (untrusted) client can decrypt the data (or a function thereof) without the server's consent, we utilize a (2, 2)-threshold encryption scheme. Informally, it means that the decryption key is partitioned between the client and the server, and each decryption requires that both of them participate. This means that the client and the server can jointly carry out the HMM computation and make the result available to either or both of them. For concreteness of exposition, we will assume that the server learns the outcome.

A public-key encryption scheme is defined by three algorithms (Gen, Enc, Dec), where Gen is a key generation algorithm that on input of a security parameter 1^κ produces a public-private key pair (pk, sk); Enc is an encryption algorithm that on input of a public key pk and message m produces ciphertext c; and Dec is a decryption algorithm that on input of a private key sk and ciphertext c produces decrypted message m or special character $\perp$ that indicates failure. For conciseness, we use notation $\mathsf{Enc}_{pk}(m)$ and $\mathsf{Dec}_{sk}(c)$ in place of $\mathsf{Enc}(pk, m)$ and $\mathsf{Dec}(sk, c)$, respectively. An encryption scheme is said to be additively homomorphic if applying an operation to two ciphertexts results in the addition of the messages that they encrypt, i.e., $\mathsf{Enc}_{pk}(m_1) \cdot$

$\mathsf{Enc}_{pk}(m_2) = \mathsf{Enc}(m_1+m_2)$. This property also implies that $\mathsf{Enc}_{pk}(m)^k = \mathsf{Enc}_{pk}(k \cdot m)$ for a known k. In a public key (n, t)-threshold encryption scheme, the decryption key sk is partitioned among n parties, and $t \leq n$ of them are required to participate in order to decrypt a ciphertext while $t - 1$ or fewer parties cannot learn anything about the underlying plaintext. Lastly, a semantically secure encryption scheme guarantees that no information about the encrypted message can be learned from its ciphertext with more than a negligible (in κ) probability. Semantically secure additively homomorphic threshold public-key encryption schemes are known, one example of which is Paillier encryption [29].

We obtain that in the two-party setting, the client and the server share the decryption key to a semantically secure additively homomorphic (2, 2)-threshold public-key encryption scheme. The client has private input $X_1, \ldots, X_T$ and its share of the decryption key sk; the server has input $\mathsf{Enc}_{pk}(\pi_i)$ for $i \in [1, N]$, $\mathsf{Enc}_{pk}(a_{ij})$ for $i \in [1, N]$ and $j \in [1, N]$, $\mathsf{Enc}_{pk}(w_i)$ for $i \in [1, \alpha]$, encryption of each element of μ_i and Σ_i for $i \in [1, \alpha]$, and its share of sk. The computation consists of executing the Viterbi algorithm on their inputs, at the end of which the server learns P^* and q_i^* for $i = 1, \ldots, T$. The size of the problem, i.e., parameters N, T, α, and f, are assumed to be known to both parties.

4.2 Multi-party computation

The second category of secure computation that we consider is secure *multi-party* computation on HMMs. In this setting, either a number of parties hold inputs to a multi-observer HMM or one or more clients wish to outsource HMM computations to a collection of servers. More generally, we divide all participants into three groups: (i) the input parties who collectively possess the private inputs, (ii) the computational parties who carry out the computation, and (iii) the output parties who receive the result(s) of the computation. These groups can be arbitrarily overlapping, which gives great flexibility in the setup and covers all possible cases of joint multi-party computation (where the input owners carry out the computation themselves, select a subset of them, or seek help of external computational parties) and outsourcing scenarios (by either a single party or multiple input owners).

To conduct computation on protected values in this setting, we utilize an information-theoretically secure threshold linear secret sharing scheme (such as Shamir secret sharing scheme [38]). In a (n, t)-threshold secret sharing scheme, a secret value s is partitioned among n participants in such a way that the knowledge of t or fewer shares information-theoretically reveals no information about s, while $t + 1$ or more shares allow for efficient reconstruction of s.[1] Such schemes avoid the use of computationally expensive public-key encryption techniques and instead operate on small integers (in a field $\mathbb{F}_p$, normally with prime p) of sufficient size to represent all values. In a linear secret sharing scheme, any linear combination of secret shared values is performed by each participant locally (which in particular includes addition and multiplication by a known), while multiplication requires interaction of the parties. It is usually required that $t < n/2$ which implies $n > 2$.

We then obtain that in this setting the input parties share their private inputs among $n > 2$ computational parties, the computational parties execute the Viterbi algorithm on secret-shared values, and communicate shares of the result to the output parties, who reconstruct the result from their shares. As before, the size of the problem—namely, the parameters N, T, α, and f—is known to all parties.

4.3 Security model

Security of any multi-party protocol (with two or more participants) can be formally shown according to one of the two standard security definitions (see, e.g., [20]). The first, weaker security model assumes that the participants are semi-honest (also known as honest-but-curious or passive), defined as they follow the computation as prescribed, but might attempt to learn additional information about the data from the intermediate results. The second, stronger security model allows dishonest participants to arbitrarily deviate from the prescribed computation. We show our techniques secure in both of these models and next present formal security definitions for semi-honest and malicious security settings.

Definition 1 Let parties $P_1, \ldots, P_n$ engage in a protocol Π that computes a (possibly probabilistic) n-ary function $f : (\{0, 1\}^*)^n \rightarrow (\{0, 1\}^*)^n$, where P_i contributes input in_i and receives output out_i. Let $\mathrm{VIEW}_\Pi(P_i)$ denote the view of participant P_i during the execution of protocol Π. More precisely, P_i's view is formed by its input and internal random coin tosses r_i, as well as messages $m_1, \ldots, m_k$ passed between the parties during protocol execution: $\mathrm{VIEW}_\Pi(P_i) = (\mathsf{in}_i, r_i, m_1, \ldots, m_k)$. Let $I = \{P_{i_1}, P_{i_2}, \ldots, P_{i_\tau}\}$ denote a subset of the participants for $\tau < n$ and $\mathrm{VIEW}_\Pi(I)$ denote the combined view of participants in I during the execution of protocol Π (i.e., $\mathrm{VIEW}_\Pi = (\mathrm{VIEW}_\Pi(P_{i_1}, \ldots, \mathrm{VIEW}_\Pi(P_{i_\tau})))$ and $f_I(\mathsf{in}_1, \ldots, \mathsf{in}_n)$ denote the projection of $f(\mathsf{in}_1, \ldots, \mathsf{in}_n)$ on

[1] We note that the meaning of t is defined differently in the literature for (n, t)-threshold encryption schemes and (n, t)-threshold secret sharing schemes. That is, in the former case, t shares are sufficient for reconstructing the secret, while in the latter case this can be achieved only with $t+1$ shares. For compatibility with prior work, we choose to follow standard definitions.

the coordinates in I (i.e., $f_I(\mathsf{in}_1, \ldots, \mathsf{in}_n)$ consists of the i_1th, ..., i_τth elements that $f(\mathsf{in}_1, \ldots, \mathsf{in}_n)$ outputs). We say that protocol Π is τ-private in the presence of semi-honest adversaries if for each coalition I of size at most τ and all $\mathsf{in}_i \in \{0, 1\}^*$ there exists a probabilistic polynomial time simulator S_I such that $\{S_I(\mathsf{in}_I, f_I(\mathsf{in}_1, \ldots, \mathsf{in}_n)), f(\mathsf{in}_1, \ldots, \mathsf{in}_n)\} \equiv \{\text{VIEW}_\Pi(I), (\mathsf{out}_1, \ldots, \mathsf{out}_n)\}$, where $\mathsf{in}_I = (\mathsf{in}_1, \ldots, \mathsf{in}_\tau)$ and "$\equiv$" denotes computational or statistical indistinguishability.

In the two-party setting, we have that $n = 2$, $\tau = 1$. The participants' inputs in_1, in_2 and outputs out_1, out_2 are set as described above. In the multi-party setting, $n > 2$, $\tau < n/2$, and the computational parties are assumed to contribute no input and receive no output (to ensure that they can be disjoint from the input and output parties). Then the input parties secret-share their inputs among the computational parties prior the protocol execution takes place, and the output parties receive shares of the output and reconstruct the result after the protocol termination. This setting then implies that, in order to comply with the above security definition, the computation used in protocol Π must be data-oblivious, which is defined as requiring the sequence of operations and memory accesses used in Π to be independent of the input.

Security of a protocol in the malicious model is shown according to the ideal/real simulation paradigm. In the ideal execution of the protocol, there is a *trusted third party* (TTP) that evaluates the function on participants' inputs. The goal is to build a simulator S who can interact with the TTP and the malicious party and construct a protocol's view for the malicious party. A protocol is secure in the malicious model if the view of the malicious participants in the ideal world is computationally indistinguishable from their view in the real world where there is no TTP. Also the honest parties in both worlds receive the desired output. This gives us the following definition of security in the malicious model.

Definition 2 Let Π be a protocol that computes function $f : (\{0, 1\}^*)^n \rightarrow (\{0, 1\}^*)^n$, with party P_i contributing input in_i. Let $\mathcal{A}$ be an arbitrary algorithm with auxiliary input x and S be an adversary/simulator in the ideal model. Let $\text{REAL}_{\Pi,\mathcal{A}(x),I}(\mathsf{in}_1, \ldots, \mathsf{in}_n)$ denote the view of adversary $\mathcal{A}$ controlling parties in I together with the honest parties' outputs after real protocol Π execution. Similarly, let $\text{IDEAL}_{f,S(x),I}(\mathsf{in}_1, \ldots, \mathsf{in}_n)$ denote the view of S and outputs of honest parties after ideal execution of function f. We say that Π τ-securely computes f if for each coalition I of size at most τ, every probabilistic $\mathcal{A}$ in the real model, all $\mathsf{in}_i \in \{0, 1\}^*$ and $x \in \{0, 1\}^*$, there is probabilistic S in the ideal model that runs in time polynomial in $\mathcal{A}$'s runtime and $\{\text{IDEAL}_{f,S(x),I}(\mathsf{in}_1, \ldots, \mathsf{in}_n)\} \equiv \{\text{REAL}_{\Pi,\mathcal{A}(x),I}(\mathsf{in}_1, \ldots, \mathsf{in}_n)\}$.

4.4 Performance evaluation of secure protocols

Performance of secure computation techniques is of grand significance, as protecting secrecy of data throughout the computation often incurs substantial computational costs. For that reason, besides security, efficient performance of the developed techniques is one of our primary goals. In both of our settings, computation of a linear combination of protected values can be performed locally by each participant (i.e., on encrypted values in the two-party setting and on secret-shared values in the multi-party setting), while multiplication is interactive. Because often the overhead of interactive operations dominates the runtime of a secure multi-party computation algorithm, its performance is measured in the number of interactive operations (such as multiplications, as well as other instances which, for example, include opening a secret-shared value in the multi-party setting or jointly decrypting a ciphertext in the two-party setting). Furthermore, the round complexity, i.e., the number of sequential interactions, can have a substantial impact on the overall execution time and serves as the second major performance metric. Lastly, in the two-party setting, public-key operations (and modulo exponentiations in particular) impose a significant computational overhead and are used as an additional performance metric.

In this work, we use notation $[x]$ to denote that the value of x is protected, either through encryption or secret sharing.

5 Building blocks

Before presenting our solution, we give a brief description of the building blocks from the literature used in our solution.

First note that having secure implementations of addition and multiplication operations alone can be used to securely evaluate any functionality on protected values represented as an arithmetic circuit. Prior literature, however, concentrated on developing secure protocols for commonly used operations which are more efficient than general techniques. In particular, the literature contains a large number of publications for secure computation on integers such as comparisons, bit decomposition, and other operations. From all of the available techniques, we have chosen the building blocks that yield the best performance for our construction, which are listed in [1].

Note that we use the following complexity for elementary arithmetic operations: Addition and subtraction of two protected values in either two- or multi-party setting involves no communication. Multiplication in the multi-party setting requires each computational party to send values to all other parties and wait for values from them, which is performed simultaneously. This is treated as an elementary interactive operation in a single round. In the two-party setting, multiplication of two encrypted values $\mathsf{Enc}_{pk}(a)$ and $\mathsf{Enc}_{pk}(b)$ is computed interactively, during which one party chooses a random value r, forms $\mathsf{Enc}_{pk}(a-r)$ using homomorphic properties of the encryption scheme, and helps the second party to decrypt $a-r$. After that the parties locally compute $\mathsf{Enc}_{pk}(br)$ and $\mathsf{Enc}_{pk}(b(a-r))$, respectively, and exchange the ciphertexts to obtain $\mathsf{Enc}_{pk}(ba)$.

5.1 Floating-point building blocks

For floating-point operations, we adopt the same floating-point representation as in [2]. Namely, a real number x is represented as 4-tuple $\langle v, p, s, z\rangle$, where v is an ℓ-bit normalized significand (i.e., the most significant bit of v is 1), p is a k-bit (signed) exponent, z is a bit that indicates whether the value is zero, and s is a bit set only when the value is negative. We obtain that $x = (1-2s)(1-z)v \cdot 2^p$. As in [2], when $x = 0$, we maintain that $z = 1$, $v = 0$, and $p = 0$.

The work [2] provides a number of secure floating-point protocols, some of which we use in our solution as floating-point building blocks. While the techniques of [2] also provide the capability to detect and report errors (e.g., in case of division by 0, overflow or underflow, etc.), for simplicity of presentation, we omit error handling in this work. The building blocks from [2] that we use here are:

- $\langle [v], [p], [z], [s]\rangle \leftarrow \mathsf{FLMul}(\langle [v_1], [p_1], [z_1], [s_1]\rangle, \langle [v_2], [p_2], [z_2], [s_2]\rangle)$ performs floating-point multiplication of its two real valued arguments.
- $\langle [v], [p], [z], [s]\rangle \leftarrow \mathsf{FLDiv}(\langle [v_1], [p_1], [z_1], [s_1]\rangle, \langle [v_2], [p_2], [z_2], [s_2]\rangle)$ allows the parties to perform floating-point division using $\langle [v_1], [p_1], [z_1], [s_1]\rangle$ as the dividend and $\langle [v_2], [p_2], [z_2], [s_2]\rangle$ as the divisor.
- $\langle [v], [p], [z], [s]\rangle \leftarrow \mathsf{FLAdd}(\langle [v_1], [p_1], [z_1], [s_1]\rangle, \langle [v_2], [p_2], [z_2], [s_2]\rangle)$ performs the computation of addition (or subtraction) of two floating-point arguments.
- $[b] \leftarrow \mathsf{FLLT}(\langle [v_1], [p_1], [z_1], [s_1]\rangle, \langle [v_2], [p_2], [z_2], [s_2]\rangle)$ produces a bit, which is set to 1 iff the first floating-point argument is less than the second argument.
- $\langle [v], [p], [z], [s]\rangle \leftarrow \mathsf{FLExp2}(\langle [v_1], [p_1], [z_1], [s_1]\rangle)$ computes the floating-point representation of exponentiation $[2^x]$, where $[x] = (1-2[s_1])(1-[z_1])[v_1]2^{[p_1]}$.

These protocols were given in [2] only for the multi-party setting, but we also evaluate their performance in the two-party setting using the most efficient currently available integer building blocks (as specified in [1]). The complexities of the resulting floating-point protocols in both settings can be found in [1].

5.2 Zero-knowledge proofs of knowledge for Paillier encryption scheme

As previously mentioned, our solution in the two-party setting relies on an additively homomorphic threshold encryption scheme. In the malicious security setting, our construction relies on zero-knowledge proofs of knowledge (ZKPKs). Instantiations of such proofs of knowledge are typically specific to the encryption scheme used, which requires us to choose an appropriate homomorphic encryption scheme.

In this work, we thus utilize Paillier cryptosystem [29], which is is a popular additively homomorphic encryption scheme, threshold variants of which are available [5,15]. To be able to specify our solutions, we first need to describe the details of the encryption scheme. At key generation time, Gen, choose two prime numbers p and q, the size of which is based on the security parameter κ, set $N = pq$, set integer $s \geq 1$, and choose an appropriate generator $g \in \mathbb{Z}^*_{N^{s+1}}$. The public key consists of g and N, and the plaintext space is $\mathbb{Z}_{N^s}$. The private decryption key sk can be distributed (as in the threshold variant) and is computed from the knowledge of factors p and q. Encryption Enc of message $x \in \mathbb{Z}_{N^s}$ is performed by choosing $r \in \mathbb{Z}^*_{N^{s+1}}$ at random and setting the ciphertext to $g^x r^{N^s} \bmod N^{s+1}$. For simplicity, we use notation $\mathsf{Enc}(\cdot)$ instead of $\mathsf{Enc}_{pk}(\cdot)$ and in what follows we also use $\mathsf{Enc}(x, r)$ to denote Paillier encryption of message x when the random value used during encryption was r. Threshold decryption is performed by each party applying their share of the decryption key sk to a ciphertext and then combining partially decrypted ciphertext to recover the plaintext. To simplify presentation, we also let $s = 1$ in the rest of the paper.

Examples of existing ZKPKs for Paillier encryption include a proof of knowledge of plaintext [5,15], a proof that two plaintexts are equal [5], a proof that a ciphertext encrypts one value from a given set [5,15], a proof that a ciphertext encrypts a product of two other given encrypted values [13], and a range proof for the exponent a of plaintext b^a [24]. We utilize three particular ZKPKs: a proof that a ciphertext encrypts one of the two given values, a proof of plaintext knowledge, and a proof of plaintext multiplication. Below we specify these ZKPKs more formally using the popular notation of [8], $\mathsf{ZKPK}\{(S, P)\colon R\}$, which states that the prover possesses set S as her secret values, the values in set P are known to both parties, and the prover proves statement R.

- $\mathsf{PK12}((a, \rho), (a', p_1, p_2)) = \mathsf{ZKPK}\{(a, \rho), (a', p_1, p_2) : (a' = \mathsf{Enc}(a, \rho)) \wedge ((a = p_1) \vee (a = p_2))\}$. Here, the prover wishes to prove to the verifier that $a' = \mathsf{Enc}(a, \rho)$ is an encryption of one of the two known plaintexts p_1 and p_2.

- $\mathsf{PKP}((a, \rho), (a')) = \mathsf{ZKPK}\{(a, \rho), (a') : (a' = \mathsf{Enc}(a, \rho))\}$. The prover wishes to prove to the verifier that he knows the value a that the ciphertext a' encrypts (and thus that a' is a valid ciphertext).
- $\mathsf{PKPM}((b, \rho_b), (a', b', c')) = \mathsf{ZKPK}\{(b, \rho_b), (a', b', c'): (b' = \mathsf{Enc}(b, \rho_b)) \wedge (a' = \mathsf{Enc}(a)) \wedge (c' = \mathsf{Enc}(c)) \wedge (c = ab)\}$. The prover wishes to prove to the verifier that c' encrypts the product of the corresponding plaintexts of a' and b', where the prover knows the plaintext of b' (i.e., this is multiplication of an encrypted value by a known plaintext value).

For additional information (such as the appropriate choice of parameters), we refer the reader to [13,15].

6 Secure Viterbi and GMM computation in the semi-honest model

Now we are ready to put everything together and describe our privacy-preserving solution for HMM and GMM computation based on Viterbi algorithm using floating-point numbers.

To execute the HMM algorithm given in Sect. 3, we first need to perform GMM computation to derive the output probabilities b_{jk} using Eq. 1. It was suggested in [40] that the ith components of a GMM, $g_i(x) = -\frac{1}{2}(x - \mu_i)^T \Sigma_i^{-1}(x-\mu_i)$, is represented as $x^T Y_i x + y_i^T x + y_{i0}$, where $Y_i = -\frac{1}{2}\Sigma_i^{-1}$, $y_i = \Sigma_i^{-1}\mu_i$, and $y_{i0} = -\frac{1}{2}\mu_i^T \Sigma_i^{-1}\mu_i$. The suggested representation increases the number of additions and multiplications than the original formula and therefore would result in a slower performance. In particular, because FLAdd is a relatively expensive protocol, we would like to minimize the use of this function. Thus, we suggest that the parties first subtract μ_i from the observed vector x and engage in matrix multiplication to compute g_i. Note that the parties can run the above computation in parallel for all values of i, j, and k in Eq. 1. After its completion, the parties proceed with performing the secure version of the Viterbi algorithm.

The Viterbi algorithm requires (floating-point) multiplication, max, and arg max. The multiplication is implemented using FLMul, while the max and arg max operations can be implemented using FLLT in a tree like fashion (i.e., we first compare every two adjacent elements, then every two maximum elements from the first round of comparisons, etc.). To perform arg max of two floating-point numbers at indices i and j, let $[b]$ be the outcome of the FLLT operation applied to the numbers at these indices. Then we set arg max to equal to $[b]j + (1-[b])i$. Therefore, arg max of a number of floating-point values can be computed in a tree like fashion using the above method for each comparison.

After completing the recursion step of the Viterbi algorithm, we need to retrieve the sequence of states in the HMM that resulted in the most likely path (steps 3b and 3c in the Viterbi algorithm). If the sequence of these states can be made publicly available, then the parties open q_i^* for $1 \le i \le T$ to learn the sequence. However, in a more likely event that this sequence needs to stay protected from one or more parties, we make use of the protocol Pow2 from [2]. To compute $[q_t^*] = \psi_{t+1}([q_{t+1}^*])$ given $[q_{t+1}^*]$, we execute $\mathsf{BitDec}(\mathsf{Pow2}([q_{t+1}^*] - 1, N), N, N)$ that will produce N bits, all of which are 0 except the bit at position q_{t+1}^*. We then multiply each bit of the result by the respective element of the vector ψ_{t+1} and add the resulting values to obtain $[q_t^*]$.

In the two-party setting, however, one of the parties (e.g., the server) will learn the sequence as its output and a more efficient approach is possible. The idea is to take advantage of the fact that all encrypted values of the matrix ψ are held by both parties. In this case, the party receiving the output retrieves $\mathsf{Enc}_{pk}(\psi_{t+1}(q_{t+1}^*))$ using its knowledge of q_{t+1}^* which became available to that party in the previous step, randomizes the ciphertext by multiplying it with a fresh encryption of 0, and sends the result to the other party. Note that this randomization does not change the value of the underlying plaintext, but makes it such that the party receiving it cannot link the randomized ciphertext to one of the encryptions it possesses. This party then applies decryption to the received ciphertext and sends it back to the receiving party, who finishes the decryption, learns q_t^*, and continues to the next iteration of the computation.

Our secure solution has the same asymptotic complexity as the original algorithm, expressed as a function of parameters N, T, α, and f. In particular, the GMM computation involves $O(\alpha f^2 NT)$ floating-point operations and the Viterbi algorithm itself uses $O(N^2 T)$ floating-point operations (the recursion step dominates the complexity of the overall algorithm). In the two-party setting, our solution that employs homomorphic encryption additionally has dependency on the computational security parameter κ and the bitlength representation of the underlying floating-point values, while in the multi-party setting based on secret sharing, the complexity has dependency only on the bitlength representation of the floating-point values. More precisely, using the complexities of our building blocks as specified in [1], we obtain that the GMM computation in the two-party setting involves $O(\alpha f^2 NT(\ell \log \ell + k))$ modulo exponentiations (which depend on the security parameter κ) and communicates $O(\alpha f^2 NT(\ell \log \ell + \log k))$ ciphertexts and/or decryption shares (which likewise depend on the security parameter κ). Here, ℓ is the significant bitlength and k is the exponent bitlength of the floating-point numbers. In the multi-party setting, the complexity becomes $O(\alpha NT(f^2 \ell \log \ell + f^2 k + \ell^2))$ interactive operations (which depend on the number of participating parties n). The Viterbi algorithm involves $O(N^2 Tk)$ modulo exponentiations and communicates $O(N^2 T \log \ell)$ ciphertexts/decryption shares in the two-party setting, and it uses $O(N^2T(\ell + k))$ interactive operations in the multi-party setting.

The security of our solution can be stated as follows:

Theorem 1 *Assuming security of prior building blocks, the Viterbi solution is secure both in the* two-party *and* multi-party *settings in the semi-honest security model.*

Proof (Proof sketch) The security of our solution according to Definition 1 is based on the fact that we only combine previously known secure building blocks. Such building blocks take protected inputs and produce protected outputs, which means that their composition does not reveal information about private values. In particular, we can apply Canetti's composition theorem [9], which states that a composition of secure sub-protocols leads to security of the overall solution. More precisely, in both two-party and multi-party settings, we can build a simulator S of the overall solution according to Definition 1, which without access to private data produces a view that cannot be distinguished from the participants' views in the real protocol execution. Our simulator calls the corresponding simulators for the underlying building blocks. Then because each underlying simulator produces a view that is either computationally or statistically indistinguishable (depending on the setting) from the view of a particular party and no information is revealed while combining the building blocks, the simulation of each participant's view in the overall protocol also cannot be distinguished from a real protocol execution. We thus obtain security of our solution in the semi-honest model.

7 Experimental results

In this section, we report on the results of implementation of our HMM solution. Note that because of numerous applications of HMMs, their use in certain contexts might differ, and we therefore chose to implement only the core HMM computation without the GMM component. The output probabilities matrix β can be computed based on observations via different means (one of which is GMM) and for the purposes of our implementation we choose discrete assignment of β_{jk}'s based on the sequence of observations $X_1, \ldots, X_T$. In particular, for each $j = 1, \ldots, N$ and $k = 1, \ldots, T$, we set $\beta_{jk} = b_{j,i}$ using matrix B, where i corresponds to the index that the value X_k takes (out of M possible outcomes). In the two-party setting, this means that the client who possesses the observations $X_1, \ldots, X_T$ receives encrypted matrix B from the server and sets $\mathsf{Enc}(\beta_{jk}) = \mathsf{Enc}(b_{j,i}) \cdot \mathsf{Enc}(0)$ according to X_k, where $\mathsf{Enc}(0)$ is used for randomization purposes. In the multi-party case, the parties jointly hold X and B in protected form and obliviously set β_{jk} based on X_k (i.e., without knowing what cell of B was used to set each β_{jk}).

Our implementations were built in C/C++. All machines used in the experiments had identical hardware with four-core 3.2 GHz Intel i5-3470 processors with Red Hat Linux 2.6.32 and were connected via a 1 Gb LAN. Only one core was used during the experiments (i.e., multi-threading was not used).

In what follows, we first describe our two-party experiments followed by the experiments in the multi-party setting. In our implementations, we represent floating-point numbers using 32-bit significands and (signed) 9-bit exponents (plus sign and zero bits as described earlier in this work).

In the two-party setting, we utilized $(2, 2)$-threshold Paillier encryption, which was implemented using Miracl library [12] for large number arithmetic. The experiments we report were conducted using a 1536-bit modulus for Paillier encryption. Because performance of our building blocks is not available in prior literature, we provide runtimes of integer and floating-point operations used in our implementation in Fig. 2 and overall HMM computation in Fig. 3. The parameters N and T for HMM experiments were chosen as suggested in the speaker recognition literature [4,25]. That is, a typical value for N is 3 and T is around 100 (using 32 ms/frame in [25]). We separately vary N and T to illustrate how performance depends on these values. All experiments were run five times and the mean value is given.

Because techniques based on homomorphic encryption are computationally intensive, we separate all work into offline and online, where the offline work consists of computation that can be performed before the inputs become available (e.g., generating random values and encrypting them). We thus measure offline work for client and server and the overall online runtime. In our experiments with identical machines, the server performs somewhat more work and thus takes longer, but in practice the server is expected to be a more powerful machine with client's performance being the bottleneck. For integer and floating-point operations, we report the time per operation when a number of operations are executed in a single batch. Batch execution reduces communication overhead when simultaneous execution of a number of operations is possible (as is the case for HMMs). This results in reducing the total online time.

Figure 2 presents performance of integer and floating-point operations in the two-party setting as described above. The two-party performance of HMM computation in Fig. 3 is consistent with the complexity of the algorithm, which has linear dependency on T and quadratic dependency on N (i.e., the slope for $N = 3$ is different from the slope for $N = 6$). In both figures, most of offline work is done by the server, which benefits overall execution time.

In the multi-party setting, we utilized three computational parties that operate on shares of the data formed using a $(3, 1)$-threshold linear secret sharing scheme. The implementation was built using the PICCO compiler [41], in which

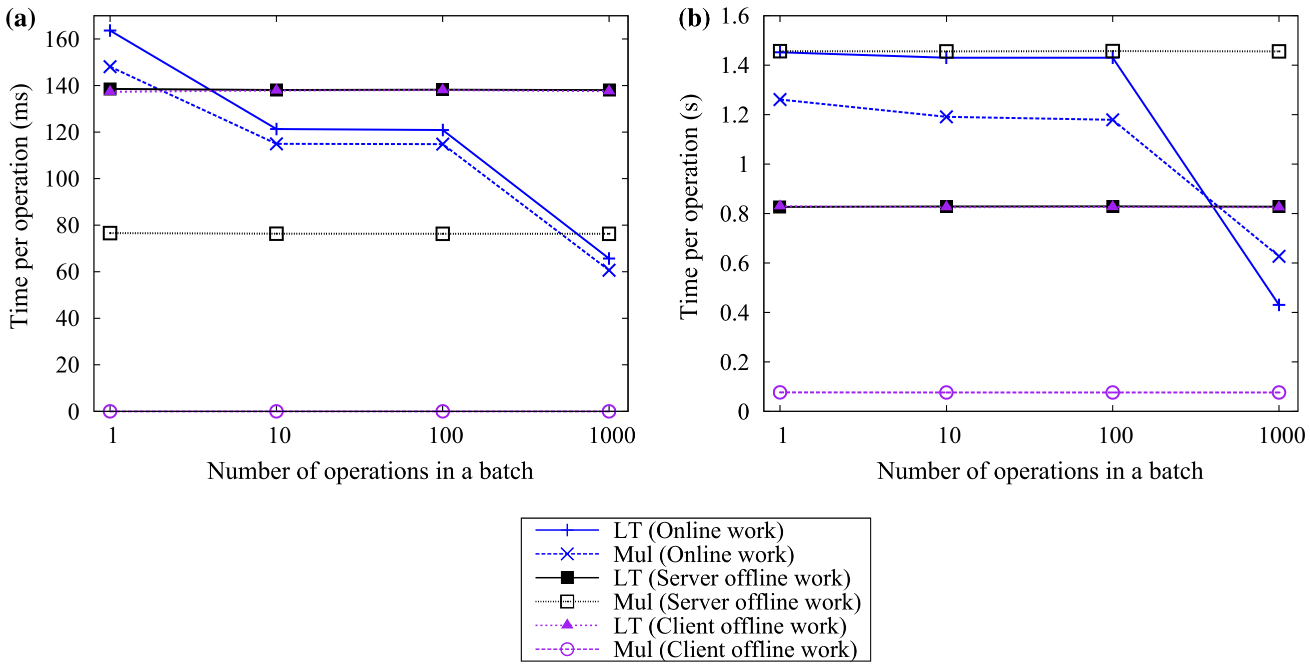

Fig. 2 Performance of HMM's building blocks in the two-party setting: **a** integer operations, **b** floating-point operations

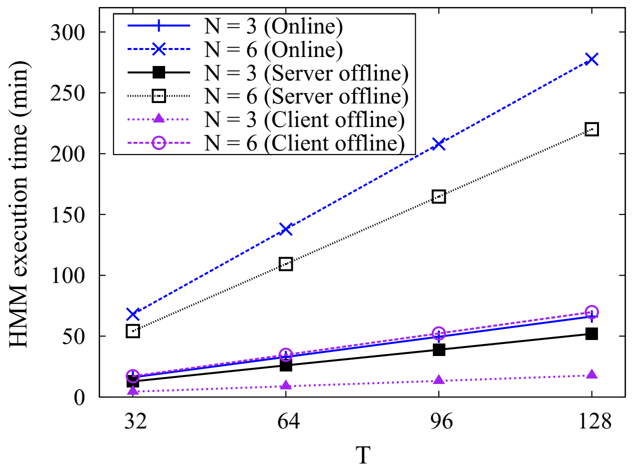

Fig. 3 Performance of HMM computation for varying N and T in the two-party setting

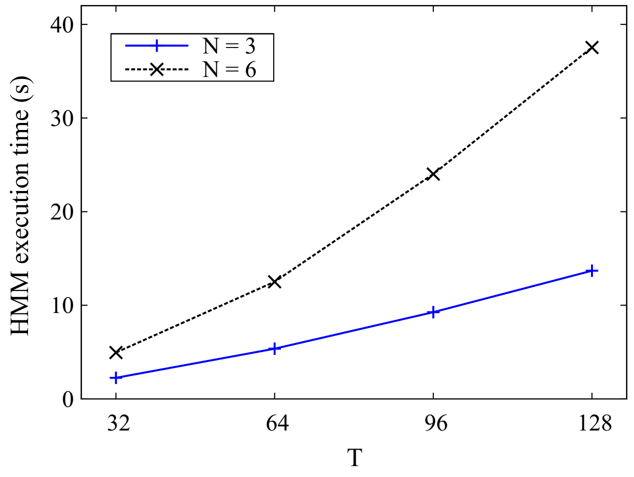

Fig. 4 Performance of HMM computation for varying N and T in the multi-party setting

an HMM program written in an extension of C was compiled into its secure distributed implementation. PICCO is a source-to-source compiler that produces a C program and utilizes the GMP [21] library for the underlying arithmetic and OpenSSL [28] implementation of AES for protecting communication. All arithmetic was performed in field $\mathbb{F}_p$ for a 114-bit prime p (the modulus size was determined as described in [41]). The results of HMM experiments are given in Fig. 4. While separation between online and offline work is also possible in this setting (e.g., the parties generate a large number of random values throughout the protocol execution), we do not distinguish between these types of work and list the overall performance in the online category. This is in part because we use an existing tool for our experiments and in part because the multi-party performance is orders of magnitude faster than encryption-based two-party computation and is already practical.

In conclusion, we note that our two-party setting was constrained in terms of the tools that could be employed for secure computation. That is, in order to provably protect HMMs from the server, we have to resort to strong protection mechanisms such as homomorphic encryption, the threshold version of which was required to ensure that no individual party could independently decrypt ciphertexts and learn unauthorized information. In general, alternative techniques of better performance (such as garbled circuits) are possible and even additively homomorphic encryption can be substantially faster (see, e.g., [7]) when threshold decryption is not required. One suggestion for improving performance of the two-party setting for this application is to involve neutral third parties, which would allow for the use of multi-party techniques.

8 Secure Viterbi and GMM computation in the malicious model

We next show how strengthen our solution to maintain security in the presence of malicious participants who can arbitrarily deviate from the prescribed behavior. Our solution for the multi-party setting, covered in Sect. 8.1, majorly follows from prior work and is described rather concisely. Most of the section is thus dedicated to the two-party setting, where in Sect. 8.2 we describe the necessary components for building a solution secure against malicious adversaries and in Sects. 8.3–8.8 present protocols for two-party multiplication, comparison, truncation, inversion, prefix multiplication, and bit decomposition, respectively, together with their malicious model security analysis.

8.1 Multi-party setting

The security of our solution in the multi-party setting can also be extended to the malicious security model. In that case, to show security in the presence of malicious adversaries, we need to ensure that (i) all participants prove that each step of their computation was performed correctly and that (ii) if some dishonest participants quit, others will be able to reconstruct their shares and proceed with the rest of the computation. The above is normally achieved using a verifiable secret sharing scheme (VSS), and a large number of results have been developed over the years (e.g., [3,14,19] and many others). In particular, because any linear combination of shares is computed locally, each participant is required to prove that it performed each multiplication correctly on its shares. Such results normally work for $t < \frac{n}{3}$ in the information theoretic or computational setting with different communication overhead and under a variety of assumptions about the communication channels. Additional proofs associated with this setting include proofs that shares of a private value were distributed correctly among the participants (when the dealer is dishonest) and proofs of proper reconstruction of a value from its shares (when not already implied by other techniques). In addition, if at any point of the computation the participants are required to input values in a specific form, they would have to prove that the values they supplied are well-formed. Such proofs are needed by the implementations of some of our building blocks (e.g., RandInt that prescribed the parties to choose a random value of a specific bitlength). Thus, security of our protocols in the malicious model in the multi-party setting can be achieved by using a VSS scheme, where a range proof such as [34] (secure in the computational setting) or an alternative mechanism suggested in [6] (secure in the information-theoretic sense) will be additionally needed for the building blocks. These VSS techniques would also work with malicious input parties, who would need to prove that they generate legitimate shares of their data.

8.2 Two-party setting

To show security of our solution in the presence of malicious adversaries in the two-party setting, we likewise need to enforce correct execution of all operations. However, unlike the multi-party setting, this time security no longer follows from prior work and requires new tools.

To ensure that both participants follow all steps of the computation, we employ ZKPKs. Because such proofs are usually tied to the internal workings of the underlying homomorphic encryption scheme, we develop our solution based on the Paillier encryption scheme.

Our approach consists of designing protocols secure in the presence of malicious adversaries for a number of building blocks used to build floating-point operations. Then after applying Canetti's composition theorem [9], we can guarantee security of larger building blocks and the overall solution. To determine which building blocks need to be implemented in the stronger security model with fully malicious participants, we analyze each floating-point operation used in this work.

- FLMul is implemented using protocols Trunc, LT, OR, XOR, and Mul as the building blocks. Trunc in turn depends on TruncPR and LT protocols.[2] OR and XOR protocols are built directly from Mul. This means that we need to realize malicious versions of multiplication Mul, truncation TruncPR, and comparison LT.
- Besides some of the building blocks listed above, FLDiv additionally uses SDiv, which in turn is built using Mul and TruncPR protocols. Thus, no additional protocols are needed.
- FLAdd calls new building blocks EQ, Pow2, BitDec, PreOR, and Inv. Our implementation of EQ is built on

[2] Throughout this description we don't describe the functionality of each building block. Such description will be given only for the building blocks that we need to implement in the malicious model.

LT and Mul. Pow2 calls BitDec and PreMul. PreOr calls PreMul and Mod2, which is equivalent to Trunc. Thus, we need to implement three new building blocks bit decomposition BitDec, inverse computation Inv, and prefix multiplication PreMul in the malicious model.
- FLLT does not call any new building blocks.
- Similarly, FLExp calls only integer building blocks discussed above and FLMul that can be assembled from integer building blocks.

To summarize, we need to provide implementations of six functionalities secure in the malicious model, which are described next:

- $\mathsf{Enc}(xy) \leftarrow \mathsf{MalMul}(\mathsf{Enc}(x), \mathsf{Enc}(y))$ is a fundamental building block, which performs multiplication of its two encrypted input values x and y.
- $\mathsf{Enc}(b) \leftarrow \mathsf{MalLT}(\mathsf{Enc}(x), \mathsf{Enc}(y), \ell)$ performs comparison of two encrypted values x and y of size ℓ and outputs encrypted bit b, where $b = 1$ iff $x < y$.
- $\mathsf{Enc}(y) \leftarrow \mathsf{MalTruncPR}(\mathsf{Enc}(x), \ell, k)$ truncates k bits of encrypted x, which has bitlength ℓ. The output is probabilistic, where the least significant bit of the result y may differ from that of $\lfloor x/2^k \rfloor$.[3]
- $\mathsf{Enc}(y) \leftarrow \mathsf{MalInv}(\mathsf{Enc}(x))$ produces (encrypted) inversion $y = x^{-1}$ of encrypted x.
- $\mathsf{Enc}(y_1), \ldots, \mathsf{Enc}(y_k) \leftarrow \mathsf{MalPreMul}(\mathsf{Enc}(x_1), \ldots, \mathsf{Enc}(x_k))$ performs prefix multiplication of k nonzero encrypted values $x_1, \ldots, x_k$, where the result is computed as $y_i = \prod_{j=1}^{i} x_j$ for each $i \in [1, k]$.
- $\mathsf{Enc}(x_{k-1}), \ldots, \mathsf{Enc}(x_0) \leftarrow \mathsf{MalBitDec}(\mathsf{Enc}(a), \ell, k)$ extracts k least significant bits of (encrypted) x, where ℓ is the bitlength of x.

Before we proceed with the description of the individual protocols, we note that the optimization to the termination step in the two-party setting described in Sect. 6 (for the semi-honest model) does not easily generalize to the malicious setting. We therefore assume that the computation proceeds according to the general solution in Sect. 6, which is constructed using building blocks which we already treat in this section (such as bit decomposition and prefix multiplication). Thus, security of the termination step will follow from the composition of other underlying secure sub-protocols.

In the rest of this section, we treat one protocol at a time and report performance of each new protocol in the malicious model (together with supporting ZKPKs) in Table 1.

[3] We note that such probabilistic version is sufficient in some cases, while in others the function can be changed to always produce correct truncation with the use of extra comparison.

8.3 Secure two-party multiplication in the malicious model

Now we describe a two-party multiplication protocol secure in the presence of a malicious participant. In this protocol, both parties hold $\mathsf{Enc}(x)$, $\mathsf{Enc}(y)$ without any knowledge of x or y and receive $\mathsf{Enc}(xy)$ as their output. The protocol is very similar to the one given in [16] for the multi-party setting. The intuition is that P_1 and P_2 blind encryption of x with their respective random values r_1 and r_2 and decrypt $c = x + r_1 + r_2$. This allows them to compute $\mathsf{Enc}(y)^c$, from which they subtract encrypted yr_1 and yr_2 to recover the result. Doing so securely will require that P_1 and P_2 prove correctness of $\mathsf{Enc}(yr_1)$ and $\mathsf{Enc}(yr_2)$, respectively, using PKPM.

$\mathsf{Enc}(xy) \leftarrow \mathsf{MalMul}(\mathsf{Enc}(x), \mathsf{Enc}(y))$

Public inputs include public key pk for (2, 2)-threshold Paillier encryption scheme and private inputs include shares of the corresponding secret key.

1. Each P_j chooses at random $r_j \in \mathbb{Z}_N$.
2. Each P_j computes $r'_j = \mathsf{Enc}(r_j, \rho_j)$, $z_j = \mathsf{Enc}(y)^{r_j} \cdot \mathsf{Enc}(0) = \mathsf{Enc}(y \cdot r_j)$, and executes $\mathsf{PKPM}((r_j, \rho_j), (\mathsf{Enc}(y), r'_j, z_j))$ to prove correctness of z_j to the other party.
3. Both parties locally compute $c' = \mathsf{Enc}(c) = \mathsf{Enc}(x) \cdot r'_1 \cdot r'_2$ and decrypt c' to recover c.
4. Each party computes $z = \mathsf{Enc}(y)^c = \mathsf{Enc}(yx + yr_1 + yr_2)$ and $\mathsf{Enc}(xy) = z \cdot z_1^{-1} \cdot z_2^{-1}$.

Following [13], we show security of this and other protocols in a hybrid model where decryption is replaced with ideal functionality. That is, instead of producing partial decryptions of a ciphertext and combining them to recover the corresponding plaintext, decryption is performed by submitting the ciphertext to a black-box which outputs the underlying plaintext. Then following the arguments of [13], we also obtain security in the real model when ideal decryption is instantiated with a real threshold decryption algorithm.

Theorem 2 *Assuming semantic security of the homomorphic encryption scheme and security of the building blocks, the above* MalMul *is secure in the malicious setting in the hybrid model with ideal decryption.*

Proof We prove security of MalMul based on Definition 2. Because this protocol is completely symmetric, without loss of generality we assume P_1 is malicious. In that case, in the ideal world a simulator S_1 locates between P_1 and the TTP and simulates P_1's view after receiving $\mathsf{Enc}(x)$ and $\mathsf{Enc}(y)$, as well as $\mathsf{Enc}(xy)$ from the TTP. During step 2, S_1 receives r'_1 and z_1, and acts as a verifier for PKPM (extracting r_1). Then, S_1 sets $r'_2 = \mathsf{Enc}(w - r_1 - x)$ for a randomly chosen $w \in \mathbb{Z}_N$, computes $z_2 = (\mathsf{Enc}(xy))^{-1}(\mathsf{Enc}(y))^{w-r_1}$, and uses the simulator of PKPM to interact with P_1 and prove validity of r'_2, z_2. During step 3, S_1 sends w as the decrypted value c to P_1. To show that P_1's output at the end of the simulation corresponds to correct $\mathsf{Enc}(xy)$, recall that the simulator sets $r_2 = w - r_1 - x$. Thus, $w = x + r_1 + r_2$ in step 3 is exactly what the simulator needs to decrypt. Therefore, P_1 correctly computes the output in step 4.

To show that the simulated view is indistinguishable from the view in the hybrid model (with ideal decryption), we note that the simulator produces encrypted values in step 2, which are indistinguishable from ciphertexts sent during the protocol execution because of semantic security of the encryption scheme. Similarly, the simulation of PKPM is indistinguishable from real execution due to its security properties. Lastly, the value w returned by the simulator in step 3 is distributed identically to the value c decrypted during protocol execution. We also note that both during the simulation and real execution the computation aborts only when a malicious party fails to correctly complete PKPM as the prover.

8.4 Secure two-party comparison in the malicious model

We next provide our comparison protocol secure in the presence of malicious adversaries. We start with the semi-honest protocol used in [23] and modify it to achieve security in a stronger model.

Before we proceed with the description of the solution itself, we observe that the protocol of [23] requires the parties to choose k-bit random numbers (for sufficiently large k) with some particular restrictions. Namely, each party chooses random r and r' with the constraint that $r > r'$ (which equivalently means that $0 < r - r' < 2^k$). Therefore, each party needs to prove to the other that her choice of random numbers is according to the protocol's requirement, without revealing the actual values of such random numbers. Since the maximum value of $r - r'$ is equal to the maximum value of r, this task is equivalent to a closed range proof. To achieve this, we develop a ZKPK that a ciphertext encrypts a value that belongs to a particular range of integers. Based on [24], $x \in [0, H]$ iff we can write $x = \Sigma_{j=0}^{M}(x_j H_j)$, where $H_j = \lfloor (H + 2^j)/2^{j+1} \rfloor$, $M = \lfloor \log_2 H \rfloor$ and $x_j \in \{0, 1\}$. Such decomposition of x into the x_j's can be computed using the following algorithm:

$\langle x_0, \ldots, x_M \rangle \leftarrow \mathsf{RangeDecompose}(x, H)$

```
1. $M = \lfloor \log_2(H) \rfloor$.
2. $a_0 = x$.
3. for $j = 0$ to $M$ do
4.     $H_j = \lfloor (H + 2^j)/2^{j+1} \rfloor$.
5.     if $a_j \geq H_j$ then $x_j = 1$ otherwise $x_j = 0$.
6.     $a_{j+1} = a_j - x_j \cdot H_j$.
7. return $\langle x_0, \cdots, x_M \rangle$.
```

We assume that this algorithm implicitly stores all computed H_j's which are available for the consequent computation.

For $x \in [L, H]$, we can use the transform of $y = x - L$, have $y \in [0, H - L]$, and obtain $y = \Sigma_{j=0}^{\lfloor \log_2(H-L) \rfloor}(y_j H_j)$ by calling RangeDecompose on y. Note that we now have $M = \lfloor \log_2(H - L) \rfloor$ and therefore $x = \Sigma_{j=-1}^{\lfloor \log_2(H-L) \rfloor}(x_j H_j)$, where $H_{-1} = L, x_{-1} = 1, H_j = \lfloor (H + 2^j - L)/2^{j+1} \rfloor$, and $x_j = y_j \in \{0, 1\}$ for j from 0 to M.

We provide our range ZKPK, called RangeProof, next. It is based on another proof from [24], which uses range decomposition of the input and ensures that each step of the algorithm was performed correctly.

$\mathsf{RangeProof}(x, L, H, e) = \mathsf{ZKPK}\{(x, \rho), (e, L, H) : (e = \mathsf{Enc}(x, \rho)) \wedge (L \leq x \leq H)\}$

1. The prover sets $\langle x_0, \ldots, x_M \rangle \leftarrow \mathsf{RangeDecompose}(x - L, H - L)$, $x_{-1} = 1$, and $H_{-1} = L$.
2. For $j = 0, \ldots, M$, the prover chooses $r_j \in \mathbb{Z}_N$ at random such that $\rho = \prod_{j=0}^{M} r_j$ and computes $c_j = \mathsf{Enc}(x_j \cdot H_j, r_j)$ and $c_{-1} = \mathsf{Enc}(L, 0)$.
3. For $j = 0, \ldots, M$, the prover executes $\mathsf{PK12}((x_j \cdot H_j, r_j), (c_j, 0, H_j))$ to prove that c_j is an encryption of either 0 or H_j.
4. The verifier checks whether $c_{-1} = \mathsf{Enc}(L, 0)$ and $e = \prod_{j=-1}^{M} c_j$.

A secure ZKPK should satisfy three properties [22]: (1) *completeness*, which informally means that if the statement is true, then an honest verifier (who follows the protocol) will be convinced by an honest prover; (2) *soundness*, which requires that if the statement is false, then no cheating prover (regardless of the cheating strategy) can convince an honest verifier, except with negligible probability, that the statement is true; and (3) *zero-knowledge*, which means that if the statement is true, then no cheating verifier can learn anything other than the fact that the statement is true. Below, we show that our range proof is a zero-knowledge proof of knowledge.

Theorem 3 *Assuming semantic security of the homomorphic encryption scheme, the ZKPK* RangeProof *is complete, sound, and zero-knowledge.*

Proof (Proof sketch) The completeness property follows easily. Namely, if the input is in fact in the range, an honest verifier will be convinced of this statement by the prover. This ZKPK is sound too, because if the input is not in the specified range and the cheating prover can successfully finish the protocol with a nonnegligible probability, then the underlying ZKPK PK12 is not sound as well. The latter, however, is not true due to [15]. The zero-knowledge property follows from the fact that if the verifier is a cheater, then the only place that she can employ a cheating strategy is in step 4. Note that in step 3, all values are encrypted and thus reveal no information to the verifier. Then because PK12 in step 4 is zero-knowledge [15], the overall solution is zero-knowledge as well. Alternatively, this property could be derived from the fact that sequential composition of ZKPKs is a zero-knowledge proof (from [22]).

Now we can turn our attention to the comparison protocol. We first detail the semi-honest protocol from [23] adopted to the two-party setting and then modify it to be secure in stronger security model with malicious participants.

$\mathsf{Enc}(b) \leftarrow \mathsf{LT}(\mathsf{Enc}(x), \mathsf{Enc}(y), \ell)$

Public inputs include public key pk for a (2, 2)-threshold Paillier encryption scheme and private inputs consist of shares of the corresponding secret key.

1. P_1 chooses $b_1 \in \{0, 1\}$ and $r_1, r_1' \in \{0, 1\}^{\ell+\kappa}$ at random, where $r_1 > r_1'$ and κ is a statistical security parameter.
2. P_2 chooses $b_2 \in \{0, 1\}$ and $r_2, r_2' \in \{0, 1\}^{\ell+\kappa}$ at random such that $r_2 > r_2'$.
3. P_1 computes $\mathsf{Enc}(c) = \mathsf{Enc}(x - y)$, $a_1 = \mathsf{Enc}(1 - b_1)$, and $a_2 = \mathsf{Enc}(b_1)$.
4. P_1 computes $a_3 = \mathsf{Enc}(c)^{(-1)^{b_1} r_1} \cdot \mathsf{Enc}((-1)^{1-b_1} r_1') = \mathsf{Enc}((-1)^{b_1} c r_1 + (-1)^{1-b_1} r_1')$ and sends a_1, a_2, a_3 to P_2.
5. P_2 computes $a_1' = a_{1+b_2} \cdot \mathsf{Enc}(0)$, $a_2' = a_{2-b_2} \cdot \mathsf{Enc}(0)$, where $\mathsf{Enc}(0)$ is used for randomization purposes.
6. P_2 computes $a_3' = a_3^{r_2 (-1)^{b_2}} \cdot \mathsf{Enc}((-1)^{1-b_2} r_2')$ and sends $\langle a_1', a_2', a_3' \rangle$ to P_1.
7. The parties decrypt a_3'. If the decrypted value is $< N^2/2$, output a_2'; otherwise, output a_1'.

In the above, a value in the range $[0, N^2/2)$ is treated as nonnegative and a value in the range $[N^2/2, N^2)$ as negative. The idea is that the output (encrypted) bit b is determined based on whether $x - y$ is positive or negative. P_1 either keeps the sign of $x - y$ or reverses it in a_3 based on its random bit b_1 obfuscates the difference $x - y$ using two random values r_1, r_1'. P_2 applies the same operations to the result using its random bit b_2 and random values r_2 and r_2'. The parties decrypt the resulting value and set the output based on the sign of the decrypted value and bits b_1 and b_2. In this protocol, κ is a statistical security parameter, the maximum value which can take to guarantee correctness is $(\lfloor \log(N^2) \rfloor - 2)/2$.

Our comparison protocol secure in the malicious model is given next. In what follows, we use the fact that $(-1)^b = 1 - 2b$ and $(-1)^{1-b} = 2b - 1$ when b is a bit. We also express a_1' as $\mathsf{Enc}(1 + 2b_1 b_2 - b_1 - b_2)$ and a_2' as $\mathsf{Enc}(b_1 + b_2 - 2b_1 b_2)$.

$\mathsf{Enc}(b) \leftarrow \mathsf{MalLT}(\mathsf{Enc}(x), \mathsf{Enc}(y), \ell)$

Public inputs include public key pk for a (2, 2)-threshold Paillier encryption scheme and private inputs consist of shares of the corresponding secret key.

1. Each P_j sets $e_1 = \mathsf{Enc}(1, 0)$, $e_{-1} = \mathsf{Enc}(-1, 0) = (e_1)^{-1}$, and $\mathsf{Enc}(c) = \mathsf{Enc}(x - y) = \mathsf{Enc}(x) \cdot \mathsf{Enc}(y)^{-1}$.
2. Each P_j chooses $b_j \in \{0, 1\}$, $r_j, r_j' \in \{0, 1\}^{\ell+\kappa}$ at random s.t. $r_j > r_j'$, and sends to $P_{3-j} z_{(j,1)} = \mathsf{Enc}(b_j, \rho_j)$, $z_{(j,2)} = \mathsf{Enc}(r_j, \rho_j')$ and $z_{(j,3)} = \mathsf{Enc}(r_j', \rho_j'')$. P_j executes $\mathsf{PK12}((b_j, \rho_j), (z_{(j,1)}, 0, 1))$, $\mathsf{RangeProof}(r_j, 0, H, z_{(j,2)})$, $\mathsf{RangeProof}(r_j', 0, H, z_{(j,3)})$, and $\mathsf{Range\!-\!Proof}(r_j - r_j', 1, H, z_{(j,2)} \cdot z_{(j,3)}^{-1})$ to prove that $z_{(j,1)}, z_{(j,2)}$, and $z_{(j,3)}$ are well-formed, where $H = 2^{\ell+\kappa} - 1$.
3. Each party locally computes $z_{(j,4)} = \mathsf{Enc}(1 - 2b_j) = e_1 \cdot (z_{(j,1)})^{-2}$ and $z_{(j,5)} = \mathsf{Enc}(2b_j - 1) = (z_{(j,1)})^2 \cdot e_{-1}$.
4. P_1 computes $z_6 = \mathsf{Enc}((1 - 2b_1)c)) = \mathsf{Enc}(c)^{1-2b_1} \cdot \mathsf{Enc}(0, \alpha_1)$, where α_1 is newly selected as randomness during encryption, $z_7 = \mathsf{Enc}(r_1(1 - 2b_1)c) = z_6^{r_1}$, and $z_8 = \mathsf{Enc}((2b_1 - 1)r_1') = z_{(1,5)}^{r_1'}$, and sends to $P_2 z_6, z_7$, and z_8. P_1 also executes $\mathsf{PKPM}((1 - 2b_1 \bmod N, \rho_1^{-2}), (\mathsf{Enc}(c), z_{(1,4)}, z_6))$, $\mathsf{PKPM}((r_1, \rho_1' + \alpha_1), (z_6, z_{(1,2)}, z_7))$, and $\mathsf{PKPM}((r_1', \rho_1''), (z_{(1,5)}, z_{(1,3)}, z_8))$ to prove that z_6, z_7, and z_8 are well-formed.
5. Each party locally computes $a_3 = \mathsf{Enc}(r_1(1 - 2b_1)c) + (2b_1 - 1)r_1') = z_7 \cdot z_8$.
6. P_2 computes $z_6' = \mathsf{Enc}(b_1 b_2) = z_{(1,1)}^{b_2} \cdot \mathsf{Enc}(0, \alpha_2)$, $z_7' = a_3^{1-2b_2} \cdot \mathsf{Enc}(0, \alpha_3)$, where α_2 and α_3 are newly selected as randomness during encryption, $z_8' = (z_7')^{r_2}$, and $z_9' = z_{(2,5)}^{r_2'}$, and sends to $P_1 z_6', z_7', z_8'$, and z_9'. P_2 executes $\mathsf{PKPM}((b_2, \rho_2 + \alpha_2), (z_{(1,1)}, z_{(2,1)}, z_6'))$, $\mathsf{PKPM}((1 - 2b_2 \bmod N, \rho_2^{-2} + \alpha_3), (a_3, z_{(2,4)}, z_7'))$, $\mathsf{PKPM}((r_2, \rho_2'), (z_7', z_{(2,2)}, z_8'))$, and $\mathsf{PKPM}((r_2', \rho_2''), (z_{(2,5)}, z_{(2,3)}, z_9'))$ to prove that z_6', z_7', z_8', and z_9' are well-formed.
7. Each party locally computes $a_3' = z_8' \cdot z_9'$.
8. The parties decrypt a_3'. If the decrypted value is $<N^2/2$, output $z_{(1,1)} \cdot z_{(2,1)} \cdot (z_6')^{-2}$; otherwise, output $e_1 (z_6')^2 \cdot (z_{(1,1)} \cdot z_{(2,1)})^{-1}$.

Our protocol closely follows the logic of the semi-honest solution, where we additionally need to employ zero-knowledge proofs to provide each party with the ability to verify that the computation was performed correctly by the other party. We do not explicitly compute both a_1' and a_2' as in the semi-honest version, but rather compute and return either a_1' or a_2' in step 8 after decrypting the content of a_3'. We also explicitly keep track of random values used for creating ciphertexts and use them as input into ZKPKs. This ensures that the protocol is fully specified.

Our security result can be stated as given next. Recall that we use a hybrid model with ideal decryption, which we then replace with a real instantiation of threshold decryption.

Theorem 4 *Assuming semantic security of the homomorphic encryption and security of the building blocks,* MalLT *protocol is secure in the presence of a malicious adversary in the hybrid model with ideal decryption.*

Proof We prove security of MalLT based on Definition 2. We separately consider the cases when P_1 is malicious and when P_2 is malicious. In the ideal world, we build a simulator S_j that is located between malicious party P_j and the TTP and simulates P_j's view of the protocol after querying the TTP for P_j's output.

First, we treat the case of malicious P_1 and build the corresponding simulator S_1. Upon obtaining the input $\mathsf{Enc}(x), \mathsf{Enc}(y), \ell, S_1$ queries the TTP for $\mathsf{Enc}(b)$. In step 1, S_1 performs the same computation as P_2. In step 2, S_1 acts as a verifier for P_1's ZKPKs PK12 and RangeProofs and extracts P_1's input to the proofs. It also chooses a random bit w and computes $\mathsf{Enc}(b^*) = \mathsf{Enc}(b)^{1-w} \cdot (e_1(\mathsf{Enc}(b))^{-1})^w = \mathsf{Enc}(b \cdot (1-w) + (1-b) \cdot w) = \mathsf{Enc}(b \oplus w)$. In other words, if $w = 0$ then $b^* = b$ and otherwise $b^* = 1 - b. S_1$ similarly computes $\mathsf{Enc}(b_2) = \mathsf{Enc}(b^* \oplus b_1)$ and chooses and encrypts two random values r_2 and r'_2 according to the protocol. S_1 now simulates P_2's proofs PK12 and RangeProof using $\mathsf{Enc}(b_2), \mathsf{Enc}(r_2)$, and $\mathsf{Enc}(r'_2)$. In steps 3–5, S_1 acts like P_2, i.e., S_1 performs the prescribed computation in steps 3 and 5 and acts as a verifier for P_1's proofs PKPMs during step 4. During step 6, S_1 computes $z'_6 = \mathsf{Enc}(b_2)^{b_1} \cdot \mathsf{Enc}(0, \beta_1)$ and $z'_7 = a_3^{1-2w'} \cdot \mathsf{Enc}(0, \beta_2)$, where w' is a newly generated random bit and β_1, β_2 correspond to freshly chosen randomness during encryption. S_1 also computes the remaining z'_8 and z'_9 according to the protocol. Note that using random w' in place of b_2 makes the content of ciphertexts z'_7 and z'_8 inconsistent with other encrypted values, but P_1 cannot tell this fact due to security of the encryption scheme. S_1 then simulates the PKPM proofs in step 6. Finally, in step 7, S_1 sends a positive properly chosen value $\hat{c} \in [0, N^2/2)$ to P_1 if $w = 0$, and a negative properly chosen value $\hat{c} \in [N^2/2, N^2)$ otherwise. To form $\hat{c}$, S_1 randomly samples the distribution of the absolute value of $x - y$ (using application-specific knowledge of distributions of x and y); let the randomly chosen value be denoted by $d. S_1$ then sets $\hat{c} = (r_1 \cdot d - r'_1)r_2 - r'_2$ if $w = 0$ and $\hat{c} = N^2 - (r_1 \cdot d - r'_1)r_2 + r'_2$ otherwise. Note that ciphertexts z'_7 and z'_8 are not used by the simulator beyond step 6 and thus their contents do not affect correctness of the output.

To show that the view simulated by S_1 results in P_1 obtaining correct output, we note that $w = b_1 \oplus b_2 \oplus b$ (because $b_1 = b_2 \oplus b \oplus w$). Now, if $w = 0, b = b_1 \oplus b_2 = b_1 + b_2 - 2b_1b_2$. Thus, S_1 needs to produce a positive value ($<N^2/2$) as the decryption of a'_3 in step 8, so that $z_{(1,1)} \cdot z_{(2,1)} \cdot (z'_6)^{-2} = \mathsf{Enc}(b_1 + b_2 - 2b_1b_2) = \mathsf{Enc}(b)$ is produced as the output in step 8. Otherwise, if $w = 1, b = 1 \oplus b_1 \oplus b_2 = 1 - (b_1 \oplus b_2) = 1 - b_1 - b_2 + 2b_1b_2$. Thus, S_1 needs to produce a negative value ($\geq N^2/2$) as the decryption of a'_3 in step 8, so that P_1 uses $e_1(z'_6)^2 \cdot (z_{(1,1)}z_{(2,1)})^{-1} = \mathsf{Enc}(1 + 2b_1b_2 - b_1 - b_2) = \mathsf{Enc}(b)$ as the output.

To show that the view simulated by S_1 is indistinguishable from P_1's view in the hybrid model (with ideal decryption), we note that most values the simulator sends to P_1 (e.g., steps 2 and 6) are encrypted and thus are indistinguishable from ciphertexts sent during the protocol execution because of semantic security of the encryption scheme. Similarly, the simulations of PK12, RangeProofs, and PKPMs are indistinguishable from real execution due to their security properties. The only value that S_1 provides to P_1 in the clear is the decryption of a'_3 in step 7. This value was chosen by S_1 in the same way as during the protocol after randomly sampling the absolute value of $x - y$ according to what is known about distributions of x and y. Thus, P_1 is unable to distinguish the value received during the simulation from the value received during the protocol execution. We also note that during the simulation S_1 aborts the computation in exactly the same circumstances when the computation is aborted in the real execution (namely, when a ZKPK does not verify) and thus the simulation cannot be distinguished from real execution on the grounds of computation termination.

Now let P_2 be malicious. We build a simulator S_2 that constructs P_2's view similar to the way S_1 did for P_1. Most of S_2's computations are the same as S_1's computations, and thus we concentrate on the differences. In this case, S_2 computes $\mathsf{Enc}(b^*)$ in step 2 in the same way S_1 did (i.e., by choosing a random bit w and using $b^* = b \oplus w$) and then sets $\mathsf{Enc}(b_1) = \mathsf{Enc}(b^* \oplus b_2)$. In step 4, S_2 selects a random bit w' and a random number ξ_1 to be used as randomness during encryption and computes $z_6 = \mathsf{Enc}(c)^{1-2w'} \cdot \mathsf{Enc}(0, \xi_1). S_2$ also computes z_7 and z_8 according the protocol and simulates the PKPM proofs (step 4). Note that using w' instead of b_1 (which S_2 doesn't know) in the computation of z_6 makes the content of ciphertexts z_7, a_3, z'_7, z'_8, and a'_3 inconsistent with other encrypted values, but P_2 is unable to tell this fact because of security of the encryption scheme. In steps 5 and 6, S_2 acts like P_1. Finally, in step 7, S_2 provides a positive properly chosen value $\tilde{c} \in [0, N^2/2)$ to P_2 if w was 0, and otherwise a negative properly chosen value $\tilde{c} \in [N^2/2, N^2)$. Note that none of incorrectly formed ciphertexts ($z_6, z_7, a_3, z'_7, z'_8$, and a'_3) are used in the computation of the protocol's output and correctness of the result is not affected.

Correctness of the output that P_2 learns at the end of S_2's simulation can be shown in a way similar to that of S_1's simulation. In other words, we now also have that $w = b_1 \oplus b_2 \oplus b$ and producing a positive value as the decryption of a_3' when $w = 0$ and producing a negative value when $w = 1$ results in P_2 computing $\mathsf{Enc}(b)$.

What remains to show is that the view simulated by S_2 is indistinguishable from P_2's view in the hybrid model. Similar to S_1's case, we have that all values that S_2 sends to P_2 are either protected via semantically secure encryption (steps 2 and 6), are simulated ZKPKs, or a plaintext $\tilde{c}$ chosen in the same way as $\hat{c}$ in S_1 simulation and is indistinguishable from the value decrypted during real protocol execution. Thus, assuming security of the building blocks, the claim follows.

8.5 Secure two-party truncation in the malicious model

In this section we are going to describe our (probabilistic) truncation protocol $\mathsf{MalTruncPR}$ secure in the malicious model. Our starting point was the semi-honest $\mathsf{TruncPR}$ from [10], which we adjusted to the two-party setting.[4] We provide the modified version of the protocol in the "Appendix". On input of an ℓ-bit encrypted x and a positive integer $k < \ell$, the protocol computes $\lfloor x/2^k \rfloor + b$, where b is either 0 or 1. In other words, the protocol truncates k least significant bits of x, but might also increment the result by 1.

At high level, the solution proceeds by the parties jointly and privately choosing two random values r' and r'' of bitlength k and $\ell - k + \kappa$, respectively, where κ is a statistical security parameter. The parties then blind x by $(\kappa + \ell)$-bit random number $r = 2^k r'' + r'$ and decrypt the sum $c = x + r$. The encrypted output y is computed as $(x + r'' - (c \bmod 2^k))2^{-k}$.

We next present our $\mathsf{MalTruncPR}$ protocol. As before, we follow the logic of the semi-honest protocol, but need to employ stronger building blocks and ZKPKs.

$\mathsf{Enc}(y) \leftarrow \mathsf{MalTruncPR}(\mathsf{Enc}(x), \ell, k)$

Public inputs include public key $pk = (g, N, \theta)$ for a (2, 2)-threshold Paillier encryption scheme and private inputs consist of shares of the corresponding secret key.

1. Each P_j randomly chooses $r'_{(j,i)} \in \{0, 1\}$ for $i \in [1, k]$, computes $z_{(j,i)} = \mathsf{Enc}(r'_{(j,i)}, \rho_{(j,i)})$ using randomness $\rho_{(j,i)}$, sends to the other party each $z_{(j,i)}$, and executes $\mathsf{PK12}((r'_{(j,i)}, \rho_{(j,i)}), (z_{(j,i)}, 0, 1))$ to prove that $z_{(j,i)}$ encrypts a bit.
2. The parties compute $z_i' = \mathsf{Enc}(r_i) = \mathsf{Enc}(r'_{(1,i)} \oplus r'_{(2,i)}) = z_{(1,i)} \cdot z_{(2,i)} \cdot (\mathsf{MalMul}(r'_{(1,i)}, r'_{(2,i)}))^{-2}$ for $i \in [1, k]$.
3. Each party locally computes $\mathsf{Enc}(r') = \mathsf{Enc}(\sum_{i=1}^{k} r_i^{2^i}) = \prod_{i=1}^{k} (z_i')^{2^i}$.
4. Each P_j randomly chooses $r_j'' \in [0, 2^{\ell+\kappa-k} - 1]$, computes $z_j'' = \mathsf{Enc}(r_j'', \rho_j')$, sends to the other party z_j'', and executes $\mathsf{RangeProof}((r_j'', \rho_j'), (0, 2^{\ell+\kappa-k} - 1, z_j''))$ to prove that z_j'' is well-formed.
5. Each party locally computes $\mathsf{Enc}(r'') = \mathsf{Enc}(r_1'' + r_2'') = \mathsf{Enc}(r_1'') \cdot \mathsf{Enc}(r_2'')$.
6. The parties locally compute $\mathsf{Enc}(c) = \mathsf{Enc}(x + 2^k r'' + r') = \mathsf{Enc}(x) \cdot \mathsf{Enc}(r'')^{2^k} \cdot \mathsf{Enc}(r')$ and jointly decrypt c.
7. Each party locally computes $c'' = \lfloor \frac{c}{2^k} \rfloor$ and and produces $\mathsf{Enc}(y) = \mathsf{Enc}(c'' - r'') = \mathsf{Enc}(c'', 0) \cdot \mathsf{Enc}(r'')^{-1}$ as the output.

One significant difference from the semi-honest protocol is the way random k-bit value r' is generated. In the semi-honest version, r' is set to the sum $r_1' + r_2'$, where r_i' is a $(k-1)$-bit random value chosen by P_i. To make this stronger for the malicious model, we could enforce that each party chooses its respective r_i' from the correct range using a range proof. Unfortunately, this is not sufficient for security. In the malicious model, the parties are not guaranteed to draw random values uniformly at random from the specified range and we can no longer expect that the sum $r_1' + r_2'$ is k bits long. Suppose that a malicious party P_i sets its r_i' to 0, which guarantees that the sum r' is $k - 1$ bits long. Then after the sum $c = x + 2^k r'' + r'$ is decrypted, the adversary can learn unintended information about the kth bit of x. In particular, if the kth bit of c is 0, the malicious party knows that the kth bit of x is 0. To eliminate this vulnerability, we instead require that both participants select their r_i''s to be k bits long and r' is computed via XOR as $r_1' \oplus r_2'$.

Another conceptual difference from the semi-honest solution is that instead of using $c \bmod 2^k$ in computing the result, the parties now use $\lfloor c/2^k \rfloor$. This simplifies computation of the output, but results in identical outcome. As before, we explicitly keep track of random values used for creating ciphertexts and use them as input into ZKPKs.

We next show security of this protocol.

Theorem 5 *Assuming semantic security of the homomorphic encryption and security of the building blocks,* $\mathsf{MalTruncPR}$ *protocol is secure in the presence of a malicious adversary in the hybrid model with ideal decryption.*

Proof We prove security of $\mathsf{MalTruncPR}$ based on Definition 2. When P_j is malicious, we need to construct simulator S_j that provides a view for P_j in the ideal world, which is indistinguishable from the protocol execution in the hybrid model. In what follows, without loss of generality, let us assume that P_1 is malicious; a very similar proof can be given for the case of malicious P_2 because of the protocol's symmetry.

[4] We also note that $\mathsf{TruncPR}$ in [10] was designed to work on both positive and negative integers, while in our case supporting only non-negative integers is sufficient.

In step 1, S_1 acts similar to what the protocol prescribes for P_2: it receives P_1's ciphertexts $z_{(1,i)}$'s, acts as a verifier for P_1's ZKPKs (extracting P_1's inputs), forms P_2's random bits and corresponding ciphertexts, and acts as a prover in ZKPKs to show their correctness. During step 2, S_1 invokes MalMul's simulator. In step 4, S_1 receives z''_1 from P_1 and acts as a verifier for P_1's $\mathsf{RangeProof}$ for z''_1 (extracting r''_1). S_1 also chooses random $\hat{c} \in \{0,1\}^{\ell+\kappa}$, computes $\tilde{c} = \hat{c} + 2^k r''_1$ and $z''_2 = \mathsf{Enc}(\lfloor \hat{c}/2^k \rfloor - y) = \mathsf{Enc}(\lfloor \hat{c}/2^k \rfloor)\mathsf{Enc}(y)^{-1}$, sends z''_2 to P_1, and simulates the $\mathsf{RangeProof}$ for z''_2. In step 6, S_1 outputs $\tilde{c}$ as the decrypted value. We note that $\tilde{c}$ is formed by the simulator inconsistently with the values used for computing r'. This is not a problem because P_1 does not use r' in producing its output and inconsistency of encrypted values cannot be detected as well.

To see that P_1 obtains the correct (encrypted) output at the end of S_1's simulation, recall that S_1 sets $z''_2 = \mathsf{Enc}(\lfloor \hat{c}/2^k \rfloor - y)$ in step 4. This means that P_1 computes in step 5 encryption of $r'' = \lfloor \hat{c}/2^k \rfloor - y + r''_1$. P_1 also learns $c = \tilde{c} = \hat{c} + 2^k r''_1$ in step 6 and consequently sets $c'' = \lfloor \hat{c}/2^k \rfloor + r''_1$. P_1 then sets the (encrypted) output to $c'' - r'' = \lfloor \hat{c}/2^k \rfloor + r''_1 - (\lfloor \hat{c}/2^k \rfloor - y + r''_1) = y$, as desired.

To show that the view simulated by S_1 is indistinguishable from the view in the hybrid model execution, we note that indistinguishability of encrypted data and all building blocks (i.e., ZKPKs, and MalMul) follows security of the building blocks. The only value revealed to P_1 in the clear is $c = \hat{c}$ in step 6. The value produced by the simulator, however, is statistically indistinguishable from the value of c used during real execution (using statistical security parameter κ). In addition, both during the simulation and real execution the computation aborts in identical circumstances when the malicious party fails to correctly complete ZKPKs as the prover. Thus, indistinguishability of simulated and real views follows.

8.6 Secure two-party inversion in the malicious model

The next protocol that we treat is computation of a multiplicative inverse of an encrypted integer x, where x is treated as a group element. As before, our starting point was a semi-honest inversion protocol, which we adapt to the two-party setting and list in the "Appendix". The main idea of this protocol is for the parties to jointly generate a random element r of the group, compute and decrypt $c = r \cdot x$, invert plaintext c, and then compute the inverse of x as $r \cdot c^{-1} = x^{-1}$ in the encrypted form.

Our protocol in the malicious model follows the logic of the semi-honest solution, but we modify the way $\mathsf{Enc}(rx)$ is computed from $\mathsf{Enc}(x)$. In particular, instead of having the parties compute $\mathsf{Enc}(r)$ and call multiplication on $\mathsf{Enc}(r)$ and $\mathsf{Enc}(x)$, we avoid calling relatively costly MalMul. We instead have each party P_j compute (and prove correctness of) $\mathsf{Enc}(x)^{r_j} = \mathsf{Enc}(r_j x)$ for its respective share r_j of r. The parties then locally compute $\mathsf{Enc}(rx) = \mathsf{Enc}(r_1 x + r_2 x)$ and proceed with the rest of the protocol as before.

$\mathsf{Enc}(y) \leftarrow \mathsf{MalInv}(\mathsf{Enc}(x))$

Public inputs include public key $pk = (g, N, \theta)$ for a $(2,2)$-threshold Paillier encryption scheme and private inputs consist of shares of the corresponding secret key.

1. Each P_j chooses at random $r_j \in \mathbb{Z}^*_N$, computes $z_j = \mathsf{Enc}(r_j, \rho_j)$ using fresh randomness ρ_j, sends z_j to the other party, and executes $\mathsf{PKP}((r_j, \rho_j), (z_j))$ to prove that z_j was formed correctly.
2. Each P_j computes $z'_j = \mathsf{Enc}(r_j x) = \mathsf{Enc}(x)^{r_j}$, sends z'_j to the other party, and executes $\mathsf{PKPM}((r_j, \rho_j), (\mathsf{Enc}(x), z_j, z'_j))$ to prove correctness of z'_j.
3. Each party locally computes $\mathsf{Enc}(c) = \mathsf{Enc}((r_1 x + r_2 x) = \mathsf{Enc}(r_1 x) \cdot \mathsf{Enc}(r_2 x)$ and the parties jointly decrypt c.
4. Each party locally computes and outputs $\mathsf{Enc}(y) = \mathsf{Enc}((r_1 + r_2)c^{-1}) = (z_1 z_2)^{c^{-1}}$.

Security of this protocol is stated as follows:

Theorem 6 *Assuming semantic security of the homomorphic encryption and security of the building blocks,* MalInv *protocol is secure in the presence of a malicious adversary in the hybrid model with ideal decryption.*

Proof We prove security of MalInv based on Definition 2. Because the protocol is symmetric, we assume without loss of generality that P_1 is malicious and build the corresponding simulator S_1. In the beginning of the protocol (step 1), S_1 receives z_1, chooses a random number $\hat{c} \in \mathbb{Z}^*_N$, computes $z_2 = \mathsf{Enc}(\hat{c} \cdot y - r_1) = \mathsf{Enc}(y)^{\hat{c}} \cdot z_1^{-1}$ using output $\mathsf{Enc}(y)$ received from the TTP, and sends z_2 to P_1. S_1 also simulates its PKP proof and acts as a verifier for P_1's proof obtaining r_1. In step 2, S_1 receives z'_1, chooses a random number $r_2 \in \mathbb{Z}^*_N$, computes $z'_2 = \mathsf{Enc}(r_2 x) = \mathsf{Enc}(x)^{r_2}$, and sends z'_2 to P_1. Both parties also execute their respective PKPM proofs, where S_1 uses simulation. Note that now z_2 and z'_2 have inconsistent contents, but this fact is not known to P_1 due to security of encryption. In step 3, S_1 output $\hat{c}$ as the result of decryption.

To show that P_1 computes correct output $\mathsf{Enc}(x^{-1})$, recall that the simulator outputs $c = \hat{c}$ and P_1 computes the result as $(z_1 z_2)^{c^{-1}}$. In the simulated view, we have $(z_1 z_2)^{c^{-1}} = (\mathsf{Enc}(r_1) \cdot \mathsf{Enc}(\hat{c} y - r_1))^{\hat{c}^{-1}} = \mathsf{Enc}(\hat{c} \cdot y \cdot \hat{c}^{-1}) = \mathsf{Enc}(y)$, as desired.

To show that the view simulated by S_1 is indistinguishable from the execution view in the hybrid model, notice that all information that P_1 receives is indistinguishable in both views due to security of the underlying building blocks with the exception of plaintext c that P_1 learns in step 3, which we need to analyze. During the simulation, S_1 outputs $\hat{c}$ chosen uniformly at random from the group. In the real execution, P_1 learns $(r_1 + r_2)x$, which is also a random element of the group. Thus, the values produced in the two worlds are indistinguishable. Lastly, in both worlds the execution aborts only when the malicious party fails to correctly complete ZKPKs, which completes this proof.

8.7 Secure two-party prefix multiplication in the malicious model

We next present prefix multiplication protocol, which on input of integers $x_1, \ldots, x_k$, outputs $y_1, \ldots, y_k$, where each $y_i = \prod_{j=1}^{i} x_j$. We provide the semi-honest prefix multiplication protocol adapted to the two-party setting from [10] in the "Appendix". We used the protocol as our starting point and modified it to be secure in the stronger security model with malicious participants.

The main idea behind PreMul protocol is for the parties to compute and open $\mathsf{Enc}(m_i) = \mathsf{Enc}(r_i \cdot x_i \cdot r_{i-1}^{-1})$ for $i \in [2, k]$ and $\mathsf{Enc}(m_1) = r_1 x_1$, where each r_i is a random element of the group and the revealed values completely hide each input x_i. Then, each party can compute the output as $y_i = r_i^{-1} \cdot (\prod_{j=1}^{i} m_j) = r_i^{-1} \cdot r_i \cdot x_i \cdot r_{i-1}^{-1} \cdots r_2 \cdot x_2 \cdot r_1^{-1} \cdot r_1 \cdot x_1$ in the encrypted form using encryptions of r_i^{-1}'s and plaintext m_i's. Each r_i is jointly chosen by the parties at random and computation of each r_i^{-1} proceeds similar to the inversion protocol. Namely, the parties also generate encryptions of random values s_i's, decrypt products $u_i = r_i \cdot s_i$, and use inverses of u_i's in the consecutive computation.

$\mathsf{Enc}(y_1), \ldots, \mathsf{Enc}(y_k) \leftarrow \mathsf{MalPreMul}(\mathsf{Enc}(x_1), \ldots, \mathsf{Enc}(x_k))$

Public inputs include public key $pk = (g, N, \theta)$ for a $(2, 2)$-threshold Paillier encryption scheme and private inputs consist of shares of the corresponding secret key.

1. Each P_j chooses $r_{(j,i)}, s_{(j,i)} \in \mathbb{Z}_N^*$ at random for $i \in [1, k]$, computes $z_{(j,i)} = \mathsf{Enc}(r_{(j,i)}, \rho_{(j,i)})$ and $z'_{(j,i)} = \mathsf{Enc}(s_{(j,i)}, \rho'_{(j,i)})$, and sends each $z_{(j,i)}$ and $z'_{(i,j)}$ to the other party. P_j also executes $\mathsf{PKP}((r_{(j,i)}, \rho_{(j,i)}), (z_{(j,i)}))$ and $\mathsf{PKP}((s_{(j,i)}, \rho'_{(j,i)}), (z'_{(j,i)}))$ for each i to prove that $z_{(j,i)}$'s and $z'_{(j,i)}$'s are well-formed.
2. Each P_j locally computes $z_i = \mathsf{Enc}(r_i) = \mathsf{Enc}(r_{(1,i)} + r_{(2,i)}) = z_{(1,i)} \cdot z_{(2,i)}$ and $z'_i = \mathsf{Enc}(s_i) = \mathsf{Enc}(s_{(1,i)} + s_{(2,i)}) = z'_{(1,i)} \cdot z'_{(2,i)}$ for $i \in [1, k]$.
3. Each P_j computes $a_{(j,i)} = \mathsf{Enc}(r_i \cdot s_{(j,i)}) = (z_i)^{s_{(j,i)}}$ for $i \in [1, k]$, sends to the other party $a_{(j,i)}$'s, and executes $\mathsf{PKPM}((s_{(j,i)}, \rho'_{(j,i)}), (z_i, z'_{(j,i)}, a_{(j,i)})$ to prove that each $a_{(j,i)}$ is well-formed.
4. Each P_j computes $b_{(j,i)} = \mathsf{Enc}(r_{i+1} \cdot s_{(j,i)}) = (z_{i+1})^{s'_{(j,i)}}$ for $i \in [1, k-1]$, sends to the other party $b_{(j,i)}$'s, and executes $\mathsf{PKPM}((s_{(j,i)}, \rho'_{(j,i)}), (z_{i+1}, z'_{(j,i)}, b_{(j,i)}))$ to prove that $b_{(j,i)}$ is well-formed.
5. The parties locally compute $\mathsf{Enc}(u_i) = \mathsf{Enc}(r_i \cdot s_i) = a_{1,i} \cdot a_{2,i}$ for $i \in [1, k]$ and jointly decrypt each u_i.
6. Each party locally computes $\mathsf{Enc}(v_i) = \mathsf{Enc}(r_{i+1} \cdot s_i) = b_{(1,i)} \cdot b_{(2,i)}$ for $i \in [1, k-1]$.
7. Each party locally sets $\mathsf{Enc}(w_1) = \mathsf{Enc}(r_1) = z_1$ and for $i \in [2, k]$ computes $\mathsf{Enc}(w_i) = \mathsf{Enc}(v_{i-1} \cdot (u_{i-1})^{-1}) = \mathsf{Enc}(v_{i-1})^{(u_{i-1})^{-1}}$.
8. Each party also locally computes $\mathsf{Enc}(t_i) = \mathsf{Enc}(s_i \cdot (u_i^{-1})) = (z'_i)^{(u_i)^{-1}}$ for $i \in [1, k]$.
9. For $i \in [1, k]$, the parties compute $\mathsf{Enc}(m_i) = \mathsf{MalMul}(\mathsf{Enc}(w_i), \mathsf{Enc}(x_i))$ and decrypt each m_i.
10. Each party sets $\mathsf{Enc}(y_1) = \mathsf{Enc}(x_1)$ and locally computes $\mathsf{Enc}(y_i) = \mathsf{Enc}(t_i \prod_{j=1}^{i} m_j) = (\mathsf{Enc}(t_i))^{\prod_{j=1}^{i} m_j}$ for $i \in [2, k]$ as the output.

The high-level logic of our solution is the same as in the semi-honest setting, but we modify how some encrypted values are computed to result in a faster solution. In particular, we avoid the use of the multiplication protocol for computing encrypted u_i's and v_i's and instead employ local multiplications and proofs of correctness using PKPM's. The computed values are the same, but the mechanism for their computation differs resulting in computational savings.

We next show security of this protocol:

Theorem 7 *Assuming semantic security of the homomorphic encryption and security of the building blocks,* $\mathsf{MalPreMul}$ *protocol is secure in the presence of a malicious adversary in the hybrid model with ideal decryption.*

Proof As before, we proceed according to the security notion from Definition 2 and build a simulator S_j that creates a view for P_j in the ideal model, which is indistinguishable from P_j's view in protocol's real execution. Because $\mathsf{MalPreMul}$ is symmetric, we assume without loss of generality that P_1 is malicious and build a corresponding simulator S_1.

In the beginning, S_1 submits inputs to the TTP and receives the output $\mathsf{Enc}(y_i)$'s. S_1 also chooses random $\hat{m}_i, d_i \in \mathbb{Z}_N^*$ and computes $\hat{u}_i = d_i(\prod_{j=1}^{i} \hat{m}_j)$ for $i \in [1, k]$. In step 1, S_1 receives $z_{(1,i)}$ and $z'_{(1,i)}$ from P_1 for each i. It chooses its own random $r_{(2,i)}$'s, encrypts them as $z_{(2,i)} = \mathsf{Enc}(r_{(2,i)}, \rho_{(2,i)})$, computes $z'_{(2,i)} = \mathsf{Enc}(y_i \cdot t_i - s_{(1,i)}) = \mathsf{Enc}(y_i)^{t_i} \cdot (z'_{(1,i)})^{-1}$, re-randomizes each $z'_{(2,i)}$ (by multiplying it to a fresh encryption of 0), and sends to P_1 each $z_{(2,i)}$ and $z'_{(2,i)}$. S_1 invokes simulator for its own and P_1's PKP proofs (extracting P_1's inputs). S_1 doesn't perform any computation in step 2. In step 3, S_1 receives $a_{(1,i)}$'s from P_1, chooses random elements $a_{(2,i)}$ from the ciphertext space, and sends these $a_{(2,i)}$'s to

P_1. S_1 uses simulation for PKPM interaction. Similarly, in step, S_1 receives $b_{(1,i)}$'s from P_1, chooses random elements $b_{(2,i)}$ from the ciphertext space, and sends these $a_{(2,i)}$'s to P_1. S_1 also uses simulation for PKPM interactions. Note that using random $a_{(2,i)}$'s and $b_{(2,i)}$'s makes the content of ciphertexts $\mathsf{Enc}(u_i)$ and $\mathsf{Enc}(v_i)$ in consecutive steps inconsistent with other encrypted values, but P_1 cannot tell this fact. In step 5, S_1 uses $\hat{u}_i$'s as decryptions and then skips steps 6–8. In step 9, S_1 invokes simulator for MalMul to interact with P_1, and provides $\hat{m}_i$'s to P_1 as decrypted values.

To show that P_1 computes correct output during S_1's simulation, first notice that each $\hat{u}_i = d_i(\prod_{j=1}^{i} \hat{m}_j)$ and thus $\prod_{j=1}^{i} \hat{m}_j = \hat{u}_i \cdot d_i^{-1}$, where $\hat{u}_i$'s and $\hat{m}_i$'s are used as u_i's and m_i's, respectively. In addition, the simulator sets $s_{(2,i)} = d_i \cdot y_i - s_{(1,i)}$, so that $s_i = s_{(1,i)} + s_{(2,i)} = d_i \cdot y_i$. Now, when P_1 computes the ith component of the output, it uses computation (on encrypted values) $t_i(\prod_{j=1}^{i} m_j) = s_i \cdot u_i^{-1}(\prod_{j=1}^{i} m_i) = d_i \cdot y_i \cdot \hat{u}_i^{-1}(\prod_{j=1}^{i} \hat{m}_i) = d_i \cdot y_i \cdot \hat{u}_i^{-1} \cdot \hat{u}_i \cdot d_i^{-1} = y_i$, as required.

To show that the view simulated by S_1 is indistinguishable from P_1's view in the hybrid model, we only need to show that plaintexts u_i's and m_i's that the simulator outputs do not violate indistinguishability, as the remaining portions of the protocol are indistinguishable because of the assumption that all building blocks and ZKPKs are secure. Similarly, indistinguishability cannot be violated if the execution aborts in the ideal or real model, but not in the other because the only time the execution terminates in either world is when the malicious party does not follow the computation and fails to complete a ZKPK. Regarding the release of $\hat{m}_i$'s and $\hat{u}_i$'s by the simulator, we first note that the release of $\hat{m}_i$'s only reveals no information to P_1 because each $\hat{m}_i$ was chosen uniformly at random. Each $\hat{u}_i$, on the other hand, is a function of $\hat{m}_i$'s, but each u_i was randomized by a new random value d_i and thus $\hat{u}_i$ is also a random element of the group. In the real protocol execution, each u_i is formed as $r_i \cdot s_i$ and each m_i (except m_1) is formed as $r_i \cdot x_i \cdot r_{i-1}^{-1}$, which are also distributed as random elements of the group. Thus, we obtain that P_1 cannot tell the difference between the simulated and real protocol execution with a non-negligible probability.

8.8 Secure two-party bit decomposition in the malicious model

Finally, we describe our last, bit decomposition, protocol secure in the malicious model. Our starting point was the bit composition protocol in the semi-honest setting from [11], which we adapted to the two-party setting and provide its two-party version in the "Appendix". On input of an ℓ-bit encrypted integer a, the protocol performs bit decomposition of k least significant bits of a.

The main idea of BitDec protocol for the parties to compute $\mathsf{Enc}(c) = \mathsf{Enc}(2^{\ell+k} + a - r)$, where r is a random $(\ell + \kappa)$-bit value and the k least significant bits of r are available to the parties in encrypted form, and decrypt c. The plaintext lets each party to compute the bits of $2^{\ell+\kappa} + a - r$ while providing statistical hiding of a. The random r is created in the same way as in the truncation protocol, where the parties separately create k least significant bits of r and choose a single random r' for the remaining bits of r. The parties then call a protocol called BitAdd that takes k least significant (plaintext) bits of c and k least significant (encrypted) bits of r and performs addition of the values provided by their bitwise representation (i.e., addition of two k-bit quantities). BitAdd outputs k encrypted bits of the sum, which are used as the output of the BitDec protocol.

In our MalBitDec protocol we need to employ a stronger version of BitAdd, which was provided for the semi-honest setting. We, however, notice that BitAdd is composed entirely of addition and multiplication operations [11] and we can obtain a protocol secure in the malicious model, which we denote by MalBitAdd, by employing protocol MalMul in place of ordinary multiplications. Adding two integers x and y in bitwise form involves computing sum and carry bits s_i and e_i, which can be sequentially computed as $e_0 = x_0 \wedge y_0 = x_0 \cdot y_0, s_0 = x_0 \oplus y_0 = x_0 + y_0 - 2e_0$, and $e_i = (x_i \wedge y_i) \vee ((x_i \oplus y_i) \wedge e_{i-1}) = x_i \cdot y_i + (x_i \oplus y_i)e_{i-1}, s_i = x_i + y_i + e_{i-1} - 2e_i$ for $i \geq 1$. Bitwise addition protocol [37] used to implement bit decomposition uses concurrent execution to compute all bits of the sum (and carry bits) using a smaller (than linear in the size of the input) number of rounds, but still implements the formulas given above. This will be relevant for our security proof.

$\mathsf{Enc}(x_{k-1}), \ldots, \mathsf{Enc}(x_0) \leftarrow \mathsf{MalBitDec}(\mathsf{Enc}(a), \ell, k)$

Public inputs include public key pk for a $(2, 2)$-threshold Paillier encryption scheme and private inputs consist of shares of the corresponding secret key.

1. For $i \in [0, k-1]$, each P_j chooses random bits $r_{(j,i)} \in \{0, 1\}$, encrypts them as $z_{(j,i)} = \mathsf{Enc}(r_{(j,i)}, \rho_{(j,i)})$, and sends each $z_{(j,i)}$ to the other party. P_j also executes $\mathsf{PK12}((r_{(j,i)}, \rho_{(j,i)}), (z_{(j,i)}, 0, 1))$ to prove that each $z_{(j,i)}$ is well-formed.
2. The parties compute $z_i = \mathsf{Enc}(r_i) = \mathsf{Enc}(r_{(1,i)} \oplus r_{(2,i)})) = \mathsf{Enc}(r_{(1,i)} + r_{(2,i)} - 2r_{(1,i)}r_{(2,i)}) = z_{(1,i)} \cdot z_{(2,i)} \cdot (\mathsf{MalMul}(z_{(1,i)}, z_{(2,i)}))^{-2}$ for $i \in [0, k-1]$.
3. Each P_j chooses random $r'_j \in [0, 2^{\ell+\kappa-k} - 1]$, encrypts it as $z'_j = \mathsf{Enc}(r'_j, \rho'_j)$, and sends it to the other party. P_j also executes $\mathsf{RangeProof}((r'_j, \rho'_i)(0, 2^{\ell+\kappa-k} - 1, z'_j))$ to prove that z'_j is well-formed.

4. Each party locally computes $\mathsf{Enc}(r) = \mathsf{Enc}(2^k(r'_1 + r'_2) + \sum_{i=0}^{k-1} r_i \cdot 2^i) = (z'_1 \cdot z'_2)^{2^k} \prod_{i=0}^{k-1} z_i^{2^i}$ and $\mathsf{Enc}(c) = \mathsf{Enc}(2^{\ell+\kappa+1} + a - r) = \mathsf{Enc}(2^{\ell+\kappa+1}, 0) \cdot \mathsf{Enc}(a) \cdot \mathsf{Enc}(r)^{-1}$.
5. The parties jointly decrypt $\mathsf{Enc}(c)$ to learn c.
6. The parties compute and output $(\mathsf{Enc}(x_{k-1}), \ldots, \mathsf{Enc}(x_0)) = \mathsf{MalBitAdd}((c_{k-1}, \ldots, c_0), (\mathsf{Enc}(r_{k-1}), \ldots, \mathsf{Enc}(r_0)))$, where $c_0, \ldots, c_{k-1}$ are k least significant bits of c.

Our protocol closely follows the logic of the semi-honest solution. We employ zero-knowledge proofs to verify that the computation was performed correctly and building blocks secure in the stronger security model. We show security of this protocol as follows:

Theorem 8 *Assuming semantic security of the homomorphic encryption and security of the building blocks,* MalBitDec *protocol is secure in the presence of a malicious adversary in the hybrid model with ideal decryption.*

Proof We prove security of MalBitDec based on Definition 2. We construct P_j's view in the ideal model by building simulator S_j, and we show it is indistinguishable form view of P_j in protocol's real execution. We assume without loss of generality that P_1 is malicious, and we build simulator S_1. We can use similar proof in case P_2 is malicious because MalBitDec is symmetric.

In step 1, S_1 receives $z_{(1,i)}$'s, and acts as a verifier for PK12's (extracting $r_{(1,i)}$'s). S_1 then chooses a random number $\tilde{c} \in \{0,1\}^{\ell+\kappa}$ and computes

1. $\mathsf{Enc}(r_i) = \mathsf{Enc}(x_i - \tilde{c}_i) = \mathsf{Enc}(x_i) \cdot \mathsf{Enc}(-\tilde{c}_i)$ for $i \in [2, k-1]$, where $\tilde{c}_i$ denotes ith least significant bit of $\tilde{c}$,
2. $\mathsf{Enc}(r_1) = \mathsf{Enc}(x_1 - \tilde{c}_1 - \tilde{c}_0 r_0)$ (if $k > 1$) as $\mathsf{Enc}(x_1) \cdot \mathsf{Enc}(c_1) \cdot \mathsf{Enc}(r_0)^{-1}$ if $\tilde{c}_0 = 1$ and $\mathsf{Enc}(x_1) \cdot \mathsf{Enc}(c_1)$ otherwise, and
3. $\mathsf{Enc}(r_0) = \mathsf{Enc}(x_0 \oplus \tilde{c}_0)$ as $\mathsf{Enc}(x_0) \cdot \mathsf{Enc}(0)$ if $\tilde{c}_0 = 0$ and $\mathsf{Enc}(1 - x_0) = \mathsf{Enc}(x_0)^{-1} \cdot \mathsf{Enc}(1)$ otherwise

using fresh randomness for each newly formed encryption. Note that as a result of this computation the value that r_i takes may no longer be a bit (e.g., when $x_i = 0$ and $\tilde{c}_i = 1$ for $i \geq 2$). For each $i \in [0, k-1]$, if $r_{(1,i)} = 0$, S_1 computes $z_{(2,i)} = \mathsf{Enc}(r_i) \cdot \mathsf{Enc}(0)$ and otherwise computes $z_{(2,i)} = \mathsf{Enc}(1 - r_i) = \mathsf{Enc}(r_i)^{-1} \cdot \mathsf{Enc}(1)$ (using a freshly formed encryption of 0 or 1). S_1 then sends each $z_{(2,i)}$ to P_1 and simulates PK12's as a prover. During step 2, S_1 uses MalMul's simulator to produce P_1's view. In step 3, S_1 follows the protocol similar to P_2's computation: it receives z'_1, verifies P_1's RangeProof (extracting r'_1), produces r'_2 and its corresponding ciphertext, and acts as a prover in RangeProof. S_1 skips step 4. In step 5, S_1 outputs $\tilde{c} + 2^{\ell+\kappa} - 2^k r'_1$ as decrypted value. As a result, in the consecutive step MalBitAdd will be called on k least significant bits of $\tilde{c}$ and k ciphertexts $\mathsf{Enc}(r_i)$. In step 6, S_1 uses MalBitAdd's simulator to interact with P_1, but introduces changes in the simulation. In particular, S_1 forces each encrypted carry bit (for $i \geq 1$) to become 0 as follows. The computation in MalBitAdd consists of computing $p_i = x_i + y_i - 2x_i y_i$ and $g_i = x_i y_i$ for each bit i of inputs x and y, followed by computing carry bits as $e_0 = g_0$ and $e_i = g_i + p_i e_{i-1}$ for $i \in [1, k-1]$ (the sum bits are computed from x_i's, y_i's and e_i's as previously described). Because one of the arguments to MalBitAdd is given in the plaintext form, computation of p_i's and g_i's is local and beyond the simulator's control. Computing each e_i (for $i \geq 1$), however, involves a call to MalMul, where we instruct S_1 to deviate from MalMul's simulation. In particular, when S_1 simulates P_1's view during a call to $\mathsf{MalMul}(\mathsf{Enc}(p_i), \mathsf{Enc}(e_{i-1}))$, instead of setting z_2 to $\mathsf{Enc}(p_i e_{i-1})^{-1} \mathsf{Enc}(e_{i-1})^{w-r_1}$ in step 2 as MalMul's simulation prescribes, S_1 sets z_2 to $\mathsf{Enc}(g_i) \mathsf{Enc}(e_{i-1})^{w-r_1}$. This will cause the product to evaluate to $-g_i$ and consequently result in e_i being 0 for each $i \geq 1$. (We note that e_i's are not computed sequentially in BitAdd to reduce the number of rounds, but this does not affect how we instruct the simulator to work.) The remaining multiplications are simulated according to MalMul's simulator.

Now we show that P_1's output at the end of simulation is computed correctly. During the simulation, S_1 sets each $r_{2,i}$ such that $r_{(2,i)} = r_i \oplus r_{(1,i)}$ and consequently $r_i = r_{(1,i)} \oplus r_{(2,i)}$, where $r_i = x_i - \tilde{c}_i$ for $i \geq 2$, $r_0 = x_0 \oplus \tilde{c}_0$, and $r_1 = x_1 - \tilde{c}_1 - \tilde{c}_0 r_0$. Then because k least significant bits of $c = \tilde{c} + 2^{\ell+\kappa} - 2^k r'_1$ correspond to k least significant bits $\tilde{c}$, MalBitAdd is going to be called on arguments $(\tilde{c}_{k-1}, \ldots, \tilde{c}_0)$ and $(\mathsf{Enc}(r_{k-1}), \ldots, \mathsf{Enc}(r_0))$. As part of $\mathsf{MalBitAddP_1}$ then computes the carry bit e_0 as $\tilde{c}_0 r_0$, while all other carry bits e_i for $i \geq 1$ will be forced to be 0 (by changing what MalMul returns) as described earlier. Recall that the output bits of MalBitAdd (and the output bits of BitDec are computed as $s_0 = \tilde{c}_0 + r_0 - 2e_0$ and $s_i = \tilde{c}_i + r_i + e_{i-1} - 2e_i$. Because all $e_i = 0$ for $i \geq 1$, but e_0 can be set to 1, we obtain that

1. $s_0 = \tilde{c}_0 + r_0 - 2\tilde{c}_0 r_0 = \tilde{c}_0 \oplus r_0 = x_0$ as required;
2. s_1 is supposed to be computed as $s_1 = \tilde{c}_1 + r_1 + e_0 - 2e_1$, but we instead have $s_1 = \tilde{c}_1 + r_1 + e_0$. Recall, however, that r_1 was set to $r_1 = x_1 - \tilde{c}_1 - \tilde{c}_0 r_0 = x_1 - \tilde{c}_1 - e_0$, which gives us $s_1 = \tilde{c}_1 + x_1 - \tilde{c}_1 - e_0 + e_0 = x_1$ as required;
3. s_i for $i \geq 2$ becomes $\tilde{c}_i + r_i$ as a result of S_1's simulation. Because r_i was set to $x_i - \tilde{c}_i$, we obtain that $s_i = \tilde{c}_i + x_i - \tilde{c}_i = x_i$ as required as well.

The last piece that we wanted to demonstrate is that P_1 will compute each r_i according to the value that S_1 expects even when r_i is not a bit (which would be a violation of the real

Table 1 Complexity of building blocks in the two-party setting

Protocol	Rounds	Communication size	Computation complexity	
			Client	Server
RangeProof	1	$6\log(H-L)C$	$5\log(H-L)$	$6\log(H-L)$
PK12	1	$4C$	3	2
PKPK	1	$2.5C$	2	2
PKPM	1	$4.5C$	4	4
MalMul	2	$13C+2D$	15	15
MalLT	4	$(52.5+36(\ell+\kappa))C+2D$	$43+33(\ell+\kappa)$	$43+33(\ell+\kappa)$
MalTruncPR	5	$(23k+12(\ell+\kappa-k)+2)C+(2k+2)D$	$4+11(\ell+\kappa-k)+22k$	$4+11(\ell+\kappa-k)+22k$
MalInv	2	$18C+2D$	18	18
MalPreMul	6	$44kC+6kD$	$45k-2$	$45k-2$
MalBitAdd	$2\log k$	$13k\log kC+2k\log kD$	$k(15\log k+2)$	$k(15\log k+2)$
MalBitDec	$2\log k+4$	$((13\log k+23)k+12(\ell+\kappa)-11)C$ $+(2\log k+2)kD$	$k(15\log k+24)$ $+11(\ell+\kappa)-10$	$k(15\log k+24)$ $+11(\ell+\kappa)-9$

protocol execution). During the simulation, r_0 is always computed as a bit, r_1 may take values -1 and -2 (in addition to 0 and 1), and r_i may take value -1 (in addition to 0 and 1). S_1 sets each $r_{2,i}$ as XOR of r_i and $r_{(1,i)}$ using the formula $r_i + r_{(1,i)} - 2r_i r_{(1,i)}$ (i.e., $r_{(2,i)}$ is either r_i or $1 - r_i$ based on the value of the bit $r_{(1,i)}$) and later P_1 computes $r_i = r_{(1,i)} + r_{(2,i)} - 2r_{(1,i)}r_{(2,i)}$. The crucial fact that we are using here is that $r_i \oplus r_{(1,i)} \oplus r_{(1,i)} = r_i$ for any value of r_i as long as $r_{(1,i)}$ is a bit. In other words, during the simulation $r_{(2,i)} = r_i$ and then $r_i = r_{(2,i)}$ when $r_{(1,i)} = 0$; and $r_{(2,1)} = 1 - r_i$ and then $r_i = 1 - r_{(2,i)} = 1 - (1 - r_i) = r_i$ when $r_{(1,i)} = 1$. We conclude that P_1 learns the correct (encrypted) output bits $x_0, \ldots, x_{k-1}$ as a result of this simulation.

To show that the view simulated by S_1 is indistinguishable from P_1's view in the hybrid model, we need to show the plaintext value the simulator produces in step 5 is indistinguishable from the value c in real execution (as the remaining building blocks have been shown to guarantee indistinguishability and all encrypted values achieve indistinguishability as well). Recall that in the real protocol execution c is formed as $2^{\ell+\kappa+1} - r + x = 2^{\ell+\kappa+1} - 2^k(r_1' + r_2') - \sum_{i=0}^{k-1} 2^i r_i + x$, while in the simulation S_1 outputs $c = \tilde{c} + 2^{\ell+k} - 2^k r_1'$. Let $\tilde{c} = 2^{\ell+\kappa} - 2^k r_2' - \sum_{i=0}^{k-1}$. Because no information about r_2' and r_i's is available to $P_1, \tilde{c}$ in the simulation and $2^{\ell+\kappa} - 2^k r_2' - \sum_{i=0}^{k-1}$ during real execution are distributed identically. We obtain that during the real execution P_1 observes $2^{\ell+\kappa+1} - r + x$, while during the simulation P_1 observes $2^{\ell+\kappa+1} - r$. These two values are statistically indistinguishable using statistical security parameter κ. Note that we have to take the value of r_1' into account when forming c during the simulation to ensure that c that the simulator outputs falls in the correct range (according to P_1's knowledge of r_1'). Lastly, we note that both during the simulation and real execution the computation aborts only when a malicious party fails to correctly complete ZKPKs as the prover. Therefore, simulated and real views are indistinguishable.

With MalMul, MalLT, RangeProof, MalTruncPR, MalInv, MalPreMul, MalBitDec, and previously mentioned prior work, we achieve security of the HMM and GMM protocols in the malicious model in the two-party setting. The complexities of these protocols are provided in Table 1. In Table 1, notation C denotes the ciphertext length in bits, and D denotes the length of the auxiliary decryption information, which when sent by one of the parties allows the other party to decrypt a ciphertext. Other notations are parameters of the protocols. Communication is measured in bits, and computation is measured in full size modulo exponentiations. We list computational overhead incurred by each party separately, with the smaller amount of work first (which can be carried out by a client) followed by the larger amount of work (which can be carried out by a server).

9 Conclusions

In this work, we treat the problem of privacy-preserving Hidden Markov Models computation which is commonly used for many applications including speaker recognition. We develop provably secure techniques for HMM's Viterbi and GMM computation using floating-point arithmetic in both two-party setting using homomorphic encryption and multi-party setting using secret sharing. These settings correspond to a variety of real-life situations and the solutions were designed to minimize their overhead.

A significant part of this work is dedicated to new secure protocols for floating-point operations in the malicious model in the two-party setting. To the best of our knowledge, this is the first time such protocols are offered in the literature. We rigorously prove security of our protocols using simulation-based proofs, which constitutes a distinct contribution of this work.

Acknowledgments We are grateful to Yihua Zhang for help with the multi-party experiments. This work was supported in part by grants CNS-1223699 and CNS-1319090 from the National Science Foundation and FA9550-13-1-0066 from the Air Force Office of Scientific Research. Any opinions, findings, and conclusions or recommendations expressed in this publication are those of the authors and do not necessarily reflect the views of the funding agencies.

Appendix: Additional two-party protocols in the semi-honest model

In this section, we provide four protocols: probabilistic truncation TruncPR, inversion Inv, prefix multiplication PreMul, and bit decomposition BitDec secure in the semi-honest setting. All of these protocols have been modified from their original versions to the two-party setting using homomorphic encryption, but the structure of the computation remains unchanged. In all cases it is assumed that the inputs are nonnegative integers.

We first describe TruncPr protocol adapted from its original version in [10]. On input of $\mathsf{Enc}(x)$, ℓ, and k, the protocol outputs $\mathsf{Enc}(y) = \mathsf{Enc}(\lfloor x/2^k \rfloor + b)$, where b is a (random) bit. High-level description of the protocol is given in Sect. 8.5.

$\mathsf{Enc}(y) \leftarrow \mathsf{TruncPR}(\mathsf{Enc}(x), \ell, k)$

Public inputs include public key $pk = (g, N, \theta)$ for a (2, 2)-threshold Paillier encryption scheme and private inputs consist of shares of the corresponding secret key.

1. Each P_i chooses $r'_i \in \{0,1\}^{k-1}$ and $r''_i \in \{0,1\}^{\ell+\kappa-k-1}$ at random and sends to the other party $a_{(i,1)} = \mathsf{Enc}(r'_i)$ and $a_{(i,2)} = \mathsf{Enc}(r''_i)$.
2. Each party computes $\mathsf{Enc}(r') = \mathsf{Enc}(r'_1 + r'_2) = \mathsf{Enc}(r'_1) \cdot \mathsf{Enc}(r'_2)$ and $\mathsf{Enc}(r'') = \mathsf{Enc}(r''_1 + r''_2) = \mathsf{Enc}(r''_1) \cdot \mathsf{Enc}(r''_2)$.
3. The parties compute $\mathsf{Enc}(c) = \mathsf{Enc}(x + 2^k r'' + r') = \mathsf{Enc}(x) \cdot \mathsf{Enc}(r'')^{2^k} \cdot \mathsf{Enc}(r')$ and decrypt c.
4. Each party locally computes $c' = c \bmod 2^k$ and produces $\mathsf{Enc}(y) = \mathsf{Enc}((x - c' + r') \cdot (2^k)^{-1}) = (\mathsf{Enc}(x) \cdot \mathsf{Enc}(c')^{-1} \cdot \mathsf{Enc}(r'))^{(2^k)^{-1}}$ as the output.

The above protocol assumes that $k \geq 2$. When $k = 1$, each P_i instead chooses r'_i as a random bit in step 1, and in step 2 the parties compute $\mathsf{Enc}(r') = \mathsf{Enc}(r'_1 \oplus r'_2) = \mathsf{Enc}(r'_1) \cdot \mathsf{Enc}(r'_2) \cdot (\mathsf{Mul}(\mathsf{Enc}(r'_1), \mathsf{Enc}(r'_2)))^{-2}$. The rest of the protocol remains unaffected.

The second protocol describes two-party computation of multiplicative inverse of x, where x is assumed to be a nonzero element of the group. High-level description of this protocol is given in Sect. 8.6.

$\mathsf{Enc}(y) \leftarrow \mathsf{Inv}(\mathsf{Enc}(x))$

Public inputs include public key $pk = (g, N, \theta)$ for a (2, 2)-threshold Paillier encryption scheme and private inputs consist of shares of the corresponding secret key.

1. Each P_i chooses $r_i \in \mathbb{Z}_N^*$, and sends to the other party $a_i = \mathsf{Enc}(r_i)$.
2. Each P_i locally computes $\mathsf{Enc}(r) = \mathsf{Enc}(r_1 + r_2) = \mathsf{Enc}(r_1) \cdot \mathsf{Enc}(r_2)$.
3. The parties compute $\mathsf{Enc}(c) = \mathsf{Mul}(\mathsf{Enc}(x), \mathsf{Enc}(r))$ and decrypt c.
4. Each party locally computes $\mathsf{Enc}(y) = \mathsf{Enc}(c^{-1} r) = (\mathsf{Enc}(r))^{c^{-1}}$ as the output.

The next protocol that we illustrate is two-party prefix multiplication PreMul, which is based on multi-party PreMulC from [10]. High-level description of this protocol is given in Sect. 8.7.

$\mathsf{Enc}(y_1), \ldots, \mathsf{Enc}(y_k) \leftarrow \mathsf{PreMul}(\mathsf{Enc}(x_1), \ldots, \mathsf{Enc}(x_k))$

Public inputs include public key $pk = (g, N, \theta)$ for a (2, 2)-threshold Paillier encryption scheme and private inputs consist of shares of the corresponding secret key.

1. Each P_j chooses random $r_{(j,i)}, s_{(j,i)} \in \mathbb{Z}_N^*$ for $i \in [1, k]$, computes $z_{(j,i)} = \mathsf{Enc}(r_{(j,i)})$, $z'_{(j,i)} = \mathsf{Enc}(s_{(j,i)})$, and sends each $z_{(j,i)}$ and $z'_{(j,i)}$ to the other party.
2. Each party locally computes $z_i = \mathsf{Enc}(r_i) = \mathsf{Enc}(r_{(1,i)} + r_{(2,i)}) = z_{(1,i)} z_{(2,i)}$ and $z'_i = \mathsf{Enc}(s_i) = \mathsf{Enc}(s_{(1,i)} + s_{(2,i)}) = z'_{(1,i)} z'_{(2,i)}$ for $i \in [1, k]$.
3. The parties compute $\mathsf{Enc}(u_i) = \mathsf{Enc}(r_i \cdot s_i) = \mathsf{Mul}(z_i, z'_i)$ for $i \in [1, k]$ and decrypt each u_i.
4. The parties compute $\mathsf{Enc}(v_i) = \mathsf{Enc}(r_{i+1} \cdot s_i) = \mathsf{Mul}(z_{i+1}, z'_i)$ for $i \in [1, k-1]$.
5. Each party sets $\mathsf{Enc}(w_1) = z_1$ and computes $\mathsf{Enc}(w_i) = \mathsf{Enc}(v_{i-1} \cdot (u_{i-1})^{-1}) = \mathsf{Enc}(v_{i-1})^{(u_{i-1})^{-1}}$ for $i \in [2, k]$.
6. Each party also locally computes $\mathsf{Enc}(t_i) = \mathsf{Enc}(s_i \cdot (u_i)^{-1}) = (z_i)^{(u_i)^{-1}}$ for $i \in [1, k]$.
7. The parties compute $\mathsf{Enc}(m_i) = \mathsf{Mul}(\mathsf{Enc}(w_i), \mathsf{Enc}(x_i))$ for $i \in [1, k]$ and decrypt each m_i.
8. Each party sets $\mathsf{Enc}(y_1) = \mathsf{Enc}(x_1)$ and locally computes $\mathsf{Enc}(y_i) = \mathsf{Enc}(t_i \cdot \prod_{j=1}^{i} m_j) = (\mathsf{Enc}(t_i))^{\prod_{j=1}^{i} m_j}$ for $i \in [2, k]$ as the output.

The last protocol that we are going to describe here is bit decomposition BitDec, which originally appeared in [11] for the multi-party setting and modified it to work in our two-party setting based on homomorphic encryption. A high-level description of the protocol can be found in Sect. 8.8.

$\mathsf{Enc}(x_{k-1}), \ldots, \mathsf{Enc}(x_0) \leftarrow \mathsf{BitDec}(\mathsf{Enc}(a), \ell, k)$

Public inputs include public key pk for a $(2, 2)$-threshold Paillier encryption scheme and private inputs consist of shares of the corresponding secret key.

1. For $i \in [0, k-1]$, each P_j chooses a random bit $r_{(j,i)}$, computes $z_{(j,i)} = \mathsf{Enc}(r_{(j,i)})$, and sends each $z_{(j,i)}$ to the other party.
2. The parties compute $z_i = \mathsf{Enc}(r_i) = \mathsf{Enc}(r_{(1,i)} \oplus r_{(2,i)}) = \mathsf{Enc}(r_{(1,i)} + r_{(2,i)} - 2r_{(1,i)}r_{(2,i)}) = z_{(1,i)} \cdot z_{(2,i)} \cdot \mathsf{Mul}(r_{(1,i)}, r_{(2,i)})^{-2}$ for $i \in [0, k-1]$.
3. Each P_j chooses $r'_j \in \{0, 1\}^{\ell+\kappa-k}$ at random, computes $z'_j = \mathsf{Enc}(r_j)$, and sends z'_j to the other party.
4. Each party locally computes $\mathsf{Enc}(r) = \mathsf{Enc}(2^k(r'_1 + r'_2) + \sum_{i=0}^{k-1}(2^i \cdot r_i)) = (z'_1 \cdot z'_2)^{2^k} \prod_{i=0}^{k-1} z_i^{2^i}$ and $\mathsf{Enc}(c) = \mathsf{Enc}(2^{\ell+\kappa} + a - r) = \mathsf{Enc}(2^{\ell+\kappa}) \cdot \mathsf{Enc}(a) \cdot \mathsf{Enc}(r)^{-1}$.
5. The parties jointly decrypt $\mathsf{Enc}(c)$ to learn c.
6. The parties compute and output $(\mathsf{Enc}(x_{k-1}), \ldots, \mathsf{Enc}(x_0)) = \mathsf{BitAdd}((c_{k-1}, \ldots, c_0), (\mathsf{Enc}(r_{k-1}), \ldots, \mathsf{Enc}(r_0)))$, where $c_0, \ldots, c_{k-1}$ are k least significant bits of c.

References

1. Aliasgari, M., Blanton, M.: Secure computation of hidden markov models. In: International Conference on Security and Cryptography (SECRYPT) (2013)
2. Aliasgari, M., Blanton, M., Zhang, Y., Steele, A.: Secure computation on floating point numbers. In: Network and Distributed System Security Symposium (NDSS) (2013)
3. Asharov, G., Lindell, Y., Rabin, T.: Perfectly-secure multiplication for any $t < n/3$. In: CRYPTO (2011)
4. Bansal, P., Kant, A., Kumar, S., Sharda, A., Gupta, S.: Improved hybrid model of HMM/GMM for speech recognition. In: Book 5 Intelligent Technologies and Applications. Institute of Information Theories and Applications FOI ITHEA (2008)
5. Baudron, O., Fouque, P.-A., Pointcheval, D., Stern, J., Poupard, G.: Practical multi-candidate election scheme. In: ACM Symposium on Principles of Distributed Computing (PODC), pp. 274–283 (2001)
6. Blanton, M., Aguiar, E.: Private and oblivious set and multiset operations. Int. J. Inf. Secur. **15**, 1–26 (2016)
7. Blanton, M., Gasti, P.: Secure and efficient protocols for iris and fingerprint identification. In: European Symposium on Research in Computer Security (ESORICS), pp. 190–209 (2011)
8. Camenisch, J., Stadler, M.: Proof systems for general statements about discrete logarithms. Technical Report TR260, Institute for Theoretical Computer Science, ETH Zurich (1997)
9. Canetti, R.: Security and composition of multiparty cryptographic protocols. J. Cryptol. **13**(1), 143–202 (2000)
10. Catrina, O. and de Hoogh, S.: Improved primitives for secure multiparty integer computation. In: Security and Cryptography for Networks (SCN), pp. 182–199 (2010)
11. Catrina, O., Saxena, A.: Secure computation with fixed-point numbers. In: Financial Cryptography and Data Security (FC), pp. 35–50 (2010)
12. CertiVox: Multiprecision Integer and Rational Arithmetic Cryptographic Library (MIRACL). http://www.certivox.com/miracl/
13. Cramer,R., Damgård, I., Nielsen, J.: Multiparty computation from threshold homomorphic encryption. In: Advances in Cryptology—EUROCRYPT, pp. 280–289 (2001)
14. Damgård, I., Ishai, Y., Krøigaard, M.: Perfectly secure multiparty computation and the computational overhead of cryptography. In: Advances in Cryptology—EUROCRYPT, pp. 445–465 (2010)
15. Damgård, I., Jurik, M.: A generalisation, a simplification and some applications of Paillier's probabilistic public-key system. In: International Workshop on Practice and Theory in Public Key Cryptography (PKC), pp. 119–136 (2001)
16. Damgård, I., Nielsen, J.: Universally composable efficient multiparty computation from threshold homomorphic encryption. In: Advances in Cryptology—CRYPTO, pp. 247–264 (2003)
17. Franz, M.: Secure Computations on Non-integer Values. Ph.D. thesis, TU Darmstadt (2011)
18. Franz, M., Deiseroth, B., Hamacher, K., Jha, S., Katzenbeisser, S., Schröder, H.: Towards secure bioinformatics services (short paper). In: Financial Cryptography and Data Security (FC), pp. 276–283. Springer, New York (2012)
19. Gennaro, R., Rabin, M., Rabin, T.: Simplified VSS and fast-track multiparty computations with applications to threshold cryptography. In: ACM Symposium on Principles of Distributed Computing (PODC), pp. 101–111 (1998)
20. Goldreich, O.: Foundations of Cryptography: Basic Applications, vol. 2. Cambridge University Press, Cambridge (2004)
21. GMP—The GNU Multiple Precision Arithmetic Library. http://www.gmplib.org
22. Goldreich, O., Oren, Y.: Definitions and properties of zero-knowledge proof systems. J. Cryptol. **7**(1), 1–32 (1994)
23. Kerschbaum, F., Biswas, D., de Hoogh, S.: Performance comparison of secure comparison protocols. In: International Workshop on Database and Expert Systems Application (DEXA), pp. 133–136 (2009)
24. Lipmaa, H., Asokan, N., Niemi, V.: Secure Vickrey auctions without threshold trust. In: Financial Cryptography (FC), pp. 87–101 (2002)
25. Matsui, T., Furui, S.: Speaker adaptation of tied-mixture-based phoneme models for text-prompted speaker recognition. In: IEEE International Conference on Acoustics, Speech, and Signal Processing (ICASSP), vol. 1, pp. 125–128 (1994)
26. Nguyen, H., Roughan, M.: Multi-observer privacy-preserving hidden markov models. In: Network Operations and Management Symposium (NOMS), pp. 514–517 (2012)
27. Nguyen, H., Roughan, M.: On the identifiability of multi-observer hidden markov models. In: International Conference on Acoustics, Speech and Signal Processing (ICASSP), pp. 1873–1876 (2012)
28. OpenSSL: The Open Source Toolkit for SSL/TLS. http://www.openssl.org
29. Paillier, P.: Public-key cryptosystems based on composite degree residuosity classes. In: Advances in Cryptology—EUROCRYPT, pp. 223–238 (1999)
30. Pathak, M., Portelo, J., Raj, B., Trancoso, I.: Privacy-preserving speaker authentication. In: Information Security Conference (ISC), pp. 1–22 (2012)
31. Pathak, M., Raj, B.: Privacy preserving speaker verification using adapted GMMs. In: Interspeech, pp. 2405–2408 (2011)
32. Pathak, M., Raj, B., Rane, S., Saragdis, P.: Privacy-preserving speech processing: cryptographic and string-matching frameworks show promise. IEEE Signal Process. Mag. **30**(2), 62–74 (2013)
33. Pathak, M., Rane, S., Sun, W., Raj, B.: Privacy preserving probabilistic inference with hidden Markov models. In: Interna-

tional Conference on Acoustics, Speech and Signal Processing (ICASSP), pp. 5868–5871 (2011)

34. Peng, K., Bao, F.: An efficient range proof scheme. In: IEEE International Conference on Information Privacy, Security, Risk and Trust (PASSAT), pp. 826–833 (2010)
35. Polat, H., Du, W., Renckes, S., Oysal, Y.: Private predictions on hidden Markov models. Artif. Intell. Rev. **34**(1), 53–72 (2010)
36. Rabiner, L.: A tutorial on hidden Markov-models and selected applications in speech recognition. IEEE Proc. **77**(2), 257–286 (1989)
37. Secure Supply Chain Management (SecureSCM) Project Deliverable: D9.2 Security Analysis (2009)
38. Shamir, A.: How to share a secret. Commun. ACM **22**(11), 612–613 (1979)
39. Shashanka, M.: A privacy preserving framework for Gaussian mixture models. In: IEEE International Conference on Data Mining Workshops (ICDMW), pp. 499–506. IEEE (2010)
40. Smaragdis, P., Shashanka, M.: A framework for secure speech recognition. IEEE Trans. Audio Speech. Lang. Process. **15**(4), 1404–1413 (2007)
41. Zhang, Y., Steele, A., Blanton, M.: PICCO: a general-purpose compiler for private distributed computation. In: ACM Conference on Computer and Communications Security (CCS), pp. 813–826 (2013)