Online Appendix to: NN-EMD: Efficiently Training *Neural Networks* using *Encrypted Multi-sourced Datasets*

Runhua Xu, James Joshi, *Senior Member, IEEE* and Chao Li

I. Difference between Homomorphic Encryption and Functional Encryption

Homomorphic Encryption (HE) is a form of cryptosystem with an additional evaluation capability for computing over ciphertexts without access to the private secret key, in which the result of operations over the ciphertexts, when decrypted, match the result of operations as if they have been performed on the original plaintext. Some typical types of HE are *partially* homomorphic, *somewhat* homomorphic, *leveled fully* homomorphic, and *fully* homomorphic encryption according to the the capability of performing different classes of computations. Unlike traditional encryption scheme that includes three main algorithms: key generation (*Gen*), encryption (*Enc*), and decryption (*Dec*), an HE scheme also has an extra *evaluation* (*Eval*) algorithm. Formally, a HE scheme $\mathcal{E}_{\text{HE}}$ includes the above four algorithms such that

$$\mathcal{E}_{\text{HE}}.Dec_{sk}(\mathcal{E}_{\text{HE}}.Eval_{pk}(f, \mathcal{E}_{\text{HE}}.Enc_{pk}(m_1), ..., \mathcal{E}_{\text{HE}}.Enc_{pk}(m_n))) = f(m_1, ..., m_n),$$

where $\{m_1, ..., m_n\}$ are the message to be protected, pk and sk are the key pairs generated by the key generation algorithm. Regarding recent emerging deep neural networks model, the CryptoNets[1] tries to apply neural networks to encrypted data by employing a leveled homomorphic encryption scheme to the training data, which allows adding and multiplying encrypted messages but requires that one knows in advance the complexity of the arithmetic circuit. Besides, the work in [2] uses the open-source FHE toolkit HElib for neural network training in a stochastic gradient descent training method.

Functional Encryption (FE) is another form of cryptosystem that also supports the computation over the ciphertext. Typically, the FE $\mathcal{E}_{\text{FE}}$ includes four algorithms: setup, key generation, encryption and decryption algorithms such that

$$\mathcal{E}_{\text{FE}}.Dec_{sk_f}(\mathcal{E}_{\text{FE}}.Enc_{pk}(m_1), ..., \mathcal{E}_{\text{FE}}.Enc_{pk}(m_n)) = f(m_1, ..., m_n),$$

where the setup algorithm creates a public key pk and a master secret key msk, and key generation algorithm uses msk to generate a new functional private key sk_f associate with the functionality f. Those two algorithms usually are run by the a trusted third-party authority.

As presented above, the main similarity between the *FE* and *HE* support the computation over the ciphertext. In a high-level respective, the main difference between the functional encryption and the homomorphic encryption is that given an arbitrary function $f(\cdot)$, the homomorphic encryption allows to compute *an encrypted result of* $f(x)$ from an encrypted x, whereas the functional encryption allows to compute *a plaintext result of* $f(x)$ from an encrypted x. Intuitively, the function computation party in the HE scheme (i.e. the evaluation party) can only contribute its computation resource to obtain the encrypted function result, but cannot learn the function result unless it has the secret key, while the function computation party in the FE scheme (i.e., usually, the decryption party) can obtain the function result with the issued functional private key. Besides, expect for several most recently proposed decentralized FE schemes, the classic FE schemes are relied on a trusted third-party authority to provide key service such as issuing a functional private key associated to a specific functionality.

Unlike the HE-based secure computation techniques that have been widely adopted as an candidate solution for the secure computation for privacy-preserving machine learning (PPML), recently proposed FE-based PPML solutions such as in [3], [4], [5] also show its promise in efficiency and practicality. The proposal in [5] proposes a practical framework to perform partially encrypted and privacy-preserving predictions which combines adversarial training and functional encryption. The work in [4] initialize a CryptoNN framework that supports training a neural network model over encrypted data by using the FE to construct the secure computation mechanism. In addition, the proposal in [3] focuses on the privacy-preserving federated learning (PPFL) by utilizing the FE to construct the secure aggregation protocol to protect each participant's input in the PPFL.

II. Adopted Functional Encryption Schemes in Detail

Here we present the underlying adopted functional encryption schemes in our *NN-EMD* framework.

R. Xu and J. Joshi are with School of Computing and Information, University of Pittsburgh, PA, USA, 15260. C. Li is with Beijing Key Laboratory of Security and Privacy in Intelligent Transportation, Beijing Jiaotong University, Beijing, China, 100044.
Emails: runhua.xu@pitt.edu, jjoshi@pitt.edu,li.chao@bjtu.edu.cn

A. Single-Input Functional Encryption for Inner-Product

We adopt the single-input functional encryption for inner-product (SI-FEIP) proposed in [6]. In SI-FEIP scheme, the supported function is described as

$$f_{\text{SIIP}}(\boldsymbol{x}, \boldsymbol{y}) = \sum_{i=1}^{\eta}(x_i y_i) \ \text{ s.t. } \ |\boldsymbol{x}| = |\boldsymbol{y}| = \eta,$$

where $\boldsymbol{x}$ and $\boldsymbol{y}$ are two vectors of length η, from different parties. The SI-FEIP scheme $\mathcal{F}_{\text{S}}$ includes four algorithms: *Setup, SKGenerate, Encrypt, Decrypt*. Here, each algorithm is constructed as follows.

- *Setup*$(1^\lambda, \eta)$: This algorithm generates a master private key and common public key pair $(\boldsymbol{pk}_{\text{com}}, \boldsymbol{msk})$ based on a given security parameter λ and vector length parameter η. Specifically, on the inputs of security parameters λ and η, the algorithm first generates two samples as follows:

  $$(\mathbb{G}, p, g) \leftarrow_{\$} \textit{GroupGen}(1^\lambda)$$
  $$\boldsymbol{s} = (s_1, ..., s_\eta) \leftarrow_{\$} \mathbb{Z}_p^\eta$$

  and then sets $\boldsymbol{pk}_{\text{com}}$ and $\boldsymbol{msk}$ as follows:

  $$\boldsymbol{pk}_{\text{com}} = (g, h_i = g^{s_i})_{i\in[1,...,\eta]}$$
  $$\boldsymbol{msk} = \boldsymbol{s}$$

  It returns the pair $(\boldsymbol{pk}_{\text{com}}, \boldsymbol{msk})$.
- *SKGenerate*$(\boldsymbol{msk}, \boldsymbol{y})$: This algorithm takes the master private key *msk* and one vector $\boldsymbol{y}$ as input, and generates a functionally derived key $sk_{f_{\text{SIIP}}} = \langle \boldsymbol{y}, \boldsymbol{s}\rangle$ as output.
- *Encrypt*$(\boldsymbol{pk}_{\text{com}}, \boldsymbol{x})$: This algorithm outputs ciphertext *ct* of vector $\boldsymbol{x}$ using the public key $\boldsymbol{pk}_{\text{com}}$. Specifically, the algorithm first chooses a random $r \leftarrow_{\$} \mathbb{Z}_p$ and computes

  $$ct_0 = g^r.$$

  For each $i \in [1, ..., \eta]$, it computes

  $$ct_i = h_i^r \cdot g^{x_i}.$$

  Then the algorithm outputs the ciphertext $ct = (ct_0, \{ct_i\}_{i\in[1,...,\eta]})$.
- *Decrypt*$(\boldsymbol{pk}_{\text{com}}, ct, sk_{f_{\text{SIIP}}}, \boldsymbol{y})$: This algorithm takes the ciphertext *ct*, the public key $\boldsymbol{pk}_{\text{com}}$ and functional key $sk_{f_{\text{SIIP}}}$ for the vector $\boldsymbol{y}$ as input, and returns the inner-product $f_{\text{SIIP}}(\boldsymbol{x}, \boldsymbol{y})$. Specifically, the algorithm firstly compute the discrete logarithm in basis g as follows

  $$g^{\langle \boldsymbol{x}, \boldsymbol{y}\rangle} = \prod_{i\in[1,...,\eta]} ct_i^{y_i} / ct_0^{sk_f}.$$

  Then, $f_{\text{SIIP}}(\boldsymbol{x}, \boldsymbol{y}) = \langle \boldsymbol{x}, \boldsymbol{y}\rangle$ could be recovered.

B. Multi-Input Functional Encryption for Inner-Product

We employ the multi-input functional encryption for inner-product (MI-FEIP) construction derived from the work proposed in [7]. In the MI-FEIP scheme, the support function is defined as

$$f_{\text{MIIP}}((\boldsymbol{x}_1, \boldsymbol{x}_2, ..., \boldsymbol{x}_n), \boldsymbol{y}) = \sum_{i=1}^{n}\sum_{j=1}^{\eta_i}(x_{ij} y_{\sum_{k=1}^{i-1}\eta_k + j}) \text{ s.t. } |\boldsymbol{x}_i| = \eta_i, |\boldsymbol{y}| = \sum_{i=1}^{n}\eta_i,$$

where $\boldsymbol{x}_i$ and $\boldsymbol{y}$ are vectors from different parties.

Accordingly, the MI-FEIP scheme $\mathcal{F}_{\text{M}}$ includes five algorithms: *Setup, PKDistribute, SKGenerate, Encrypt, Decrypt* Below, we present the construction of each algorithm:

- *Setup*$(1^\lambda, \vec{\boldsymbol{\eta}}, n)$: It generates a master private key and public key pair $(\boldsymbol{pk}_{\text{com}}, \boldsymbol{mpk}, \boldsymbol{msk})$ given security parameter λ and functional parameters $\vec{\boldsymbol{\eta}}$ and n, where n is the maximum number of input parties while $\vec{\boldsymbol{\eta}}$ is a vector where each element represents the maximum input length vector of the corresponding party, and hence $|\vec{\boldsymbol{\eta}}| = n$. Specifically, the algorithm first generates secure parameters as

  $$\mathcal{G} = (\mathbb{G}, p, g) \leftarrow_{\$} \textit{GroupGen}(1^\lambda),$$

  Then, it generates several samples as

  $$\boldsymbol{a} = (1, a)^\intercal, a \leftarrow_{\$} \mathbb{Z}_p$$
  $$\boldsymbol{W}_i \leftarrow_{\$} \mathbb{Z}_p^{\eta_i \times 2}, i \in [1, ..., n]$$
  $$\boldsymbol{u}_i \leftarrow_{\$} \mathbb{Z}_p^{\eta_i}, i \in [1, ..., n]$$

Then, it generates the keys as

$$\begin{aligned}\boldsymbol{pk}_{\text{com}} &= (\mathbb{G}, p, g)\\ \boldsymbol{mpk} &= (\mathcal{G}, g^{\boldsymbol{a}}, g^{\boldsymbol{Wa}}),\\ \boldsymbol{msk} &= (\boldsymbol{W}, (\boldsymbol{u}_i)_{i\in[1,\ldots,n]}).\end{aligned}$$

- *PKDistribute($\boldsymbol{mpk}, \boldsymbol{msk}$, id_i)*: It delivers the public key $\boldsymbol{pk}_{\text{i}}$ for party id_i given the master public/private keys. Specifically, it looks up the existing keys via id_i and returns the *public key* as

$$\boldsymbol{pk}_i = (\mathcal{G}, g^{\boldsymbol{a}}, (\boldsymbol{Wa})_i, \boldsymbol{u}_i).$$

- *SKGenerate($\boldsymbol{mpk}, \boldsymbol{msk}, \boldsymbol{y}$)*: It takes the master public/private keys and vector $\mathbf{y}$ as inputs, and generates a function derived key $sk_{f_{\text{MIIP}}}$ as output. Specifically, the algorithm first partitions $\boldsymbol{y}$ into $(\boldsymbol{y}_1||\boldsymbol{y}_2||...||\boldsymbol{y}_n)$, where $|\boldsymbol{y}_i|$ is equal to η_i. Then it generates the function derived key as follows:

$$\boldsymbol{sk}_{f,\boldsymbol{y}} = (\{\boldsymbol{d}_i^{\intercal} \leftarrow \boldsymbol{y}_i^{\intercal}\boldsymbol{W}_i\}, z \leftarrow \sum \boldsymbol{y}_i^{\intercal}\boldsymbol{u}_i).$$

- *Encrypt($\boldsymbol{pk}_i, \boldsymbol{x}_i$)*: It outputs ciphertext *ct* of vector $\mathbf{x}_i$ using the public key $\boldsymbol{pk}_i$. Specifically, the algorithm first generates a random nonce $r_i \leftarrow_R \mathbb{Z}_p$, and then computes the ciphertext as follows:

$$\boldsymbol{ct}_i = (\boldsymbol{t}_i \leftarrow g^{\boldsymbol{a}r_i}, \boldsymbol{c}_i \leftarrow g^{\boldsymbol{x}_i} g^{\boldsymbol{u}_i} g^{(\boldsymbol{Wa})_i r_i}).$$

- *Decrypt($\boldsymbol{pk}_{\text{com}}, S_{\boldsymbol{ct}}, \boldsymbol{sk}_{f_{\text{MIIP}}}, \boldsymbol{y}$)*: It takes the ciphertext set $S_{\boldsymbol{ct}}$, the public key $\boldsymbol{pk}_{\text{com}}$ and functional key $\boldsymbol{sk}_{f_{\text{MIIP}}}$ as input, and returns the inner-product $f_{\text{MIIP}}(\{\boldsymbol{x}_i\}, \boldsymbol{y})$. Specifically, the algorithm first calculates as follows:

$$C = \frac{\prod_{i\in[1,\ldots,n]}([\boldsymbol{y}_i^{\intercal}\boldsymbol{c}_i]/[\boldsymbol{d}_i^{\intercal}\boldsymbol{t}_i])}{z},$$

and then recovers the function result as

$$f((\boldsymbol{x}_1, \boldsymbol{x}_2, ..., \boldsymbol{x}_n), \boldsymbol{y}) = \log_g(C).$$

REFERENCES

[1] Ran Gilad-Bachrach, Nathan Dowlin, Kim Laine, Kristin Lauter, Michael Naehrig, and John Wernsing. Cryptonets: Applying neural networks to encrypted data with high throughput and accuracy. In *ICML*, pages 201–210, 2016.

[2] Karthik Nandakumar, Nalini Ratha, Sharath Pankanti, and Shai Halevi. Towards deep neural network training on encrypted data. In *CVPR Workshops*, 2019.

[3] Runhua Xu, Nathalie Baracaldo, Yi Zhou, Ali Anwar, and Heiko Ludwig. Hybridalpha: An efficient approach for privacy-preserving federated learning. In *Proceedings of the 12th ACM Workshop on Artificial Intelligence and Security*, pages 13–23, 2019.

[4] Runhua Xu, James Joshi, and Chao Li. Cryptonn:training neural networks over encrypted data. In *2019 39th IEEE International Conference on Distributed Computing Systems (ICDCS)*, pages 1199–1209. IEEE, 2019.

[5] Théo Ryffel, Edouard Dufour Sans, Romain Gay, Francis Bach, and David Pointcheval. Partially encrypted machine learning using functional encryption. *arXiv preprint arXiv:1905.10214*, 2019.

[6] Michel Abdalla, Florian Bourse, Angelo De Caro, and David Pointcheval. Simple functional encryption schemes for inner products. In *IACR PKC*, pages 733–751. Springer, 2015.

[7] Michel Abdalla, Dario Catalano, Dario Fiore, Romain Gay, and Bogdan Ursu. Multi-input functional encryption for inner products: function-hiding realizations and constructions without pairings. In *CRYPTO*, pages 597–627. Springer, 2018.