Privacy-Preserving HMM Forward Computation

Jan Henrik Ziegeldorf, Jan Metzke, Jan Rüth, Martin Henze, Klaus Wehrle
Communication and Distributed Systems (COMSYS), RWTH Aachen University, Germany
{ziegeldorf, metzke, rueth, henze, wehrle}@comsys.rwth-aachen.de

ABSTRACT

In many areas such as bioinformatics, pattern recognition, and signal processing, Hidden Markov Models (HMMs) have become an indispensable statistical tool. A fundamental building block for these applications is the Forward algorithm which computes the likelihood to observe a given sequence of emissions for a given HMM. The classical Forward algorithm requires that one party holds both the model and observation sequences. However, we observe for many emerging applications and services that the models and observation sequences are held by different parties who are not able to share their information due to applicable data protection legislation or due to concerns over intellectual property and privacy. This renders the application of HMMs infeasible. In this paper, we show how to resolve this evident conflict of interests using secure two-party computation. Concretely, we propose *Priward* which enables two mutually untrusting parties to compute the Forward algorithm securely, i.e., without requiring either party to share her sensitive inputs with the other or any third party. The evaluation of our implementation of *Priward* shows that our solution is efficient, accurate, and outperforms related works by a factor of 4 to 126. To highlight the applicability of our approach in real-world deployments, we combine *Priward* with the widely used HMMER biosequence analysis framework and show how to analyze real genome sequences in a privacy-preserving manner.

1. INTRODUCTION

Hidden Markov Models (HMMs) are used to model discrete stochastic processes whose internal state cannot be observed. They have become an indispensable statistical tool in many application areas, ranging from natural language processing and pattern recognition to bioinformatics or even finance and economics. An important algorithm associated with HMMs is the *Forward algorithm* which computes the probability that a given HMM generated a given sequence of observations. Forward computation is not only an integral part during training HMMs, but is also ubiquitously used in all application areas to score how well a given HMM explains the observed stochastic process. In bioinformatics, e.g., the Forward algorithm is applied to efficiently analyze the similarity of genome sequences [1].

CODASPY'17, March 22 - 24, 2017, Scottsdale, AZ, USA

ISBN 978-1-4503-4523-1/17/03…$15.00

DOI: http://dx.doi.org/10.1145/3029806.3029816

Traditionally, both the HMM and the observation sequences are held by the same entity. However, we observe that a second scenario arises in which the HMM and the observation sequences are held by two parties that are unwilling or even forbidden to share this information with one another. For example, consider a service that offers HMM-based genetic disease testing. Clearly, it requires significant research efforts to build accurate disease models. To remain competitive, the service must hence protect its intellectual property. At the same time, users have valid security and privacy concerns which prevent them from sharing sensitive information such as their DNA or medical test results with untrusted services. We find similar examples in other application areas, e.g., biometric identification [39], location services [45], or speech processing [35].

This evident conflict of business interests, regulatory issues, and privacy concerns leads to the question whether two parties can compute the HMM Forward algorithm without either party learning the other's input? Secure multi-party computation [24] presents a promising answer. However, we face two difficult challenges in practice: performance and accuracy. Secure computations typically require large numbers of cryptographic and interactive operations that may well cause unreasonable overheads in real-world applications. Achieving accuracy is challenging as Forward computation entails computation over very small probabilities, which is known to cause problems even with plaintext floating point arithmetic [38].

Related work faces these challenges in different ways: [36] uses efficient blinding techniques together with a semi-trusted third party. The authors of [21, 35, 40] independently propose to compute in logspace using homomorphic encryption and oblivious lookup tables [21]. Techniques for secure floating point arithmetic are proposed in [4, 17]. Unfortunately, none of these approaches provides both sufficient accuracy and satisfiable performance in real-world use cases, hence leaving secure Forward computation an open problem.

In this paper, we propose *Priward* that allows two mutually untrusting parties to *efficiently* and *accurately* compute the Forward algorithm while remaining *oblivious* to each other's inputs. Our approach is based on a combination of additive secret sharing and garbled circuits that is secure against semi-honest adversaries. We carry out all computations in logspace using fixed-point precision which achieves sufficient accuracy and low runtimes for real-world problem instances. The following are our main contributions:

Problem Analysis: We first motivate the need for secure computation on HMMs and then compile a set of requirements from real-world use cases. Our rigorous analysis of related works reveals problems and pitfalls inherent to their design that need to be avoided in any secure computation on probabilities.

Secure HMM Forward Computation: We propose *Priward*, an efficient secure two-party computation protocol for the Forward algorithm. Thereby, we provide efficient and accurate techniques

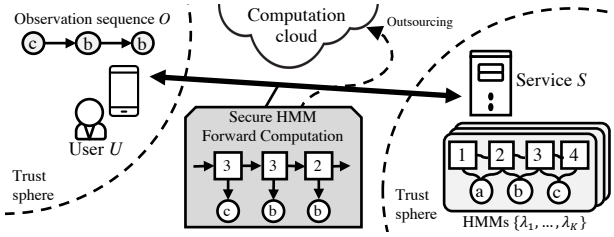

Figure 1: Our problem scenario: User $\mathcal{U}$ holds an observation sequence O, e.g., a genome string, while service $\mathcal{S}$ holds a database of HMMs $\{\lambda_1, ..., \lambda_K\}$, e.g., encoding certain diseases. Neither party wants to share her input with the other (trust spheres). *Priward* enables both parties to compute the Forward algorithm (gray box) in this setting while remaining oblivious to each other's inputs, and even to securely outsource computations to an untrusted cloud.

for computing in logspace that are relevant beyond the scope of this work. Our thorough evaluation shows that *Priward* is efficient and outperforms related work by a factor of 4 to 126. Additionally, our approach enables resource constrained devices that still cannot cope with the involved overheads to outsource computations (e.g., to an untrusted compute cloud) without loss of security.

Real-World Use Case: To showcase the applicability of *Priward*, we implement a secure version of the widely used HMMER biosequence analysis framework [1] and demonstrate privacy-preserving matchings against the Pfam protein families database [3]. Our results show that *Priward* achieves high accuracy and feasible overheads even on large real-world HMMs and long observation sequences.

This paper is structured as follows. Section 2 motivates the scenario, distils important requirements, and identifies limitations of previous proposals. We give a primer on secure computations and the cryptographic building blocks used in this paper in Section 3. Section 4 presents *Priward*, our approach to secure Forward computation. In Section 5, we present a general evaluation of *Priward* and show how to realize a real-world bioinformatics use case in a secure and privacy-preserving manner. Section 6 concludes this paper.

2. PROBLEM STATEMENT

Computations using HMMs have so far assumed that a single party holds all inputs, e.g., the model and observation sequences. We motivate an emerging problem scenario (Sect. 2.1) where the model and observation sequences are held by two different parties that do not trust each other. We proceed to establish concise requirements for Forward computation in this scenario based on real-world applications (Sect. 2.2) and finally analyze related work with regards to these requirements (Sect. 2.3). Our analysis identifies significant limitations among previous approaches that leave privacy-preserving HMM Forward computation an important and open problem.

2.1 Scenario

We consider two parties, a user $\boldsymbol{\mathcal{U}}$ and a service $\boldsymbol{\mathcal{S}}$ as illustrated in Figure 1. $\mathcal{U}$ holds an observation sequence O, while $\mathcal{S}$ holds a database of HMMs $\{\lambda_1, ..., \lambda_K\}$. Together, $\mathcal{U}$ and $\mathcal{S}$ want to compute $P(O|\lambda_i)$, i.e., the probability that a certain model has generated the observation sequence. This scenario is ubiquitous in different application areas of HMMs. In genetic disease testing, e.g., $\mathcal{S}$ has a database of HMMs representing specific diseases against which $\mathcal{U}$ wants to match relevant parts of her sequenced genome to determine susceptibility to certain diseases. In speech recognition [38], each of $\mathcal{S}$'s HMMs is trained to recognize single words or short phrases, while $\mathcal{U}$ supplies a sequence of spoken words to be transformed written text.

In traditional deployments, the desired results are computed with the Forward algorithm by one party which must obtain both inputs. However, in many emerging scenarios neither party is willing to share their inputs with the other or any third party, as indicated by the trust spheres in Figure 1. As a motivating example, consider that $\mathcal{S}$ offers a service and has invested significant research efforts to build high quality HMMs. Sharing the HMMs would give away the intellectual property that distinguishes $\mathcal{S}$ from other service providers [21]. On the other hand, $\mathcal{U}$ must carefully consider the privacy implications of sharing her personal data, especially for sensitive data such as DNA [7], voice recordings [35] or locations [45]. Finally, legal requirements may forbid to share models or observation sequences, e.g., HMMs might be trained on private patient data that must not be shared even in aggregated form.

In this paper, we thus pose the question how $\mathcal{U}$ and $\mathcal{S}$ can compute the desired results *obliviously*, i.e., without learning each other's inputs. Such a solution not only reconciles the evident conflict of business and privacy interests but enables new services that are hitherto prevented by applicable privacy legislation. Remaining oblivious to $\mathcal{U}'s$ sensitive inputs, the service provider $\mathcal{S}$ also does not have to fear the negative consequences of disclosure of customer data in case of attacks, database leaks, or seizure by governments.

2.2 Requirements

We survey how HMMs are used in real-world applications and distil a set of requirements for secure Forward computation.

Requirement 1 (Performance). Efficiency of Forward computation is of high importance in many real-world use cases. In speech processing, e.g., typically one five-state HMM is used to recognize a single word or utterance [38]. Thus, the performance of Forward computation directly determines the size of the vocabulary that, e.g., a voice command system can efficiently recognize. In contrast, profile HMMs used for sequence alignments in bioinformatics often have hundreds of states with observation sequences of equal lengths [3]. Although less time-sensitive than speech applications, Forward computation in bioinformatics must still be efficient to handle the involved large problem instances in adequate time. To ensure the applicability of secure Forward computation in its many diverse use cases, the first requirement is to minimize overheads w.r.t. the size and number of HMMs and observation sequences.

Requirement 2 (Outsourcing). The increasing number of *mobile* users, e.g., using speech-to-text services, poses additional challenges to secure Forward computation. Due to limitations on processing, bandwidth, or energy, mobile users may not be able to carry out computations themselves. It is thus highly desirable to allow outsourcing computations to more capable peers, e.g., to an untrusted computation cloud (cf. Fig. 1). To present a real alternative to constrained users, outsourcing must be both very efficient and uphold all security guarantees.

Requirement 3 (Accuracy). Ideally, a secure HMM Forward algorithm should compute results identical to those of a standard insecure implementation on plaintexts. However, most cryptographic protocols operate over integers and require heavy quantization to handle non-integers [15, 20] while those that operate over floats still introduce significant overheads [17]. Fortunately, inaccuracies may be tolerable to some degree in practice which allows us to strike a balance between performance and accuracy: In speech recognition, to recognize a word, we are only interested in the best guess, i.e., we compute only the $\arg\max$ while the exact probabilities are less important. In sequence alignments, the goal is to separate matching from non-matching models, i.e., Forward scores are only compared against a certain threshold. Since the actual required accuracy of the numerical results depends on the actual use case, we only require

that accuracy can be flexibly traded off against performance. This can often be achieved by adjusting the size of the HMMs or the length of the observation sequence. However, this usually requires expert knowledge and almost always involves expensive retraining of HMMs. We thus require a trade-off independent of the model and observation sequence that allows computation of either an accurate result when necessary or a quick approximation when sufficient.

Requirement 4 (Security). The capabilities of $\mathcal{U}$ and $\mathcal{S}$ to attack the computation are defined by the semi-honest model [23]: A *semi-honest* attacker correctly follows the protocol but may try to infer additional information from the protocol transcript. The semi-honest model has many applications and advantages: It allows for efficient protocols and protects against insider and outsider attacks when both parties are not actively cheating. This is reasonable to assume in our scenario since the user and service provider have a strong interest in executing the computation correctly. We further observe that data involved in typical HMM applications, e.g., in bioinformatics or speech, remains sensitive for longer periods of time. To sufficiently protect such data, we thus argue that all involved cryptographic primitives must be parameterized for long-term security, e.g., by choosing adequate key lengths as recommended in [8].

2.3 Analysis of Related Work

We discuss related work in chronological order and analyze the different approaches along our four requirements (cf. Table 1).

Smaragdis et al. [40] were first to consider privacy-preserving HMM computation in the context of speech recognition. Their approach is based on homomorphic encryption (HE) throughout. HE causes high performance overheads, especially for long-term security levels. Without an evaluation of performance overheads, it thus remains unclear whether their approach is practical. This is aggravated by the fact that many of their secure protocols, e.g., the inner product, require plaintext knowledge of the inputs, which prevents outsourcing Forward computation to an untrusted computation cloud. A further concern is the numerical stability of the results: Forward computation involves real numbers while the crypto primitives of their approach operate over the integers. The authors neither explain how they represent probabilities as integers nor do they quantify the involved errors and how they propagate. Finally, missing discussions of security and the use of insecure primitives render the overall security of their approach doubtful.

Pathak et al. [34,35] adapt and improve the techniques from [40] to similar applications such as keyword recognition and speaker identification. Their approach, as well, makes heavy use of HE which is expensive and scales poorly to long-term security levels. This is confirmed by their evaluation results. As in the approach by Smaragdis et al. [40], outsourcing is not possible since some subprotocols require plaintext knowledge. Interestingly, Pathak et al. show that a fixed-point representation can indeed achieve reasonable accuracy on small HMMs when normalizing forward probabilities in each iteration. However, it remains questionable whether reasonable accuracy can also be achieved on larger HMMs and observation sequences. It is also left unclear whether the authors fixed the security issues found in [40].

Polat et al. [36] present a different approach based on additive blindings which promises high efficiency. However, their approach uses additive blinding over probabilities as if they were integers from a finite field which raises serious concerns over the numerical accuracy of the computed results. The authors provide only a limited evaluation of the performance of their subprotocols on random inputs and do not analyze the achieved accuracy. As in the previous approaches and for the same reasons, outsourcing is not possible. Furthermore, they rely on a third party to generate correlated randomness for computing scalar products. If this third party colludes with either party, the other party's privacy is lost.

Approach	R1	R2	R3	R4
Smaragdis et al. [40]	○	○	○	◑
Pathak et al. [34, 35]	○	○	◑	◑
Polat et al. [36]	●	○	○	◑
Franz et al. [20, 21]	◑	◑	●	●
Aliasgari / Kamm / Demmler et al. [5, 17, 28]	○	●	◑	●
Priward (this paper)	●	●	●	●

Table 1: Comparison against related work by the requirements, R1) performance, R2) outsourcing, R3) accuracy, R4) security. ●, ◑, and ○ mark completely, partly, or unfulfilled requirements.

Franz et al. [20] propose a framework for secure computations on non-integer values in logarithmic representation which they apply to secure bioinformatics services [21]. Their approach is first to provide reasonable accuracy for computations on real-world HMMs and observation sequences. It uses lookup tables to compute the critical logsum operation that causes problems in the approaches by Smaragdis et al. [40] and Pathak et al. [34,35]. The size of the lookup tables constitutes a trade-off between performance (smaller tables) and accuracy (larger tables). However, the size of the lookup tables grows exponentially in the bit-length of inputs, which makes their solution rather inefficient when very accurate results are required. As this approach frequently relies on HE primitives, it scales poorly to long-term security levels and cannot be fully outsourced.

Aliasgari et al. [4,5], **Kamm et al. [28]** and **Demmler et al. [17]** propose provably secure floating point primitives in the multi- and two-party setting that can be fully outsourced. The proposed primitives could be used to implement the classical Forward algorithm in a secure manner but none of these works presents a concrete implementation. This leaves unclear whether standard IEEE 754 floating point numbers achieve sufficient accuracy or whether additional measures are required to avoid underflows of the very small probabilities involved in Forward computation [38]. Furthermore, the performance comparison of the proposed primitives presented in [17] indicates significant overheads.

Summary. The results of our analysis show that no approach provides a satisfying solution to the problem of computing the Forward algorithm in the secure two-party setting. Notably, almost all previous solutions depend on HE primitives and are thus subject to much higher overheads for long-term security levels. We conclude that the efficient, accurate, and secure computation of the HMM Forward algorithm is still an open and important problem.

3. CRYPTOGRAPHIC BUILDING BLOCKS

We give a brief overview of three secure two-party computation (STC) techniques, i.e., oblivious transfer, garbled circuits, and additive secret sharing, which are important to understand our approach and its advantages over related work.

Oblivious Transfer. Oblivious Transfer (OT) is an important building block for STC. OT is a protocol executed between a sender and a receiver. In the scope of this work, sender and receiver are always the service $\mathcal{S}$ and the user $\mathcal{U}$, respectively. The intuitive goal of OT is to allow $\mathcal{U}$ to choose exactly one of many secrets held by $\mathcal{S}$ without $\mathcal{S}$ learning $\mathcal{U}$'s choice and $\mathcal{U}$ learning the other secrets held by $\mathcal{S}$. In the most basic case, 1-2-OT, $\mathcal{S}$ holds two secret bits s_0 and s_1 while $\mathcal{U}$ holds a choice bit r. Running the OT protocol, $\mathcal{U}$ obtains exactly s_r and learns nothing about s_{1-r}, while $\mathcal{S}$ learns nothing about the choice r. As a short notation, we write $s_r \leftarrow \text{1-2-OT}(r, (s_0, s_1))$. 1-2-OT can be generalized to 1-n-OT$_l$, where $\mathcal{S}$ holds n l-bit secrets and $\mathcal{U}$ learns exactly one secret without

revealing her choice nor learning any of the other secrets. A batch of m parallel 1-n-OT$_l$ is denoted by 1-n-OT$_l^m$. We can efficiently instantiate 1-n-OT$_l^m$ by first reducing it to a batch of $m \cdot \log_2(n)$ runs of 1-2-OT$_l$ [32, 33] and then reducing this large number of long l-bit OTs to a small number of short t-bit *base OTs* [6, 27] (*OT Extension*), where t is the symmetric security parameter. The base OTs can be precomputed and their processing and communication overheads amortize quickly over a large number of OT Extensions.

Garbled Circuits. Yao's Garbled Circuits (GCs) [42] were the first generic STC protocol. It allows two parties $\mathcal{U}$ and $\mathcal{S}$ with private inputs x and y to evaluate a function $\mathcal{F}(x, y)$ without either party learning the other party's input. Yao's protocol runs in three rounds, i) garbling and transmitting the circuit, ii) garbling and transmitting the inputs, and iii) evaluating the GC and exchanging the results. First, the desired computation $\mathcal{F}(x, y)$ is represented as a Boolean circuit $\mathcal{F}_{Bool}(x, y)$ which is facilitated by specific compilers [17]. Party $\mathcal{S}$ then *garbles* the circuit by encrypting and permuting the truth table entries of each logic gate. $\mathcal{S}$ sends the GC $\tilde{\mathcal{F}}_{Bool}(\cdot)$ to $\mathcal{U}$. In the second step, $\mathcal{S}$ sends its own garbled input $\tilde{y}$ to $\mathcal{U}$, while $\mathcal{U}$ obtains her own garbled input $\tilde{x}$ from $\mathcal{S}$ via OT from $\mathcal{S}$. This ensures that $\mathcal{S}$ learns nothing about $\mathcal{U}$'s input x. Finally, $\mathcal{U}$ evaluates $\tilde{\mathcal{F}}_{Bool}(\tilde{x}, \tilde{y})$ by decrypting the GC gate by gate.

The processing and communication overheads of GCs are mainly determined by the circuit size. It is thus critical to construct *size-efficient* circuits. Circuit size is measured in the number of non-XOR gates, as XOR gates cause virtually no overhead due to the optimizations proposed in [30]. Different size-efficient circuit building blocks have been proposed in [29] and are partly used in this work.

Additive Secret Sharing. Additive secret sharing is an alternative STC technique [12, 18] that uses an *arithmetic* circuit representation, i.e., the desired functionality $\mathcal{F}(\cdot)$ is represented using addition and multiplication gates. An arithmetic circuit $\mathcal{F}_{arith.}(\cdot)$ over the ring $\mathbb{Z}_{2^l}$, i.e., all operations are modulo 2^l, is evaluated securely as follows. Both parties first share their input among each other, e.g., party $\mathcal{U}$ holding an input x draws a uniformly random $r \in_U \mathbb{Z}_{2^l}$ and sends $x^{\mathcal{S}} = x - r$ to $\mathcal{S}$ and keeps $x^{\mathcal{U}} = r$ as her own share (and vice versa if $\mathcal{S}$ shares to $\mathcal{U}$). Note that $x^{\mathcal{U}} + x^{\mathcal{S}} = x \mod 2^l$, thus we call the two shares $x^{\mathcal{U}/\mathcal{S}}$ an additive sharing of x. We denote the operation of creating shares from a plaintext input x by $x^{\mathcal{U}/\mathcal{S}} \leftarrow \texttt{Share}(x)$. Having shared their inputs, $\mathcal{U}$ and $\mathcal{S}$ then evaluate the arithmetic circuit representing the desired functionality f using only these shares. Note that addition gates can be evaluated locally, since $\mathbb{Z}_{2^l}$ preserves the commutative nature of additions, i.e., we have $x + y \mod 2^l = (x^{\mathcal{U}} + x^{\mathcal{S}}) + (y^{\mathcal{U}} + y^{\mathcal{S}}) \mod 2^l = (x^{\mathcal{U}} + y^{\mathcal{U}}) + (x^{\mathcal{S}} + y^{\mathcal{S}}) \mod 2^l$. In contrast, multiplication gates require an interactive protocol between $\mathcal{U}$ and $\mathcal{S}$, which can be sped up using precomputed multiplication triples (MTs) [9, 18]. Eventually, $\mathcal{U}$ and $\mathcal{S}$ obtain shares $r^{\mathcal{U}}, r^{\mathcal{S}}$ of the final result r which they exchange and add to obtain $r = r^{\mathcal{U}} + r^{\mathcal{S}} \mod 2^l$. We denote this operation by $r \leftarrow \texttt{Recombine}(r^{\mathcal{U}/\mathcal{S}})$.

Processing and communication overheads of this STC technique are dominated by the effort to generate the required MTs, i.e., by the number of multiplications. Similarly, the round complexity is determined by the multiplicative depth of the arithmetic circuit. Efficient high-level building blocks have been proposed in [14, 15]. Note that arithmetic circuits can also be evaluated using homomorphic encryption as many related works propose (cf. Sect. 2.3). However, STC over additive shares is more efficient for many tasks [18].

Hybrid STC. GCs are rooted in Boolean logic and are thus very efficient for logical operations such as comparisons. In contrast, the additive sharing approach is more efficient for arithmetic operations, i.e., addition and multiplication. Therefore, it is reasonable to ask whether the two representations can be combined to build efficient hybrid STCs. This idea was first rigorously followed by the *Tasty* framework [25] and recently improved by the *ABY* framework [18]. The basic idea of these approaches is to build efficient conversion protocols and execute required operations in the most efficient representation. A rigorous performance evaluation and different use cases are presented in [18]. The results show that hybrid protocols can significantly improve the efficiency of STCs.

4. SECURE HMM FORWARD ALGORITHM

We have observed scenarios and emerging use cases that make evident the need for a secure Forward algorithm that allows users and services to keep their sensitive or valuable inputs private, e.g., patient data or intellectual property. Our analysis of related work reveals different limitations which render previous approaches unsuitable for our problem scenario. Hence, to solve the open problem of secure HMM Forward computation, we now present *Priward* which achieves the desired balance of accuracy and performance, allows outsourcing, and provides adequate long-term security. *Priward* combines additive sharings and garbled circuits (cf. Sect. 3) with custom protocols to an efficient hybrid secure Forward computation protocol. In the following, we briefly introduce basic HMM theory and our notation then present the design rationale and a high-level overview of *Priward*. The required subprotocols are explained in detail afterward. Due to space constraints, the security discussions of all (sub-) protocols are given in Appendix A.

HMM Primer. HMMs are used to model stochastic processes whose internal state and corresponding transitions are hidden and only the process' output, i.e., its emissions, can be observed. An HMM is defined by $\lambda = (S, V, A, B, \pi)$ with i) hidden states $S = \{s_1, ..., s_N\}$, ii) emission alphabet $V = \{v_1, ..., v_M\}$, iii) state transition matrix $A \in \mathbb{R}^{N \times N}$ with $a_{ji} = P(s_i|s_j)$ the probability of moving from state s_j into state s_i, iv) emission matrix $B \in \mathbb{R}^{N \times M}$ with $b_i(v_j) := b_{ij} = P(v_j|s_i)$ the probability of emitting v_j in state s_i, and v) initial state distribution $\pi \in \mathbb{R}^N$ with $\pi_i = P(s_i)$ the probability of starting in state s_i. In our scenario, the HMM is held by the service $\mathcal{S}$ while $\mathcal{U}$ holds a sequence of observations made about the Markov process. An observation sequence $O = o_1...o_T \in V^{1 \times T}$ is simply a sequence of emission symbols $o_t \in V$ of length T, e.g., a sequence of utterances in a speech recognition application. One important problem is to compute how likely it is that a given HMM has generated a given observation sequence. This problem can be solved using the classical *HMM Forward algorithm* which is also an integral part of training HMMs. The Forward algorithm computes $P(O|\lambda)$, i.e., the probability of λ to have generated O, iteratively using dynamic programming:

i) *Initialization:* $\alpha_1(i) = \pi_i \cdot b_i(o_1),\ \forall i.$
ii) *Recursion:* $\alpha_t(i) = \left(\sum_{j=1}^{N} \alpha_{t-1}(j) \cdot a_{ji}\right) \cdot b_i(o_t),\ \forall t, i.$
iii) *Termination:* $P(O|\lambda) = \sum_{i=1}^{N} \alpha_T(i).$

Note that the probabilities $\alpha_t(i)$ get progressively smaller which quickly causes underflows and introduces numerical instability in the computation [19, 38]. Rabiner [38] proposes to normalize the forward variables $\alpha_t(i)$ after each iteration to deal with this problem. As an alternative, Durbin et al. [19] propose to compute and store all values in logarithmic space. We refer to probabilities in logspace as *scores*, with $\log(P(O|\lambda))$ being the *Forward score*.

Design Rationale. In our requirements analysis (Sect. 2.2), we identified performance and accuracy of a secure Forward algorithm as two main challenges. Indeed, we found no approach in related work (Sect. 2.3) that provides a satisfiable solution to these challenges. Our approach to overcome these challenges is threefold: First, to achieve reasonable accuracy, we store all involved probabilities in logarithmic representation and with fixed-point precision

Input: $\mathcal{U}$ has $O \in V^{1\times T}$, $\mathcal{S}$ has $\lambda = (S, V, A, B, \pi)$
Output: Forward score $\hat{P}(O|\lambda)$

Initialization: For $1 \leq i \leq N$

$\mathcal{U} \Leftrightarrow \mathcal{S}:$ $\hat{b}_i(o_1)^{\mathcal{U}/\mathcal{S}} \leftarrow \texttt{Emission}(o_1, \hat{B}_i)$
$\mathcal{U}, \mathcal{S}:$ $\hat{\alpha}_1(i)^{\mathcal{U}/\mathcal{S}} = \hat{b}_i(o_1)^{\mathcal{U}/\mathcal{S}} + \hat{\pi}_i^{\mathcal{U}/\mathcal{S}}$
with $\hat{\pi}_i^{\mathcal{U}} = \texttt{Logzero}$ and $\hat{\pi}_i^{\mathcal{S}} = \hat{\pi}_i$

Recursion: For $2 \leq t \leq T, 1 \leq i \leq N$

$\mathcal{U} \Leftrightarrow \mathcal{S}:$ $\hat{b}_i(o_t)^{\mathcal{U}/\mathcal{S}} \leftarrow \texttt{Emission}(o_t, \hat{B}_i)$
$\mathcal{U}, \mathcal{S}:$ $X_t^{\mathcal{U}/\mathcal{S}} = (\hat{\alpha}_{t-1}(1) + \hat{a}_{1i}, ..., \hat{\alpha}_{t-1}(N) + \hat{a}_{Ni})^{\mathcal{U}/\mathcal{S}}$
with $\hat{a}_{ji}^{\mathcal{U}} = \texttt{Logzero}$ and $\hat{a}_{ji}^{\mathcal{S}} = \hat{a}_{ji}$
$\mathcal{U} \Leftrightarrow \mathcal{S}:$ $\hat{\alpha}'_t(i)^{\mathcal{U}/\mathcal{S}} \leftarrow \texttt{Logsum}(X_t^{\mathcal{U}/\mathcal{S}})$
$\mathcal{U}, \mathcal{S}:$ $\hat{\alpha}_t(i)^{\mathcal{U}/\mathcal{S}} = \hat{\alpha}'_t(i)^{\mathcal{U}/\mathcal{S}} + \hat{b}_i(o_t)^{\mathcal{U}/\mathcal{S}}$

Termination:

$\mathcal{U}, \mathcal{S}:$ $X_T^{\mathcal{U},\mathcal{S}} = (\hat{\alpha}_T(1), ..., \hat{\alpha}_T(N))^{\mathcal{U}/\mathcal{S}}$
$\mathcal{U} \Leftrightarrow \mathcal{S}:$ $\hat{P}(O|\lambda)^{\mathcal{U}/\mathcal{S}} \leftarrow \texttt{Logsum}(X_T^{\mathcal{U}/\mathcal{S}})$
$\mathcal{U} \Leftrightarrow \mathcal{S}:$ $\hat{P}(O|\lambda) \leftarrow \texttt{Recombine}(\hat{P}(O|\lambda)^{\mathcal{U}/\mathcal{S}})$

Protocol 1: The secure `Forward` protocol: $\mathcal{U}$ holds an observation sequence O that she wants to match against the HMM λ held by $\mathcal{S}$. Using the secure `Emission` and `Logsum` primitives proposed in this work, both parties are able to compute the Forward score $\hat{P}(O|\lambda)$ securely, i.e., without either party revealing their sensitive input to the other. `Logzero` represents the special case $\log(0)$.

which allows us to avoid the use of expensive secure floating point primitives [5, 17]. Second, based on the ideas proposed in [37], we replace the critical logsum operation that caused prohibitive inaccuracies and high overheads in related works [21, 34, 35, 40] with a piece-wise linear approximation (PLA) that we compute efficiently using GCs. The granularity of the approximation directly yields a performance/accuracy trade-off. Third, we replace the expensive HE primitives of related work with additive secret sharing and oblivious transfer which is more efficient, scales to long-term security levels, and supports outsourcing computations from constrained devices to the cloud without compromising security and at almost no costs.

Notation. Our protocols use the following notation: Forward computation is carried out between the user $\mathcal{U}$ and the service $\mathcal{S}$. A single message sent by $\mathcal{U}$ to $\mathcal{S}$ is written as $\mathcal{U} \Rightarrow \mathcal{S}$, while $\mathcal{U} \Leftrightarrow \mathcal{S}$ denotes multiple messages and rounds of communication between the two parties. Values in our logspace representation (cf. Sect. 4.1) are denoted by $\hat{x} \in \mathbb{Z}$ and normal real-valued probabilities by $x \in \mathbb{R}^+$. $\texttt{Logzero} := \log(0)$ represents the special case $x = 0$. Secure subprotocols are marked in `verbatim` and are realized using OT, GCs, and additive sharings. An additive secret sharing of $x \in \mathbb{Z}_{2^l}$ is denoted by $x^{\mathcal{U}/\mathcal{S}}$ where each party holds one share. $r^{\mathcal{U}/\mathcal{S}} = (expr(x_1, ..., x_n))^{\mathcal{U}/\mathcal{S}}$ denotes the evaluation of an arithmetic expression on the variables x_i of which at least one is additively shared such that the result r is shared between them as well. $\mathcal{GC}_C(\tilde{x_1}, .., \tilde{x_n}, \mathcal{P})$ denotes the secure evaluation of a Boolean circuit C on *garbled* inputs $\tilde{x}_1, .., \tilde{x}_n$ and a set $\mathcal{P}$ of cleartext inputs using Yao's GC protocol [42]. All computations on additive sharings and within GCs are carried out in $\mathbb{Z}_{2^l}$ if not stated otherwise.

Overview. *Priward* computes exactly the same initialization, recursion, and termination steps as the classical Forward algorithm. However, to enable both parties to keep their inputs private, we substitute those steps that require both parties' inputs with secure interactive protocols carried out by $\mathcal{U}$ and $\mathcal{S}$. On the highest level, we only need two secure protocols, `Emission` and `Logsum`, to build the secure Forward. `Emission` takes an observation held by $\mathcal{U}$ and the emission score matrix held by $\mathcal{S}$ and selects the corresponding emission score and shares it additively between the two parties without either party learning the other's input. `Logsum` computes the sum of two logspace values that are given as additive shares and returns the result again in shared form such that neither party is able to learn anything from the shares. Intuitively, having subprotocols operate over additive shares allows us to easily compose them in a secure manner. Protocol 1 shows the details of our secure Forward algorithm realized with these two primitives. In the following, we explain how $\mathcal{U}$ and $\mathcal{S}$ compute each of the three phases (initialization, recursion, and termination) in more detail. We defer discussions on outsourcing to Sect. 4.4.

Initialization: At the start, $\mathcal{U}$ holds an observation sequence $O = o_1...o_T$ and $\mathcal{S}$ holds the HMM λ. The goal of initialization is to compute additive shares of $\hat{\alpha}_1(i) = \hat{\pi}_i + \hat{b}_i(o_1)$ for $i = 1 \ldots N$. To this end, $\mathcal{U}$ and $\mathcal{S}$ invoke our `Emission` primitive N times after which both parties obtain additive shares of the emission scores $\hat{b}_{i=1..N}(o_1)^{\mathcal{U}/\mathcal{S}}$. $\mathcal{S}$ then adds the prior state scores $\hat{\pi}_i$ and $\mathcal{U}$ adds `Logzero` to compute the desired sharing $\hat{\alpha}_1(i)^{\mathcal{U}/\mathcal{S}}$. We base `Emission` (cf. Sect. 4.2) on OT, as this is significantly more efficient than HE-based constructions suggested in related work.

Recursion: The goal of the recursion step is to compute additive shares of the forward variables $\hat{\alpha}_t(i)^{\mathcal{U}/\mathcal{S}}$ for $i = 1 \ldots N$ given the additive shares $\hat{\alpha}_{t-1}(j)^{\mathcal{U}/\mathcal{S}}$ from the previous iteration. As before, $\mathcal{U}$ and $\mathcal{S}$ first invoke `Emission` to additively share the emission scores $\hat{b}_i(o_t)^{\mathcal{U}/\mathcal{S}}$. Next, they use our secure `Logsum` primitive to compute additive shares of $\hat{\alpha}'_t(i) = \log(\sum_{j=1}^{N} \alpha_{t-1}(j) \cdot a_{ji})$. Unlike related work [34, 35, 40], we avoid to work on the actual probabilities $\alpha_{t-1}(j)$ and a_{ji} because this is liable to cause critical inaccuracies and overheads. Instead, we compute a piece-wise linear approximation in logspace as explained in detail in Section 4.3. The result of `Logsum` is again distributed as additive shares between $\mathcal{U}$ and $\mathcal{S}$ and they only need to locally add their shares of the emission scores $\hat{b}_i(o_t)^{\mathcal{U}/\mathcal{S}}$ to obtain the desired additive sharing $\hat{\alpha}_t(i)^{\mathcal{U}/\mathcal{S}}$.

Termination: Finally, in the termination step the forward variables $\alpha_T(i)$ are summed up. Since we compute in logspace and over additive shares, we need to employ the `Logsum` primitive again. As the final result, the two parties $\mathcal{U}$ and $\mathcal{S}$ each hold additive shares $\hat{P}(O|\lambda)^{\mathcal{U}/\mathcal{S}}$ of the Forward score. Depending on who should learn the result in the concrete use case, the parties exchange their shares to enable reconstruction of the Forward score $\hat{P}(O|\lambda)$.

In the following, we explain our number representation (Sect. 4.1), our secure `Emission` (Sect. 4.2) as well as `Logsum` primitive (Sect. 4.3), and how to outsource computations (Sect. 4.4).

4.1 Number Representation

One of the main challenges for (secure) Forward computation are the extremely small probabilities involved. To this end, Aliasgari et al. [4] argue that full floating point precision is required. Indeed, the numerical instabilities encountered in the approach by Polat et al. [36] (cf. Sect. 2.3) underline that the dynamic range of the occurring probabilities is indeed too large to compute with *fixed-point precision in probability space*. Unfortunately, even recent highly optimized secure floating point primitives still incur high overheads [4, 17] and are still too expensive for secure computations on HMMs. Hence, we decide to follow the alternative approach proposed by Durbin et al. [19] and carry out all computations *in logspace using a fixed-point representation* of any involved non-integers. As the results presented in [21, 35] indicate, this approach achieves sufficiently accurate results for real-world use cases.

Formally, we transform a probability $p \in (0, 1] \subset \mathbb{R}$ to a fixed-point logspace representation $\hat{p}' = \lfloor 2^s \cdot \log(p) \rceil$ ($\lfloor \cdot \rceil$ denoting the nearest integer) and map $\hat{p}'$ to the ring $\mathbb{Z}_{2^l}$ by computing $\hat{p}' \mod 2^l$, where l is the chosen bitlength and typically $l \in \{32, 64\}$.

Input: Additive sharing $x^{\mathcal{U}/\mathcal{S}}$ of $x \in \mathbb{Z}_{2^l}$
Output: Additive sharing $x'^{\mathcal{U}/\mathcal{S}}$ of $x' = \lfloor x/2^s \rceil \in \mathbb{Z}_{2^{l-s}}$

$$\begin{aligned}
\mathcal{U} \Rightarrow \mathcal{S}: &\quad x_r^{\mathcal{U}} = x^{\mathcal{U}} + r, \;\; r \in_R \mathbb{Z}_{2^{l+\kappa}} \\
\mathcal{U}: &\quad x'^{\mathcal{U}} = -(r \gg s) \mod 2^{l-s} \\
\mathcal{S}: &\quad x_r = x^{\mathcal{S}} + x_r^{\mathcal{U}} \\
&\quad x'^{\mathcal{S}} = x_r \gg s \mod 2^{l-s}
\end{aligned}$$

Protocol 2: The `Rescale` primitive adapted to additive sharings from [15]: $\mathcal{U}$ and $\mathcal{S}$ hold additive shares of $x \in \mathbb{Z}_{2^l}$. By recombining a blinded x_r they are able to securely compute additive shares of $\lfloor x/2^s \rceil \in \mathbb{Z}_{2^{l-s}}$. Rescaling is required after each multiplication of two fixed-point values represented as integers scaled by 2^s.

Note that scaling and rounding to integers is required due to our cryptographic building blocks. We handle the case $p = 0$ by the special symbol `Logzero` which is practically represented by a sufficiently small number, e.g., -2^{l-1}. After transforming the inputs, any intermediate values and results are also expressed in fixed-point logspace representation. Hence, we must ensure that no value exceeds the bit-length l to avoid incorrectness due to over- and underflows. Note that the sum of two scaled values is again an integer that is scaled by the same factor. However, multiplication leads to an accumulation of the scaling factors, i.e., the product is scaled by 2^{2s}, which would quickly exceed the maximum bit-length l and hence cause errors. To avoid this, we scale the product down by the scaling factor 2^s before any subsequent additions or multiplications are performed.

On plaintext values, this rescaling is a simple matter of division and rounding. However, in our approach, all values are additively shared in $\mathbb{Z}_{2^l}$ which prevents straightforward division and we cannot recombine them for rescaling without violating our security requirements. Instead, we propose an efficient and secure protocol, `Rescale` (Protocol 2). We adapt this protocol from [15] and extend it to work over additive shares in the two-party setting. Our adapted protocol proceeds as follows. Note that all operations are performed in $\mathbb{Z}$, i.e., without modular arithmetic. Initially, $\mathcal{U}$ and $\mathcal{S}$ hold shares $x^{\mathcal{U}}$ respectively $x^{\mathcal{S}}$ of an intermediate value $x \in \mathbb{Z}_{2^l}$ which is scaled by 2^{2s}. First, $\mathcal{U}$ blinds her share $x^{\mathcal{U}}$ using a random number r of length $l + \kappa$ bit and sends it to $\mathcal{S}$. Then, $\mathcal{U}$ truncates the lower s bits of r (a right shift by s bit scales down by 2^s), and uses the negative result as her share $x'^{\mathcal{U}}$. $\mathcal{S}$ obtains the blinded input $x_r = x^{\mathcal{S}} + x_r^{\mathcal{U}}$, similarly truncates the lower s bits of x_r and uses the result as its share $x'^{\mathcal{S}}$. The resulting values $x'^{\mathcal{U}}, x'^{\mathcal{S}}$ share the desired downscaled value x' in $\mathbb{Z}_{2^{l-s}}$. Note that `Rescale` introduces a random error in the least significant bit of the rescaled value, i.e., $x' = \lfloor x/2^s \rfloor + u$ with $u \in_R \{0, 1\}$. It is, however, significantly more efficient than the deterministic pendant proposed in [41] and our experiments show that this error can be tolerated.

4.2 Secure Emission Primitive

The `Emission` primitive (Protocol 3) is used during initialization and at the beginning of each recursion step. It obliviously selects the required emission score $\hat{b}_i(o_t)$ and then shares it additively between both parties so that they can compute securely with it. `Emission` proceeds in the following steps: At the start, the user $\mathcal{U}$ holds the observation o_t and the service $\mathcal{S}$ inputs the i^{th} row of the HMM's emission matrix $\hat{B}$. To hide the real values, $\mathcal{S}$ first blinds the entire row $\hat{B}_i$ by adding the same random value $r_{\mathcal{S}}$ to each emission score $\hat{b}_i(v_j) \in \hat{B}_i$. Both parties then engage in 1-M-OT$_l$ after which $\mathcal{U}$ obtains the blinded emission score $\hat{b}'_i(o_t)$ corresponding to her observation o_t. The use of OT guarantees that $\mathcal{U}$ learns only $\hat{b}'_i(o_t)$ and that $\mathcal{S}$ learns nothing about o_t. Note that $\hat{b}_i(o_t)^{\mathcal{U}/\mathcal{S}} = (\hat{b}'_i(o_t), -r_{\mathcal{S}})$ already is the desired additive sharing of the emission score $\hat{b}_i(o_t)$.

Input: $\mathcal{U}$ has $o_t \in V$, $\mathcal{S}$ has $\hat{B}_i = (\hat{b}_i(v_1), ..., \hat{b}_i(v_M))$
Output: Additive sharing $\hat{b}_i(o_t)^{\mathcal{U}/\mathcal{S}}$ of $\hat{b}_i(o_t)$

$$\begin{aligned}
\mathcal{S}: &\quad \hat{B}'_i = (b_i(v_1) + r_{\mathcal{S}}, ..., b_i(v_M) + r_{\mathcal{S}}) \\
&\quad \text{with } r_{\mathcal{S}} \in_R \mathbb{Z}_{2^l} \\
\mathcal{U} \Leftrightarrow \mathcal{S}: &\quad \hat{b}_i(o_t)^{\mathcal{U}} \leftarrow \text{1-M-OT}_l(o_t, \hat{B}'_i) \\
\mathcal{S}: &\quad \hat{b}_i(o_t)^{\mathcal{S}} = -r_{\mathcal{S}}
\end{aligned}$$

Protocol 3: The `Emission` primitive: $\mathcal{U}$ holds an observation o_t and $\mathcal{S}$ holds the emission scores $\hat{B} \in \mathbb{R}^{N \times M}$. $\mathcal{U}$ and $\mathcal{S}$ securely compute additive shares of $\hat{b}_i(o_t)$ using additive blindings and a single invocation of 1-M-OT$_l$.

During the Forward algorithm, `Emission` is invoked once for each state $s_i \in S$ in each time step $1 \leq t \leq T$, i.e., a total of $N \cdot T$ times. Since for time step t the choice o_t is the same for all $s_i \in S$, we reduce the N calls to 1-M-OT$_l$ per time step t to one 1-M-OT$_{Nl}$ which is more efficient. Additionally, we batch all remaining T calls to `Emission` together to further improve efficiency resulting in one call to 1-M-OT$^T_{Nl}$.

4.3 Secure Logsum Primitive

In the recursion step of the logspace Forward, we need to compute the logarithm over a sum, i.e., $\hat{z} = \log(\sum_{j=1}^{N} \alpha_{t-1}(j) a_{ji})$, where we know the summands $\alpha_{t-1}(j) a_{ji}$ only as values in logspace, i.e., $\hat{\alpha}_{t-1}(j) + \hat{a}_{ji}$. This operation is referred to as *logsum* and is ubiquitous not only in HMM computation but in signal processing and pattern classification in general [37]. A logsum over N logspace values is usually reduced to $N - 1$ successive or tree-wise calls to

$$\text{logsum}(\hat{x}, \hat{y}) = \hat{x} + \log(1 + \exp(\hat{y} - \hat{x})) \tag{1}$$

with $\hat{x} \geq \hat{y}$ (w.l.o.g.) where the term $\log(1 + \exp(\hat{y} - \hat{x}))$ is either computed directly or looked up in a precomputed table.

To compute the Forward algorithm securely, we need a secure `Logsum` primitive which computes Equation 1 on shared inputs $\hat{x}^{\mathcal{U}/\mathcal{S}}, \hat{y}^{\mathcal{U}/\mathcal{S}}$ and returns the results as additive shares, denoted by

$$\hat{z}^{\mathcal{U}/\mathcal{S}} \leftarrow \texttt{Logsum}(\hat{x}^{\mathcal{U}/\mathcal{S}}, \hat{y}^{\mathcal{U}/\mathcal{S}}) \tag{2}$$

We briefly discuss approaches in related work to securely computing logsums and then explain our approach in detail. Equation 1 could be computed using the secure floating point primitives from [5,17] or using homomorphic encryption and fixed-point precision with rescaling as proposed in [35,40]. Franz et al. [21] compute Equation 2 based on HE and oblivious lookup tables which grow exponentially in the bit-lengths of the inputs. We deem these approaches too expensive for our use case and follow the alternative idea of Portelo et al. [37] to compute a *piecewise linear approximation* (PLA) of Equation 2. While Portelo et al. [37] propose a completely GC-based solution, we propose a hybrid solution that efficiently combines GCs with additive sharings and achieves better performance than previous secure logsum computations.

The details of `Logsum` are given in Protocol 4. In a precomputation step (that can happen at any time and needs to be computed only once), $\mathcal{S}$ computes the parameters for the PLA: $\mathcal{S}$ selects k intervals $[l_i, r_i]_{1 \leq i \leq k}$ and computes a linear regression $m_i x + n_i$ of $\log(1 + \exp(-x))$ with $x \in [l_i, r_i]$. In the first protocol step, $\mathcal{U}$ and $\mathcal{S}$ convert their additively shared inputs into garbled inputs by evaluating a garbled addition circuit [18]. Both parties then evaluate the first part of Portelo's circuit which obliviously computes $\max(\hat{x}, \hat{y})$ and $d = |\hat{x} - \hat{y}|$ and then obliviously selects parameters $(l_i, r_i, m_i, n_i) \in \mathcal{P}$ where $l_i \leq d < r_i$. Different to [37], we now convert back to additive shares using the OT-based subtraction protocol proposed in [18]. The arithmetic representation then allows us to compute the final result $m \cdot d + n$ over additive shares much more efficiently than using GCs as proposed by [37].

Input: Two shared summands $\hat{x}^{\mathcal{U}/\mathcal{S}}, \hat{y}^{\mathcal{U}/\mathcal{S}}$, PLA parameter $k \in \mathbb{N}$
Output: Additive sharing $\hat{z}^{\mathcal{U}/\mathcal{S}}$ of $\hat{z} = \log(x + y)$

$\mathcal{S}$: Compute PLA $\mathcal{P} = \{(l_i, r_i, m_i, n_i)_{1 \le i \le k}\}$
$\mathcal{U} \Leftrightarrow \mathcal{S}$: $\tilde{x}, \tilde{y} \leftarrow \mathcal{GC}_{C_{Add}}(\hat{x}^{\mathcal{U}/\mathcal{S}}, \hat{y}^{\mathcal{U}/\mathcal{S}})$
$\mathcal{U} \Leftrightarrow \mathcal{S}$: $(\tilde{max}, \tilde{d}, \tilde{m}, \tilde{n}) \leftarrow \mathcal{GC}_{C_{Select}}(\tilde{x}, \tilde{y}, \mathcal{P})$
$\mathcal{U} \Leftrightarrow \mathcal{S}$: $(max, d, m, n)^{\mathcal{U}/\mathcal{S}} \leftarrow \mathcal{GC}_{Sub}(\tilde{max}, \tilde{d}, \tilde{m}, \tilde{n})$
$\mathcal{U} \Leftrightarrow \mathcal{S}$: $md^{\mathcal{U}/\mathcal{S}} \leftarrow \texttt{Rescale}((m \cdot d)^{\mathcal{U}/\mathcal{S}})$
$\mathcal{U}, \mathcal{S}$: $\hat{z}^{\mathcal{U}/\mathcal{S}} = (max + md + n)^{\mathcal{U}/\mathcal{S}}$

Protocol 4: The `Logsum` primitive: We adapt the idea from [37] to compute the logsum of two logspace values by a piecewise linear approximation. To improve efficiency, only the selection of the approximation parameters is done using GCs while arithmetic operations are performed over additive shares.

4.4 Outsourcing

We consider a scenario where the two parties $\mathcal{U}$ and $\mathcal{S}$ need to outsource computation of the Forward algorithm to two other peers $P_{\mathcal{U}}$ and $P_{\mathcal{S}}$ of their choice, e.g., due to processing or bandwidth constraints. This could be the case, e.g., for a mobile user $\mathcal{U}$ that communicates to a cloud service which has become a ubiquitous communication pattern, today. Though this example would require only one party to outsource computations, we show how *both* parties outsource computations for the sake of generality. A scenario where only one party outsources is then straightforward even simpler to realize. Note that the outsourcing step itself must be inexpensive such that it respects $\mathcal{U}'s$ and $\mathcal{S}'s$ resource constraints. However, outsourcing must not break security, i.e., $P_{\mathcal{U}}$ and $P_{\mathcal{S}}$ must remain oblivious of the inputs and outcome of the computation.

To outsource computations, $\mathcal{U}$ and $\mathcal{S}$ first need to execute our `Emission` primitive to compute $\hat{b}_i(o_t)\ \forall t, i$. `Emission` is the only part that cannot be outsourced because it requires the executing parties to know the observation sequence O and the model λ in clear. However, our evaluation (Sect. 5) shows that `Emission` is very efficient and can thus be computed even by resource constrained devices. $\mathcal{U}$ now sends her shares $\hat{b}_i(o_t)^{\mathcal{U}}$ to $P_{\mathcal{U}}$ and $\mathcal{S}$ sends $\hat{b}_i(o_t)^{\mathcal{S}}$ to $P_{\mathcal{S}}$. Additionally, $\mathcal{S}$ shares all transition scores $\hat{a}_{ij}, 1 \le i, j \le N$ and the prior state distribution $\hat{\pi}_i,\ 1 \le i \le N$ to $P_{\mathcal{U}}$ and $P_{\mathcal{S}}$ by invoking `Share`$(\cdot)$ on each item, individually. $P_{\mathcal{U}}$ and $P_{\mathcal{S}}$ then compute `Forward` (Protocol 1) on the given shares. Finally, $P_{\mathcal{U}}$ and $P_{\mathcal{S}}$ send their share of the final result to $\mathcal{U}$ and $\mathcal{S}$, respectively, who only have to compute one local addition to reconstruct $\hat{P}(O|\lambda)$. Note that $P_{\mathcal{U}}$ and $P_{\mathcal{S}}$ must not collude which is the standard assumption in the secure two-party setting and clearly reasonable in our scenario since $P_{\mathcal{U}}$ and $P_{\mathcal{S}}$ are chosen individually by $\mathcal{U}$ and $\mathcal{S}$.

5. EVALUATION

To thoroughly quantify and evaluate the performance and accuracy of our approach, we implemented a prototype of *Priward*. Note that security of *Priward* is discussed in Appendix A. We first perform synthetic benchmarks on fully connected HMMs to derive a thorough understanding of the performance and accuracy of *Priward* (Sect. 5.1). We then show the applicability of our approach in a real-world bioinformatics use case by integrating *Priward* into the well-established HMMER framework [1] (Sect. 5.2). In contrast to generic fully connected HMMs, the special architecture of the HMMs involved in this use case allows significant performance optimizations. Finally, we qualitatively compare the performance of our approach against related works (Sect. 5.3). Now, we provide further details on our implementation and experimental setup.

Implementation. We implement the *Priward* prototype in C++. `Emission` requires the 1-n-OT$_l^m$ primitive which we implement as one invocation of 1-2-OT$_l^{m \log_2(n)}$ according to [32, 33]. For 1-2-OT$_l^m$ we employ the efficient OT Extensions described in [6]. `Logsum` requires building and evaluating GCs, for which we build upon the ABY two-party computation framework [18]. Besides the OT Extension and ABY framework, which are fully multithreaded, the rest of our implementation realizes only obvious optimizations, e.g., batching of the operations in the inner *for* loop of the Forward algorithm. Hence, further optimizations are possible, e.g., pipelining GC generation and evaluation as proposed in [26].

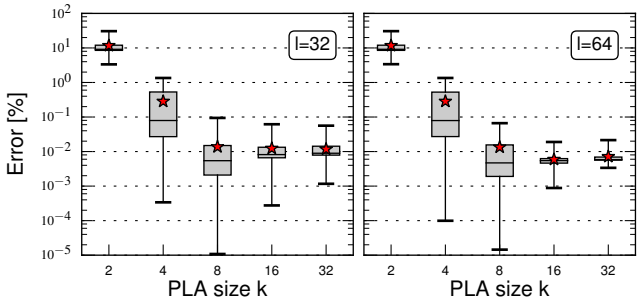

Figure 2: Relative error of `Forward` on matching and non-matching sequences: As we increase the number of approximation intervals k (x-axis) the accuracy increases. Increasing fixed-point precision from $l = 32$ (left) to $l = 64$ (right) shows less improvement but reduces the variance of the error.

Experimental Setup. We perform all experiments between two standard desktop machines (Ubuntu 14.04 LTS, Intel i7-4770 @ 3.10 GHz, 16 GB RAM) that communicate over a Gigabit LAN. To offer long-term security according to NIST [8], we set the statistical security parameter to $\kappa = 80$ bits and the symmetric security level to $t = 128$ bits.

5.1 Generic HMMs

For the synthetic benchmarks, we create a dataset of ergodic (fully connected) HMMs and observation sequences to serve as inputs. Ergodic HMMs are the most general cases and also the most challenging in terms of performance and accuracy, as they require to take all other states into account as predecessors during the recursion steps of the Forward algorithm. Our dataset consists of random and circular HMMs as well as matching and non-matching observation sequences. Random HMMs and non-matching sequences are sampled completely at random, while for the circular HMMs, we sample state transitions and emission probabilities from a Gaussian distribution centered on the topologically next state leading to a roughly *circular* structure. With the intention to create good matches, we sample observation sequences for the circular HMMs as a noisy linear walk through the states of the HMM. Both sets of HMMs (random and circular) contain HMMs with a varying number of states $N = 10, 20, ..., 100$, emission alphabet sizes $M = 10^2, ..., 10^4$, and number of observations $T = 10, 20, ..., 100$. Our selection of parameters covers a wide range of actual use cases and choices made in related work [4, 34, 35, 38].

5.1.1 Accuracy

Our approach introduces numerical inaccuracies at two points, i) through the fixed-point representation of probabilities and probabilistic rescaling (cf. Sect. 4.1), and ii) through the approximation of logsum operations (cf. Sect. 4.3). In the following, we compare the results of our secure `Forward` against a reference implementation on plaintexts to show that our approach nevertheless computes accurate results. Our reference for accurate results is the widely-

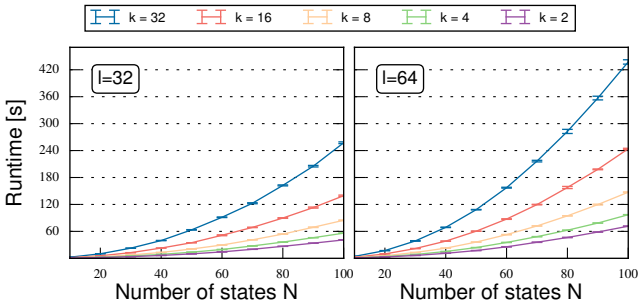

Figure 3: Runtime of `Forward` on HMMs with a different number of states N (x-axis) for different k (lines) and l (left and right plot): The runtime grows quadratically in N and linearly in k and l.

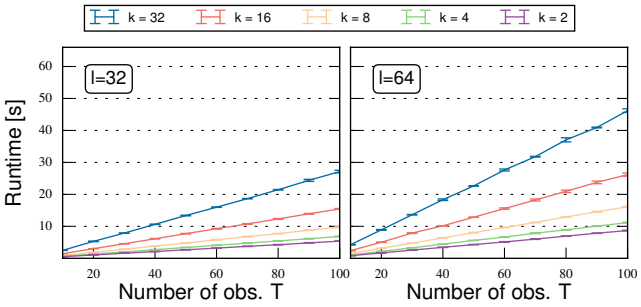

Figure 4: Runtime of `Forward` for HMMs with a different number of observations T (x-axis) for different choices of k (lines) and l (left and right plot): The runtime grows linearly in T, k and l.

used natural-language toolkit (NLTK) [2] which is implemented in double precision in Python.

Figure 2 plots the errors our approach introduces relative to the reference results obtained with NLTK. On the x-axis, we vary the number of approximation intervals $k = 2, 4, ..., 32$ in the piecewise linear approximation; left and right plots vary the bit-length $l = 32, 64$ used for the fixed-point logspace number representation. We show the second and third quartiles (boxes), the and max (whiskers) as well as the means (stars). As expected, the error decreases as we increase the accuracy of the PLA by increasing the number of approximation intervals k. While an approximation with only $k = 2$ intervals causes significant errors, the error decreases quickly for bigger values of k. Already, $k = 4$ achieves a mean error below 0.3 % for both $l = 32, 64$. For $k = 8$ even the maximum error (indicated by the whiskers) drops below 0.1 %. We observe that the average accuracy does not significantly improve beyond $k = 8$. This is due to the numerical errors introduced by the fixed-point number representation and rescaling protocol (cf. Sect. 4.1). Interestingly, using a higher bit-length of $l = 64$ shows only marginal improvements to the average accuracy. However, the variance of the error is reduced and outliers are less extreme.

In conclusion, our results confirm the expected: more approximation intervals k and higher fixed-point precision l improve the numerical accuracy of the results. Unfortunately, an increase in either parameter will also increase the size of the involved GCs which will incur a noticeable decrease in performance. We will put these numbers into perspective when discussing a real-world use case in Sect. 5.2. The performance results presented in the next subsection will allow us to strike a reasonable trade-off between accuracy and performance.

5.1.2 *Runtime and Communication*

We first analytically derive the critical parameters for *Priward*'s runtime and then thoroughly evaluate its performance. The runtime complexity $\mathcal{O}(TN^2)$ of the Forward algorithm is quadratic in the number of states N and linear in the number of observations T. In the secure `Forward` protocol, we additionally need to sahre all emission scores $\hat{b}_i(o_t)$ securely using `Emission` which scales in $\mathcal{O}(TMN)$ with M the size of the emission alphabet. Thus, N, T, and M are the critical parameters for which we have to analyze the runtime. All results are aggregated over 20 independent runs and plots show the mean values with corresponding standard deviation.

Number of States. Figure 3 plots the runtime of `Forward` on HMMs with $N = 10, 20, ..., 100$ states for different PLA sizes $k = 2, 4, ..., 32$ and bit-lengths $l = 32, 64$ with fixed $M = 1000$ and $T = 10$. As indicated by the overall complexity of the Forward algorithm, the runtime increases quadratically in the number of states. Qualitatively, we observe the same growth for the communication overhead. Still, our approach is efficient and can evaluate a fully connected HMM with $N = 100$ states with a reasonable error smaller 1 % ($k = 4, l = 32$) in less than one minute requiring 1.59 GB for communication. As expected in the context of secure computations, the communication overhead is significant and may overtax especially mobile users. However, GCs are an active research area and all ongoing and future optimizations of garbling schemes [10,43] will also benefit our approach and reduce overheads further. Meanwhile, *Priward* can be outsourced to a computation cloud which can easily manage current communication overheads.

Increasing parameters k or l increases runtime and communication linearly. This is due to the fact that the runtime is dominated by the $(T-1)N(N-1)+(N-1)$ calls of the `Logsum` primitive whose core is a GC of size roughly linear in k and l. In total, the `Logsum` calls account for more than 99 % of the overall runtime and for more than 95 % of the total communication. The rest of the computational overhead is due to the `Emission` primitive, while the overhead of the operations on additive sharings are negligible. Since efficiently calculating secure `Logsum` is relevant beyond the scope of this work, e.g., for the secure computation of Gaussian Mixture Models [4, 35], we provide a more elaborate evaluation of `Logsum` in Appendix B.

Length of the Observation Sequence. Figure 4 plots the runtime of `Forward` for $T = 10, 20, ..., 100$ observations and fixed $N = 10$ and $M = 1000$. Runtime and communication scale linearly in T as the number of required `Logsum` operations scales linearly in T. For example, processing $T = 10$ observations costs 0.65 s and 15.45 MB as opposed to 6.95 s and 167.05 MB for $T = 100$ observations. For the same reason as before, runtime and communication also scale roughly linearly in k and l.

Size of the Emission Alphabet. The alphabet size M only influences the performance of `Emission` which accounts for less than 1 % of the overall overhead of `Forward`. To provide greater detail, Figure 5 plots the runtime of the isolated `Emission` primitive. We choose different $T = 10, 20, ..., 100$ and $M = 10^2, 10^3, 10^4$ while fixing $N = 10$. Clearly, the runtime of `Emission` is linear in T which conforms with its complexity $\mathcal{O}(TMN)$. The communication overhead shows qualitatively the same growth. We increase M exponentially to show that, even for huge emission alphabets, `Emission` is very efficient. For example, less than 0.5 s and about 40.13 MB are required to securely share $T = 100$ emission scores over a huge alphabet of 10000 possible emissions. The ability to efficiently handle huge emission alphabets makes our `Emission` primitive a candidate for computing fast approximations of the emission scores for HMMs with continuous probability distributions such as Gaussian Mixture Models used in speech processing [35, 38].

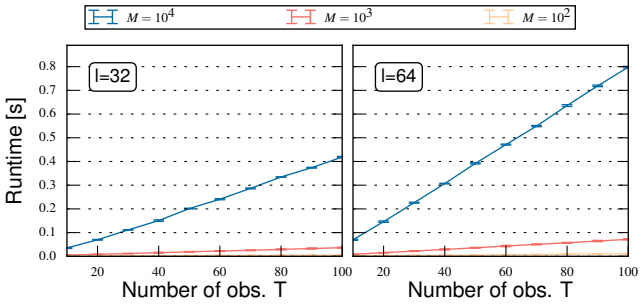

Figure 5: Runtime of `Emission` for a varying number of observations T (x-axis), alphabet sizes M (lines) with $l = 32$ (left) and $l = 64$ (right): Runtime grows linearly in T and M and causes only 1 % of the total runtime overhead of `Forward` (cf. Figs. 3 and 4).

Outsourcing. We briefly discuss the overheads imposed on either party $\mathcal{U}$ and $\mathcal{S}$ for outsourcing their computation to another peer. As explained in Section 4.4, the overheads consist in executing the batched `Emission` primitive as well as sending the resulting shares to the computation peers. The overheads for `Emission` are very low even for a large number of observations and huge emission alphabets, e.g., less than 0.5 s for 100 emissions out of an alphabet of 10000 possible observations (cf. Fig. 5). Distributing shares to the computation peers only requires tens of KB even for the larger considered models. The overhead is smaller for $\mathcal{U}$ than for $\mathcal{S}$ which has to share the whole HMM. This is desirable as in many cases only $\mathcal{U}$ will need to outsource computations (cf. Sect. 4.4). Hence, outsourcing is clearly feasible for most mobile devices nowadays.

5.2 Use Case: Secure Bioinformatics Services

Recent advances have made whole genome sequencing (WGS) fast, accurate, and affordable for the masses. It is widely expected that WGS will pave the way for innovative research and novel applications [16, 44]. As we can already observe, an industry will emerge that offers genomics-based services such as drug testing or diagnosis of diseases based on proprietary research [44]. To remain competitive, service providers will need to protect the mathematical models upon which their businesses are built. On the other hand, users of such services are required to contribute genomic data which is most sensitive information [7]. The approach presented in this paper allows preserving the service provider's intellectual property while offering strongest protection for users' genomic data.

To present a concrete use case, we consider the following genetic disease testing scenario: The service provider holds a set of HMMs that model specific diseases. The user holds an observation sequence, e.g., parts of her sequenced genome, that she wants to test against the service's database. Concretely, we use HMMs from the Pfam database [3], which contains 16 295 protein families that relate to, e.g., certain phenotypes and diseases. HMMER [1] is widely used tool in bioinformatics to query Pfam. Thus, we implemented *Priward* in the most recent version, HMMER 3.1. It is important to note that HMMER and Pfam are based on *profile HMMs*. Profile HMMs have a special architecture that allows only a certain subset of transitions which significantly speeds up the Forward computation. Since we adapt these optimizations which are specific to the HMMER framework and the profile HMM architecture, the results presented in the remainder of this section are not comparable to those presented in Sect. 5.1 which were obtained on fully connected HMMs. However, it becomes clear that our approach is flexible enough to capitalize on the optimization potential offered by certain reduced HMM architectures such as profile HMMs.

From Pfam [3], we choose the same models as [21], i.e., SH3_1 (Length $L = 48$), Ras ($L = 162$), BID ($L = 191$), and added one of the smallest model found in Pfam, Extensin_1 ($L = 10$), another midsize model, Ribosomal_S3_C ($L = 83$) as well as two of the largest models, IDO ($L = 408$) and 3HBOH ($L = 689$). The average length[1] of HMMs in Pfam is 175 and more than 98.5 % of the HMMs have a length smaller than 3HBOH, the largest model we consider. We use observation sequences of the same length as the HMM length ($T = L$) of two types: i) *matching* sequences where we use the seeds on which the respective models were trained and ii) *non-matching* sequences which we choose randomly from the seeds of other models. The considered profile HMMs are built over the amino acids alphabet which has $M = 20$ symbols.

5.2.1 Accuracy

Figure 6 plots the relative error *Priward* introduces in comparison to the real scores computed by the HMMER framework on plaintexts. We restricted the evaluation to $k = 2, 4, 8$ and $l = 32$ since these choices achieve the best trade-off between accuracy and performance according to the results presented in Sect. 5.1. Clearly, $k = 2$ leads to large errors that grow roughly linearly with the combined length of the model and observation sequence. For $k = 4, 8$ the error mostly drops below 1 % and now seems mostly model specific with little correlation to the length of model and observation sequence. Considering the use case, the more important question is whether our approach is accurate enough to distinguish matching from non-matching sequences. To answer this question, we classify sequences according to the *noise cutoffs* (NC) and *trusted cutoffs* (TC) specified for each model in the Pfam database: Anything below the NC can safely be considered a non-matching sequence and anything above the TC a match. For $k = 4, 8$, our approach is able to perfectly distinguish between matching and non-matching sequences. Notably, even for $k = 2$ our classification is perfectly accurate for all but the largest model, 3HBOH.

5.2.2 Runtime and Communication

Figure 7 plots the performance overhead for the chosen Pfam models. The x-axis denotes both length L of the model[1] and length T of the observation sequence. As before, the runtime is dominated by the overhead for `Logsum`. Since the Forward algorithm over profile HMMs as implemented in HMMER 3.1 requires $T(7L + 2)$ `Logsum` operations, the runtime grows linear in both T and L. Note that we increase both L and T in Figure 7, thus the growth is quadratic. While, the smaller models can be computed in the order of seconds (e.g., 0.82 s for Extensin_1, 13.43 s for SH3_1, and 37.46 s for Ribosomal_S3_C for $k = 2$), the larger models range in the order of minutes (e.g., 2.28 min for Ras, 3.15 min for BID, and 13.75 min for IDO, 38.5 min for 3HBOH for $k = 2$). Although runtimes in the latter cases are not unreasonable, they emphasize the necessity and benefit for mobile users to be able to securely outsource computations as offered by *Priward*.

Similar to the runtime, the communication overhead is dominated by the calls to `Logsum` and thus grows quadratically as well, e.g., from 10.68 MB for the smallest model (Extensin_1) to 47.19 GB for the largest considered model (3HBOH) with $k = 2$ and $l = 32$. Clearly, the communication overhead is significant and may result in additional runtime overheads when our approach is deployed in networks with less bandwidth or higher latency than assumed in our evaluation setup. However, these overheads are clearly manageable by outsourcing *Priward* to a computation cloud.

[1] L determines the number of *nodes* in a profile HMM and each node has three distinct states. Together with four special states, a profile HMM thus has a total of $N = 3L + 4$ states.

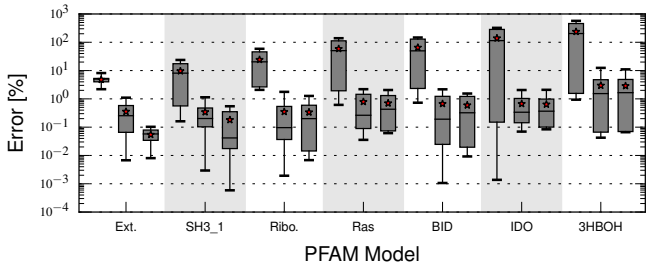

Figure 6: Relative error of `Forward` on different profile HMMs from Pfam [3] for $k = 2, 4, 8$: $k = 2$ (left bar per model) causes errors that increase linearly. For $k = 4, 8$ (middle and right bars) the accuracy increases significantly and classification of matching and non-matching sequences is 100 % correct.

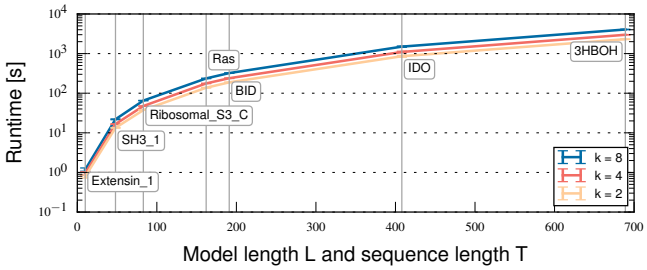

Figure 7: Runtime of `Forward` on profile HMMs from Pfam [3] with different length L on observation sequences of length $T = L$ (x-axis): Short models can be computed in the order of seconds while medium to large models are in the order of minutes. Growth is quadratic since T increases equally with L.

5.3 Comparison to Related Work

In Section 2.3, we have qualitatively discussed other approaches to secure Forward computation and summarized them in Table 1. In this section, we compare the performance of our approach quantitatively to these works by the reported performance numbers.

Pathak et al. [34,35] report a runtime of 784.58 s for the evaluation of one HMM with $N = 5$ states on an observation sequence of length $T = 98$ on a 3.2 GHz CPU. For a fair comparison, we restricted *Priward* to run on one core at 3.1 GHz and set $k = 4$ and symmetric security to $t = 80$ bits which achieves comparable accuracy and security as in [34, 35]. In this setting, we measure the runtime of *Priward* at 6.21 s which is 126x faster.

Franz et al. [21] do not evaluate their approach on generic fully connected HMMs but concentrate on profile HMMs as in our use case (cf. Sect. 5.2). Unfortunately, we could not obtain their source code and a direct comparison with the results presented in Section 5.2 is unreasonable for different reasons: First, the Forward algorithm in HMMER 3.1 requires to compute an additional $T \cdot L + 1$ logsum operations compared to version 2.3.2 used in [21]. Switching to HMMER 2.3.2 would reduce the overheads of our approach by ~ 14 %. Second, Franz et al. do not implement any networking, yet we observe networking to cause non-negligible overheads even on the local loopback interface. Third, Franz et al. use legacy/short-term security while we use long-term security which causes additional overheads in comparison. Finally, the evaluation machines differ between a 2.1 GHz 8 core processor used in [21] and a 3.1 GHz 4 core processor machine used in our evaluation. The authors report runtimes of 33 s, 499 s, and 632 s on the Pfam models SH3_1, Ras, and BID, respectively. On the same models, we achieve a runtime of 16.95 s, 180.68 s, and 241.61 s, respectively. In a more comparable setting (communication over the loopback interface, short-term security, and HMMER 2.3.2 style Forward computation), the runtime decreases by roughly a factor of 2 to 8.15 s, 89.92 s, and 125.25 s improving over [21] by a factor of 4 to 5. However, we emphasize again that latter results were still obtained on different machines and are thus only a rough indicator.

Aliasgari et al. [4,5], **Kamm et al. [28]** and **Demmler et al. [17]** propose secure computation on floats which could be used to implement a secure Forward algorithm over probability space and using normalization [38] to avoid underflows. Then, only selection of the required emission probabilities is not straightforward, but can be realized through component-wise multiplication which is the fastest previous method proposed in [21]. To draw a concrete comparison, we estimate runtimes by counting the calls to the required primitives and weighting them according to the performance measurements presented in [17], for which we choose a high batch-size of 1000 which yields very defensive estimates. In this setting, we estimate that Forward computation of $T = 100$ observations over an HMM with $N = 10$ states and an alphabet of $M = 100$ symbols would cost at least 251.82 s, 51.76 s, and 39.37 s using the primitives from [5], [28], and [17], respectively. In contrast, even when parameterized with $k = 8$ for high accuracy, our approach requires only 9.85 s which is approximately 25x, 5x, and 4x faster.

6. CONCLUSION

We presented *Priward*, which computes the HMM Forward algorithm in an efficient and privacy-preserving manner. At the core of our approach are efficient techniques to compute securely and accurately over non-integers in a fixed-point logspace representation, which are relevant beyond the scope of this work for a variety of privacy-preserving services [20, 35]. As a thorough evaluation shows, our novel and improved building blocks make *Priward* faster than previous works by factors of 4 to 126 while providing tuneable accuracy. Despite our significant improvements, secure computations on HMMs still involve overheads that may well overtax the resources of the protocol peers, e.g., drain the battery of a mobile user. To overcome such limitations, our approach allows outsourcing computations very efficiently and securely, e.g., to an untrusted computation cloud which remains oblivious of the inputs and results of the computation. As a concrete use case, we implement *Priward* in the widely used HMMER framework and demonstrate the feasibility of privacy-preserving bioinformatics services. Here, our approach improves upon the performance of related work by a factor of 4 to 5 while the accuracy of the computed results clearly satisfies the use case's requirements. To conclude, *Priward* provides the basis for a wide variety of privacy-preserving HMM-based services ranging from genomic testing over speech processing to localization and makes them affordable even for mobile users.

Acknowledgments

This work has been funded by the German Federal Ministry of Education and Research (ref. no. 16KIS0443). The responsibility for the content of this publication lies with the authors.

7. REFERENCES

[1] HMMER. http://hmmer.org/.
[2] Natural Language Toolkit. http://www.nltk.org/.
[3] Pfam Database, version 29.0. http://pfam.xfam.org/, 2015.
[4] M. Aliasgari and M. Blanton. Secure computation of Hidden Markov models. In *SECRYPT*, 2013.
[5] M. Aliasgari, et al. Secure Computation on Floating Point Numbers. In *NDSS*, 2013.

[6] G. Asharov, et al. More Efficient Oblivious Transfer and Extensions for Faster Secure Computation. In *ACM CCS*. ACM, 2013.
[7] E. Ayday, et al. The Chills and Thrills of Whole Genome Sequencing. *Computer*, 99(PrePrints):1, 2013.
[8] E. Barker, et al. Recommendation for Key Management - Part 1: General (Revised). In *NIST Special Publication 800-57*. NIST, 2007.
[9] D. Beaver. Efficient Multiparty Protocols Using Circuit Randomization. In *CRYPTO'91*. Springer, 1991.
[10] M. Bellare, et al. Efficient Garbling from a Fixed-Key Blockcipher. In *SP'13*. IEEE, 2013.
[11] M. Bellare, V. T. Hoang, and P. Rogaway. Foundations of garbled circuits. In *CCS'12*. ACM, 2012.
[12] M. Ben-Or, S. Goldwasser, and A. Wigderson. Completeness Theorems for Non-Cryptographic Fault-Tolerant Distributed Computation. In *STOC '88*. ACM, 1988.
[13] R. Canetti. Security and composition of multiparty cryptographic protocols. *Journal of CRYPTOLOGY*, 13(1):143–202, 2000.
[14] O. Catrina and S. De Hoogh. Improved Primitives for Secure Multiparty Integer Computation. In *SCN'10*. Springer, 2010.
[15] O. Catrina and A. Saxena. Secure Computation with Fixed-point Numbers. In *FC'10*. Springer, 2010.
[16] E. De Cristofaro, et al. Genodroid: Are Privacy-preserving Genomic Tests Ready for Prime Time? In *WPES'12*. ACM, 2012.
[17] D. Demmler, et al. Automated Synthesis of Optimized Circuits for Secure Computation. In *CCS'15*. ACM, 2015.
[18] D. Demmler, T. Schneider, and M. Zohner. ABY – A Framework for Efficient Mixed-Protocol Secure Two-Party Computation. In *NDSS'15*. The Internet Society, 2015.
[19] R. Durbin, et al. *Biological Sequence Analysis: Probabilistic Models of Proteins and Nucleic Acids*. Cambridge University Press, 1998.
[20] M. Franz, et al. Secure computations on non-integer values. In *WIFS'10*. IEEE, 2010.
[21] M. Franz, et al. Towards Secure Bioinformatics Services (Short Paper). In *FC'11*. Springer, 2011.
[22] N. Gilboa. Two party rsa key generation. In *Annual International Cryptology Conference*, pages 116–129. Springer, 1999.
[23] O. Goldreich. *Foundations of Cryptography: Volume 2 – Basic Applications*. Cambridge University Press, 2004.
[24] O. Goldreich, S. Micali, and A. Wigderson. How to Play ANY Mental Game. In *ACM STOC*, pages 218–229. ACM, 1987.
[25] W. Henecka, et al. TASTY: Tool for Automating Secure Two-party Computations. In *CCS'10*. ACM, 2010.
[26] Y. Huang, et al. Faster Secure Two-Party Computation Using Garbled Circuits. In *USENIX Security Symposium*, 2011.
[27] Y. Ishai, et al. Extending Oblivious Transfers Efficiently. In *CRYPTO'03*. Springer, 2003.
[28] L. Kamm and J. Willemson. Secure floating point arithmetic and private satellite collision analysis. *International Journal of Information Security*, 14(6):531–548, 2015.
[29] V. Kolesnikov, A.-R. Sadeghi, and T. Schneider. Improved Garbled Circuit Building Blocks and Applications to Auctions and Computing Minima. In *CANS'09*. Springer, 2009.
[30] V. Kolesnikov and T. Schneider. Improved Garbled Circuit: Free XOR Gates and Applications. In *ICALP '08*. Springer, 2008.
[31] B. Kreuter, A. Shelat, and C.-H. Shen. Billion-gate secure computation with malicious adversaries. In *USENIX Security*, 2012.
[32] M. Naor and B. Pinkas. Efficient Oblivious Transfer Protocols. In *SODA*, pages 448–457. SIAM, 2001.
[33] M. Naor and B. Pinkas. Computationally Secure Oblivious Transfer. *Journal of Cryptology*, 18(1):1–35, 2005.
[34] M. Pathak, et al. Privacy preserving probabilistic inference with Hidden Markov Models. In *ICASSP'11*. IEEE, 2011.
[35] M. A. Pathak, et al. Privacy-Preserving Speech Processing: Cryptographic and String-Matching Frameworks Show Promise. *IEEE Signal Processing Magazine*, 30(2):62–74, 2013.
[36] H. Polat, et al. Private predictions on hidden Markov models. *Artificial Intelligence Review*, 34(1):53–72, 2010.
[37] J. Portêlo, B. Raj, and I. Trancoso. Logsum Using Garbled Circuits. *PLoS ONE*, 10(3):1–16, 2015.
[38] L. R. Rabiner. A Tutorial on Hidden Markov Models and Selected Applications in Speech Recognition. *Proceedings of the IEEE*, 1989.
[39] A.-R. Sadeghi, T. Schneider, and I. Wehrenberg. Efficient privacy-preserving face recognition. In *ICISC*. Springer, 2009.
[40] P. Smaragdis and M. Shashanka. A Framework for Secure Speech Recognition. *TASLP*, 15(4), 2007.
[41] J. R. Troncoso-Pastoriza and F. Pérez-González. Efficient protocols for secure adaptive filtering. In *ICASSP'11*. IEEE, 2011.
[42] A. Yao. How to Generate and Exchange Secrets. In *IEEE SFCS*, 1986.
[43] S. Zahur, M. Rosulek, and D. Evans. Two Halves Make a Whole. In *EUROCRYPT*. Springer, 2015.
[44] P. J. Zettler, J. S. Sherkow, and H. T. Greely. 23andMe, the Food and Drug Administration, and the Future of Genetic Testing. *JAMA Internal Medicine*, 174(4):493–494, 2014.
[45] J. H. Ziegeldorf, et al. Poster: Privacy-preserving indoor localization. *WiSec'14*, 2014.

APPENDIX

A. SECURITY DISCUSSION

We now discuss the security of *Priward*. We assume the semi-honest adversary model (cf. Sect. 2.2). We first summarize security proofs of the building blocks that underly *Priward*. We then argue that our proposed protocols `Rescale`, `Emission`, `Logsum`, and `Forward` are secure based on modular sequential composition [13].

Security of the Building Blocks. We heavily rely on three well established secure computation techniques: OT, additive sharings, and GCs. First, Security of 1-n-OT has been proven in [32, 33] and the security of OT Extensions in [6, 27]. Second, additive sharings over $\mathbb{Z}_{2^l}$ realize perfect blinding and thus essentially represent perfectly secure One-Time-Pad encryption. We perform additions and multiplications over additive sharings. Additions are local operations and thus irrelevant for security in the semi-honest model, while the security of the multiplication protocol directly follows from [18, 22]. Finally, security of Garbled Circuits including recent optimizations has been rigorously addressed in [10, 11].

Security of `Rescale`. To argue security of `Rescale` (Sect. 4.1, Prot. 2), we show that neither party learns the value x that is rescaled. First, note that the shares $x^{\mathcal{U}}, x^{\mathcal{S}}$ are random and a single share conveys no information about x to either party. During the protocol, $\mathcal{U}$ blinds her share $x^{\mathcal{U}}$ by a $l+\kappa$ bit random number r and sends it to $\mathcal{S}$. $\mathcal{S}$ learns $x_r^{\mathcal{U}} + x^{\mathcal{S}} = x + r$. Blinding with a large random r over $\mathbb{Z}$ achieves statistical security towards $\mathcal{S}$ with security parameter κ. $\mathcal{U}$ receives no information from $\mathcal{S}$ and learns nothing about x.

Security of `Emission`. To show that our `Emission` primitive (Sect. 4.2, Prot. 3) is secure in the semi-honest model, we show that $\mathcal{U}$ does not learn anything about the emission scores $\hat{b}_i(v_j)$ held by $\mathcal{S}$ and $\mathcal{S}$ does not learn anything about $\mathcal{U}$'s observation o_t. In Step 1, $\mathcal{S}$ blinds the emission scores additively over $\mathbb{Z}_{2^l}$ with the random value $r_{\mathcal{S}} \in \mathbb{Z}_{2^l}$. From $\mathcal{U}'s$ perspective, the blinding represents a One-Time-Pad encryption which is perfectly secure. Since all emission scores are blinded by the same random value $r_{\mathcal{S}}$, we use 1-n-OT$_l^m$ to guarantee that $\mathcal{U}$ learns only exactly one blinded emission score in Step 2. $\mathcal{U}$ then arithmetically shares the value with $\mathcal{S}$, which does not leak information as the employed additive sharing uses perfect blinding over $\mathbb{Z}_{2^l}$. In the final step, $\mathcal{S}$ subtracts $r_{\mathcal{S}}$ from its share, which is a local operation and reveals no information. We conclude that `Emission` is secure.

Security of `Logsum`. To argue security of `Logsum` (Sect. 4.3, Prot. 4), we show that neither party learns anything about the summands $\hat{x}$ and $\hat{y}$ or the result $\hat{z}$. First note, that the summands $\hat{x}$ and $\hat{y}$ are given as additive shares and each party holds only a single share which does not reveal any information since shares appear completely random. Further, the PLA parameters k and $\mathcal{P}$ are com-

pletely independent of the inputs and thus reveal no information either. Steps 2, 3, and 4 are realized in one monolithic GC and involve i) input conversion, ii) the selection of approximation parameters, and iii) the conversion of outputs. We emphasize that we differentiate these three steps in our protocol description only for reasons of clarity but implement them in one single GC which yields better performance. Consisting of only one GC, security for these steps follows directly from the security of the GC building block. The output of these steps, i.e., $max^{\mathcal{U}/\mathcal{S}}$, $d^{\mathcal{U}/\mathcal{S}}$, $m^{\mathcal{U}/\mathcal{S}}$, and $n^{\mathcal{U}/\mathcal{S}}$, is additively shared over both parties which reveals no information to either party holding only a single share of each output since additive sharing implements perfect blinding over $\mathbb{Z}_{2^l}$. It is also important to note that the structure of the circuit is independent of all parameters except for the public parameter k, therefore leaking no sensitive information. Step 5 computes the product over additive shares using the secure protocols from [18, 22] and uses the secure `Rescale` on it. All outputs are again additively shared and reveal no information to either party. The last step involves an addition operation over additive shares which is executed locally and has no security implications in the semi-honest model. Finally, the output z is obtained by the two parties in shared form where a single share is indistinguishable from a random value and reveals no information. In summary, security of `Logsum` depends on the security of GCs and the `Rescale` protocol as well as the randomness of the additive sharings. As `Rescale` offers statistical security against a semi-honest $\mathcal{S}$, `Logsum` itself offers statistical security as well.

Security of `Forward`. To argue security of `Forward` (Sect. 4, Prot. 1), we show that $\mathcal{U}$ learns nothing about the HMM λ, i.e., $\mathcal{S}'s$ private input, and vice versa $\mathcal{S}$ learns nothing about the observations $o_1, ..., o_T$, i.e., $\mathcal{U}'s$ private input, except for what is implied in the final result $\hat{P}(O|\lambda)$. We argue that `Forward` is secure because the only interaction between $\mathcal{U}$ and $\mathcal{S}$ happens through the `Emission` and `Logsum` primitives. Since these primitives are secure and their output is received in the form of random additive sharings, their use reveals no information and neither does their composition according to the security of modular sequential compositions of semi-honest protocols [13]. All other steps are local operations that have no security implications in the semi-honest model. Finally, one or both parties learn the Forward score $\hat{P}(O|\lambda)$ by recombining their shares of the result, which is of course as intended.

Limitations. It is important to note that in our `Forward` design, $\mathcal{U}$ learns the dimensions of $\mathcal{S}'s$ HMM and, vice versa, $\mathcal{S}$ learns the length of $\mathcal{U}'s$ observation sequence. In this work and in most related works [4, 21, 40], such information is not considered sensitive. If desired, we can prevent leakage of this information by padding the HMM with dummy states and the observation sequence with predefined dummy symbols to a common predefined length. However, padding necessarily increases the size of the HMM and observation sequence and thus comes at the cost of performance and communication overheads. Finally, note that we develop protocols in the semi-honest model. However, established techniques to make semi-honest computation robust against malicious behavior [31] can be applied to our approach when the problem scenario requires this.

B. EVALUATION OF THE SECURE LOGSUM PRIMITIVE

Secure and accurate computation of logsum operations is challenging and has significant performance impacts in most previous works [21, 34, 35, 37]. Indeed, `Logsum` (Sect. 4.3, Prot. 4) clearly dominates the runtime of our approach. Since secure computation over non-integers is relevant beyond the scope of our work [15, 20], e.g., for securely evaluating Gaussian Mixture Models [35], we provide a detailed evaluation of the `Logsum` primitive in this section. All presented results are aggregated over 1000 runs and were obtained in the evaluation setup described in Section 5. The measured accuracy, runtime, and communication overheads for different choices of k and l are summarized in Table 2.

Accuracy. `Logsum` uses a piece-wise linear approximation with k intervals to compute the result. As expected, the error of the approximation decreases with increasing k. Increasing fixed-point precision by choosing $l = 64$ additionally improves accuracy, but only beyond $k = 8$. For the smaller values of k, the approximation error dominates the additional accuracy in the fixed-point representation. However, increasing k and l comes at the cost of runtime and communication overheads as analyzed in the following.

Runtime. We first measure runtime for *sequential* `Logsum` operations. The runtime grows linearly in k and l with 2.35 ms in our least accurate setup ($k = 2, l = 32$) to 23.52 ms in the most accurate setup ($k = 128, l = 64$). We now batch all m operations into one single invocation of `Logsum` and again report the average runtime of a single logsum operation. Sequential operation requires evaluation of m small GCs and requires $3 \cdot m$ rounds of communication, whereas batched operation corresponds to the evaluation of one very large GC in only three rounds of communication. Thus, batching achieves the best speed-up for small circuits (small k and l). For $l = 32$ and $k = 2$, we observe a speedup of 5.6x and still up to 4.13x for $l = 64, k = 2$. As expected, this decreases for bigger values of k, e.g., to 1.5x and 1.3x for $k = 128$ and $l = 32, 64$, respectively. We evaluated different batchsizes and observed no significant speedup beyond batches bigger than $m = 1000$. For both sequential and batched operation, we observe that to achieve a certain accuracy it is more efficient to increase k than to increase l.

Communication. Communication scales linearly in k and l and is in the order of tens to hundreds of kilobytes per run of `Logsum`. The dominating part is the transmission of two t bit keys per gate in the GC. Hence, communication overheads can be traded off against the security level t, e.g., switching to short term security $t = 80$ reduces communication by approximately 34 %. Also ongoing and future optimizations of the GC foundations [10, 43] will further reduce communication overheads.

Bit-length *l*	**32**							**64**						
PLA size *k*	**2**	**4**	**8**	**16**	**32**	**64**	**128**	**2**	**4**	**8**	**16**	**32**	**64**	**128**
Avg. abs. error	6.0e-2	4.4e-3	9.2e-4	5.3e-4	5.1e-4	5.4e-4	6.2e-4	6.0e-2	4.3e-3	7.7e-4	2.0e-4	5.5e-5	1.4e-5	2.7e-6
Runtime (seq.)	2.35	2.83	3.30	4.22	5.79	9.23	14.90	3.10	3.25	4.25	5.61	8.21	13.67	23.52
Runtime (batch)	0.42	0.59	0.89	1.49	2.70	5.10	9.98	0.75	1.00	1.52	2.61	4.66	8.88	17.74
Communication	0.02	0.02	0.03	0.05	0.08	0.15	0.28	0.03	0.04	0.05	0.08	0.14	0.26	0.49

Table 2: Evaluation of the secure `Logsum` primitive: The runtime [ms] and communication overhead [MB] increase and the approximation error decrease as the number of approximation intervals *k* and the bit-length *l* of the garbled circuits increase.