Toward Practical Lattice-based Unbounded Inner Product Functional Encryption: Construction and Implementation

Suprava Roy[1*] and Ratna Dutta[1]

[1]Department of Mathematics, Indian Institute of Technology Kharagpur, , Kharagpur, 721302, West Bengal, India.

*Corresponding author(s). E-mail(s): supravaroy9@kgpian.iitkgp.ac.in;
Contributing authors: ratna@maths.iitkgp.ac.in;

Abstract

Cloud computing enables data processing, storing and sharing in untrusted environments whose growing adoption necessitates a focus on data security and privacy. *Inner product functional encryption* (IPFE) is a promising cryptographic technique that enables fine-grained access control over sensitive data in untrusted cloud environments. *Post-quantum cryptography* focuses on developing cryptographic protocols resilient to quantum computer attacks, with lattice structures being crucial in designing these protocols.

This paper aims to implement the lattice-based public key unbounded IPFE (uIPFE) scheme proposed by Dutta et al. (2022) [1] which ensures that no specific vector's length bound needs to be fixed during public parameter generation.

Furthermore, we extend the ALS-IPFE scheme of Agrawal et al. (2016) [2] to create a new public key scheme uIPFE #1, achieving adaptive indistinguishability security in the random oracle model under the Learning with Errors assumption, while avoiding trapdoor generation and pre-image sampling algorithms.

This work aims to enhance the practical applicability of uIPFE in cloud computing environments. We implement both the schemes uIPFE and uIPFE #1 in C programming language and execute the code on IBM Power 9 server. Moreover, we analyze the running time and discuss the performance of both schemes based on varying message vector lengths.

Keywords: Inner product functional encryption, unbounded, cloud computing, adaptive, post-quantum security

1 Introduction

The global embrace of smart technology has led to an exceptional surge in electronic data generation. Effectively managing and sharing the vast amount of data generated across various sectors becomes critical with this increase in digital data volume. Cloud servers have appeared as a viable solution for data storage and offer seamless data/message transmission and cost-effectiveness across various sectors, encompassing medicine, digital data storage, social media, large-scale data analytics and e-learning platforms. For instance, the *Internet of Medical Things* (IoMT) enables seamless connectivity between medical devices and cloud-based healthcare services, facilitating remote patient monitoring and predictive analytics. The IoMT refers to a network comprising medical devices, software, sensors and systems that work together and integrate with healthcare IT infrastructures. This innovation is transforming the healthcare sector by enabling real-time data collection, continuous monitoring and advanced data analysis. To conduct such analysis effectively, the IoMT system relies on automation tools or services at the central site that process the data while guarantee security and privacy. However, ensuring data privacy while outsourcing computations to the cloud remains a challenge. *Functional Encryption* (FE) [3] addresses this issue by allowing computations on encrypted medical data, enabling secure AI-driven analysis without exposing sensitive patient information.

FE surpasses the barrier of existing encryption systems that impose an all-or-nothing approach to decryption capabilities. From its inception, FE has offered solutions to many unresolved cryptographic challenges, covering diverse areas such as obfuscation, deniable encryption, distributed protocols and beyond. The fundamental advantage of FE is the provision of fine-grained access control over message space. Specifically, FE for a function class $\mathcal{F}$ employs a master secret key Msk to generate decryption-key Dk_f for any function $f \in \mathcal{F}$ and a master public key Mpk to create ciphertexts Ct_M corresponding to a message M, ensuring that the recipient learns no more about the message M than $f(M)$ after decrypting Ct_M.

The primary security demand of FE is *collusion resistance* which ensures that a collective set of secret keys cannot collectively divulge any details about the message M except what can be derived from each individual secret key. This concept is defined through an *indistinguishability-based* (IND) security experiment in which any polynomial-time adversary $\mathcal{A}$ selects challenge messages M_0 and M_1 and is provided access to the key extraction oracle. The task of $\mathcal{A}$ is to differentiate between two ciphertexts Ct_{M_0} and Ct_{M_1}. For any function f queried to the key extraction oracle, the adversary $\mathcal{A}$ is required to maintain the condition that $f(M_0) = f(M_1)$ in order to be restricted from trivially differentiating the ciphertexts. In selective IND (Sel-IND) setting, $\mathcal{A}$ is required to commit the challenge messages M_0, M_1, prior to the commencement of the experiment whereas adaptive IND (Adp-IND) is more robust as it gives $\mathcal{A}$ the freedom to choose M_0, M_1 at any point of the security experiment.

Inner product functional encryption (IPFE) is a specific instance of FE that leverages inner product computations for access control. It is suitable for scenarios where fine-grained management of encrypted data is required for vector-based data representations. Abdalla et al. [4] introduced the notion IPFE where a decryption key $\mathsf{Dk}_{\boldsymbol{y}}$ for a key vector $\boldsymbol{y}$ reveals the inner product $\langle \boldsymbol{x}, \boldsymbol{y} \rangle$ after decrypting a ciphertext $\mathsf{Ct}_{\boldsymbol{x}}$

corresponding to a message vector $\boldsymbol{x}$. The ciphertext $\mathsf{Ct}_{\boldsymbol{x}}$ is generated using a master public key Mpk and the decryption key $\mathsf{Dk}_{\boldsymbol{y}}$ is derived using master secret key Msk. The master public-secret key pair $(\mathsf{Mpk}, \mathsf{Msk})$ is generated during Setup by a key generation centre. The authors provided several IPFEs in Sel-IND model based on the standard assumptions. Later, the Sel-IND security was improved by Agrawal at el. [2] to Adp-IND security and IPFE constrained was proposed relying on the standard assumptions achieving Adp-IND security. In recent years, substantial research has been dedicated to these frameworks through various techniques. In case of IPFEs, this includes improved security guarantees (e.g. [2, 5, 6]), multi-users extension (e.g. [7, 8]) and unbounded size vectors (e.g. [9, 10]).

The majority of IPFEs are classified as '*bounded*' since they can solely perform inner product computations on vectors of fixed lengths predefined during the generation of public parameters. Since the size of public parameters in a bounded IPFE increases proportionally with the magnitude of vectors, leading to an unnecessary efficiency decline. Constructing an *unbounded* IPFE (uIPFE) is gaining attraction in research community since it makes it possible to generate public parameters without having any previous information regarding the magnitude of any vector. This indicates that the size of the master secret key and master public key does not depend on the length of the input vectors. Concurrently, Takashima et al. [9] and Dufour-Sans et al. [10] introduced the unbounded feature within the framework of IPFE. It is worth noting that both constructions attained IND security by leveraging pairing-based assumptions. In the literature, two distinct concepts regarding unbounded vectors exist: strict and permissive. The strict notion guarantees that the computation of inner product is viable only when the index sets associated with the message and key vectors are identical. On the other hand, in a permissive scenario, computation becomes feasible exclusively when the index set of the key vector is contained in the index set of the message vector.

To understand how an IPFE works, let us consider an example. In IoMT, hospitals and health monitoring devices continuously collect sensitive patient data, such as heart rate, glucose levels and ECG readings. However, privacy concerns prevent hospitals from directly sharing raw patient data with cloud-based AI services for predictive analysis. Patient health data is represented as a vector $\boldsymbol{x}$, encrypted employing an IPFE scheme to generate a ciphertext $\mathsf{Ct}_{\boldsymbol{x}}$ using a master public key Mpk and sent to the cloud. A hospital or AI provider defines a weight vector $\boldsymbol{y}$ corresponding to a machine learning model. The functional key or decryption key $\mathsf{Dk}_{\boldsymbol{y}}$ for $\boldsymbol{y}$ is generated by using the master secret key Msk and provided to the cloud server. The Msk and Mpk are generated by the key generation centre in the Setup phase. To obtain AI-based health insights, the cloud server evaluates the inner product computation on encrypted data without learning the sensitive health data $\boldsymbol{x}$. The unbounded property of uIPFE ensures that the sizes of $\boldsymbol{x}$ and $\boldsymbol{y}$ do not need to be fixed beforehand as required in this is more realistic scenario since patient health data $\boldsymbol{x}$ may not fully known at the beginning of the treatment. In Figure 1, the patient's health sheet, denoted as $\boldsymbol{x}_i$, is continuously updated as the treatment progresses. The health components, represented by $\boldsymbol{y}_i$, are assigned specific weights determined by Clinical and Laboratory Standards Institute (CLSI). As uIPFE allows generation of the ciphertext and secret

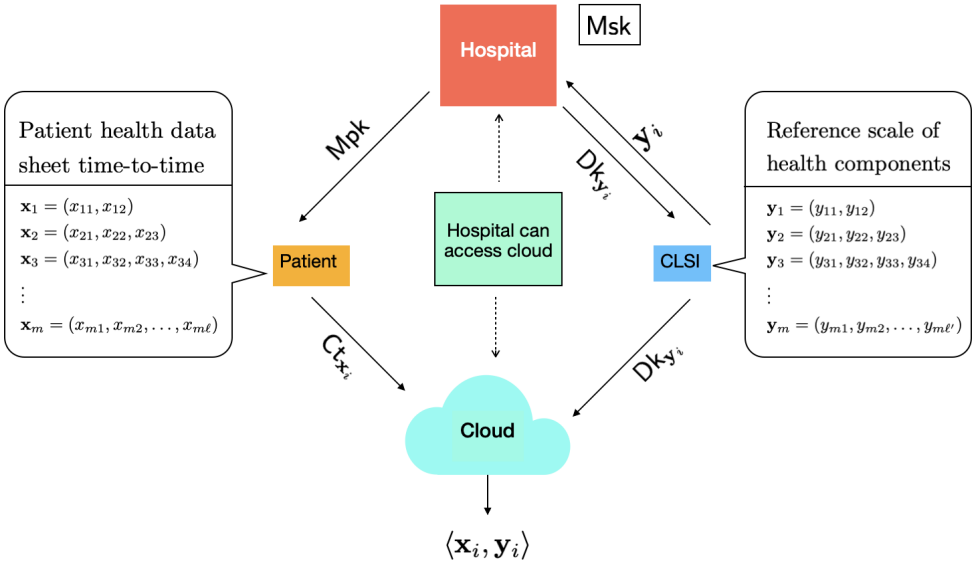

Fig. 1: Application of uIPFE in IoMT

keys for all updated versions are encrypted using the *same* master public-secret key pair (Mpk, Msk), it is an ideal candidate in computing the required statistics while accommodating variations in data size over time. Consequently, unbounded property enhances efficiency and makes IPFE more user-friendly. Without this unbounded feature, encrypting health parameters using IPFE would require frequent updates of (Mpk, Msk) pair by the key generation centre.

All existing works [9, 10], except [1] based on uIPFE relies on discrete logarithm-based assumptions such as symmetric external Diffie-Hellman (SXDH) and decisional bilinear Diffie-Hellman assumption (DBDH), making them vulnerable to quantum attacks. Considering the practical significance of uIPFE and the growing threat of quantum attacks, our objective is to develop efficient post-quantum resistant uIPFE based on learning with errors (LWE) assumptions, an area that had not been explored prior to the work of Dutta et al. [1].

Our Contribution. In this work, we aim to implement the uIPFE of [1] based on the standard LWE assumption. The uIPFE was originally introduced by Dutta et al. in [1] and later was published in [11]. In addition, we have designed and implemented a new public-key unbounded IPFE scheme, referred to as uIPFE #1, that eliminates the need for both trapdoor generation and preimage sampling algorithms. Both the schemes are strict in the sense that they ensure successful decryption when the index set of the message coincides with the secret key vector. From a practical standpoint, the uIPFE

protocol is particularly fruitful in scenarios where the message length is unknown in advance as it allows inner product computation for input vectors of arbitrary size.

Table 1 furnishes a comparative analysis of uIPFE [1] and our scheme uIPFE #1 with other existing uIPFE schemes. The uIPFE [1] and our uIPFE #1 rely on the LWE assumption whereas other constructions depend on number theoretic assumptions like DBDH and SXDH assumptions. We consider implementation of the uIPFE of Dutta et al. [1] as it is the first lattice-based uIPFE scheme, more efficient regarding dimensions of ciphertext and decryption key than the prior works. To be more specific, the approach in uIPFE [1] necessitates ciphertexts of linear size and secret keys of constant size without the use of any bilinear maps and exponentiation operations. Pairing operations are widely recognized to involve complex arithmetic over large extension fields and multiple sub-operations such as elliptic curve multiplications and final exponentiations, making them inherently intensive. In contrast, the uIPFE [1] and our uIPFE #1 are more flexible, efficient and demand significantly lower computations. To date, public-key uIPFE constructions have exclusively relied on bilinear maps and the development of pairing-free schemes grounded in standard cryptographic assumptions remains an open and largely unexplored direction. In the implementation point of view, we observe that there are some implementaions on pairing-based IPFE schemes including uIPFE like [12], but there is no existing implementation of a post-quantum secure uIPFE. So we choose the first and only existing lattice-based uIPFE for our implementation as lattice-basaed constructions are quantum-secure. Also for fast execution, we use parallel execution for some parts of our code and we make it as fast as possible. Considering the substantial computational overhead of pairings, the design of pairing-free uIPFE schemes presents a more pragmatic and efficient option. Such schemes can further serve as promising candidates for building lattice-based IPFE constructions, as eliminating pairing operations leads to considerably lower computational costs compared to their pairing-based counterparts.

Our implementation is in C language which mainly involves the implementation of lattice-based algorithms $\mathsf{TrapdoorGen}(\cdot)$ and $\mathsf{SamplePreimage}(\cdot)$ and their integration in the uIPFE. The algorithm $\mathsf{TrapdoorGen}(\cdot)$ is a trapdoor generation algorithm produces a random matrix $\mathbf{A} \in \mathbb{Z}_q^{n\times m}$ along with a strong trapdoor $\mathbf{T} \in \mathbb{Z}^{m\times m}$ consisting of short integer vectors satisfying $\mathbf{AT} = \mathbf{0} \mod q$ where q is a prime number. The matrix $\mathbf{T}$ is often interpreted as a basis for the lattice $\Lambda_q^{\perp}(\mathbf{A})$ defined by $\mathbf{A}$ where $\mathbf{A}$ serves as a 'parity-check' matrix. The algorithm $\mathsf{SamplePreimage}(\cdot)$ is the pre-image sampling algorithm that uses the trapdoor $\mathbf{T}$ to efficiently compute a short preimage $\mathbf{Z} \in \mathbb{Z}^{m\times \ell}$ for any given target image $\mathbf{U} \in \mathbb{Z}^{n\times \ell}$ such that $\mathbf{AZ} = \mathbf{U}$. The uIPFE [1] is adaptively indistinguishable secure in the random oracle model under the hardness problem of LWE. They leverage the security of the IBE of Gentry et al. [13] to handle all secret keys that do not decrypt the challenge ciphertext and employ the proof techniques from [2, 6] to handle the secret keys that successfully decrypt the challenge ciphertext. A high level overview of this uIPFE is described below in subsection 2.4.
The setup phase generates system parameter $n, q, m, \ell_{\mathsf{max}}, X, Y, K, \psi, \alpha, \sigma$ and executes the $\mathsf{TrapdoorGen}(n, q, m, \psi)$ [14] to generate the pair $(\mathbf{A}, \mathbf{T}) \in \mathbb{Z}_q^{n\times m} \times \mathbb{Z}^{m\times m}$ and sets $\mathbf{A} \in \mathbb{Z}_q^{n\times m}$ as the master public key and $\mathbf{T} \in \mathbb{Z}^{m\times m}$ as the corresponding

Table 1: A comparative analysis of various existing IPFE schemes

uIPFE schemes	|Msk|	|Ct|	|Dk|	Total Operations	Assum.	Security	ROM /SM
[10]	$\|\mathbb{G}_1\|+\|\mathbb{Z}_q\|$	$(\|I\|+1)\|\mathbb{G}\|$	$\|I'\|\|\mathbb{Z}_q\|+\|\mathbb{G}_2\|$	$(2\|I\|+1)\cdot \mathsf{p}+(2\|I\|+\|I'\|+1)\cdot \mathsf{m}+(2\|I\|+\|I'\|+2)\cdot \mathsf{e}$	DBDH	Sel	ROM
	$\|\mathbb{G}_1\|+\|\mathbb{Z}_q\|$	$(\|I\|+1)\|\mathbb{G}\|$	$\|I'\|\ \|\mathbb{Z}_q\|+\ \|\mathbb{G}_2\|$	$(2\|I\|+1)\cdot \mathsf{p}+(2\|I\|+\|I'\|+1)\cdot \mathsf{m}+(2\|I\|+\|I'\|+2)\cdot \mathsf{e}$	Lin. Ext. DBDH	Sel	ROM
[9]	$\|\mathsf{PRF\ key}\|$	$4\|I\|\ \|\mathbb{G}_1\|$	$4\|I'\|\ \|\mathbb{G}_2\|$	$(\|I'\|)\cdot \mathsf{p}+(16\|I\|+16\|I'\|)\cdot \mathsf{m}+(4\|I\|+4\|I'\|)\cdot \mathsf{e}$	SXDH	Adp-FH	SM
	$28\cdot\|\mathbb{Z}_q\|$	$7\|I\|\|\mathbb{G}_1\|$	$7\|I'\|\|\mathbb{G}_2\|$	$(\|I'\|)\cdot \mathsf{p}+(49\|I\|+49\|I'\|)\cdot \mathsf{m}+(7\|I\|+7\|I'\|+28)\cdot \mathsf{e}$	SXDH	Adp	SM
[1]	$\mathcal{O}(1)$	$(\|I\|+m)\ \|\mathbb{Z}_q\|$	$(\|I'\|+m)\ \|\mathbb{Z}_q\|$	$(mn+m+n\|I\|+m\|I'\|+\|I\|+\|I'\|)\cdot \mathsf{m}+$ $0\cdot \mathsf{p}+0\cdot \mathsf{e}+\mathsf{t}+\mathsf{s}$	LWE	Adp	ROM
Our work	$\mathcal{O}(1)$	$(\|I\|+m)\ \|\mathbb{Z}_q\|$	$(\|I'\|+m)\ \|\mathbb{Z}_q\|$	$(mn+m+(n\|I\|+m\|I'\|+mn)\ell_{\mathsf{max}}+\|I\|+\|I'\|)\cdot \mathsf{m}+$ $0\cdot \mathsf{p}+0\cdot \mathsf{e}+0\cdot \mathsf{t}+0\cdot \mathsf{s}$	LWE	Adp	ROM

- Adp, FH, Sel: adaptive, functional-hiding and selective respectively.
- Lin. Ext.: Linearly Extended.
- DBDH, SXDH: decisional bilinear Diffie-Hellman and symmetric external Diffie-Hellman assumptions respectively.
- |Msk|, |Ct|, |Dk|: the size of the master secret key, ciphertext and decryption key respectively.
- $|I|, |I'|$: magnitude of the encrypted vector and the decryption key vector respectively.
- ℓ_{max}: maximum magnitude of the encrypted vector and the decryption key vector.
- q: a large prime.
- m, n: order of matrix.
- $|\mathbb{G}|, |\mathbb{G}_1|, |\mathbb{G}_2|, |\mathbb{Z}_q|, |\mathsf{PRF\ key}|$: size of the groups $\mathbb{G}, \mathbb{G}_1, \mathbb{G}_2, \mathbb{Z}_q$ and size of pseudorandom key respectively.
- p, m, e, t, s: number of pairings, multiplications, exponents, trapdoor generation and preimage sampling operations respectively.
- SM, ROM: standard model and random oracle model respectively.

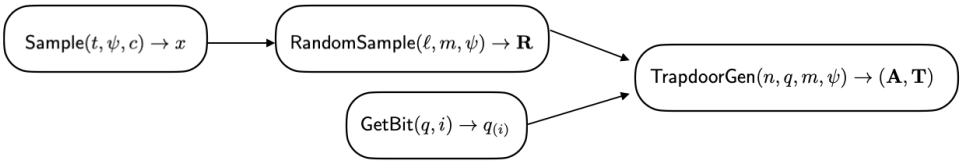

Fig. 2: Basic Overview of algorithm TrapdoorGen

master secret key. The trapdoor $\mathbf{T}$ serves as a short basis for the lattice $\Lambda_q^{\perp}(\mathbf{A})$ satisfying $\mathbf{AT} = \mathbf{0} \bmod q$ where q is a prime and $m = n\lceil \log q \rceil$. A full domain hash function $\mathcal{H} : \mathbb{N} \to \mathbb{Z}_q^n$ is used which is modeled as a random oracle. For encryption, the indices of the message vector $\boldsymbol{x} = (x_i)_{i \in I_{\boldsymbol{x}}} \in \mathbb{Z}_q^{|I_{\boldsymbol{x}}|}$ where $\boldsymbol{u}_i = \mathcal{H}(I_{\boldsymbol{x}}[i]) \in \mathbb{Z}_q^n$ and $I_{\boldsymbol{x}}[i]$ is the i-th element of the index set $I_{\boldsymbol{x}} \subset \mathbb{N}$ of $\boldsymbol{x}$, are used to construct a matrix $\mathbf{U} = [\boldsymbol{u}_1 \| \boldsymbol{u}_2 \| \cdots \| \boldsymbol{u}_{|I_{\boldsymbol{x}}|}] \in \mathbb{Z}_q^{n \times |I_{\boldsymbol{x}}|}$. A column vector $\boldsymbol{c}_1 = \mathbf{A}^\top \boldsymbol{s} + \boldsymbol{e}_1 \in \mathbb{Z}_q^m$ is set as the first ciphertext component where $\boldsymbol{s}$ is uniformly random column vectors in $\mathbb{Z}_q^n$ and $\boldsymbol{e}_1$ is selected at random from discrete Gaussian distribution $\mathfrak{D}_{\mathbb{Z}^m, \psi}$ with ψ is Gaussian parameter. a column vector $\boldsymbol{c}_2 = \mathbf{U}^\top \boldsymbol{s} + \lfloor \frac{q}{K} \rfloor \boldsymbol{x} + \boldsymbol{e}_2 \in \mathbb{Z}_q^{|I_{\boldsymbol{x}}|}$ where K is set as polynomially constrained in the security parameter λ, is used to generate the second ciphertext component. The decryption key related to secret key column vector $\boldsymbol{y} = (y_i)_{i \in I_{\boldsymbol{y}}} \in \mathbb{Z}_q^{|I_{\boldsymbol{y}}|}$ is set as $\mathsf{Dk}_{\boldsymbol{y}} = (\boldsymbol{y}^\top, \boldsymbol{y}^\top \mathbf{Z}^\top) \in \mathbb{Z}_q^{1 \times |I_{\boldsymbol{y}}|} \times \mathbb{Z}_q^{1 \times m}$ where $\mathbf{Z} \in \mathbb{Z}^{m \times |I_{\boldsymbol{y}}|}$ is produced by using the preimage sampling algorithm $\mathsf{SamplePreimage}(n, q, \mathbf{A}, \mathbf{T}, \sigma, \mathbf{U}', \ell')$ [15] with $\mathbf{U}' = [\boldsymbol{u}_1' \| \boldsymbol{u}_2' \| \ldots \| \boldsymbol{u}_{|I_{\boldsymbol{y}}|}'] \in \mathbb{Z}^{n \times |I_{\boldsymbol{y}}|}$, $\boldsymbol{u}_i' = \mathcal{H}(I_{\boldsymbol{y}}[i]) \in \mathbb{Z}_q^n$, $I_{\boldsymbol{y}}[i]$ is the i-th element of the index set $I_{\boldsymbol{y}} \subset \mathbb{N}$ of $\boldsymbol{y}$ using the master secret key $\mathbf{T}$ with σ as a Gaussian parameter and $\mathbf{Z} \in \mathbb{Z}^{m \times |I_{\boldsymbol{y}}|}$ satisfying $\mathbf{U}' = \mathbf{AZ} \in \mathbb{Z}^{n \times |I_{\boldsymbol{y}}|}$. The decryptor can use the decryption key $\mathsf{Dk}_{\boldsymbol{y}} = (\boldsymbol{y}^\top, \boldsymbol{y}^\top \mathbf{Z}^\top) \in \mathbb{Z}_q^{1 \times |I_{\boldsymbol{y}}|} \times \mathbb{Z}_q^{1 \times m}$ to recover the inner product $\langle \boldsymbol{x}, \boldsymbol{y} \rangle$ from the ciphertext $\mathsf{Ct}_{\boldsymbol{x}} = (\boldsymbol{c}_1, \boldsymbol{c}_2)$ if $I_{\boldsymbol{x}} = I_{\boldsymbol{y}}$ by computing

$$\mu' = \boldsymbol{y}^\top \boldsymbol{c}_2 - \boldsymbol{y}^\top \mathbf{Z}^\top \boldsymbol{c}_1 \approx \langle \boldsymbol{x}, \boldsymbol{y} \rangle + \mathsf{noise}.$$

Our implementation efficiently sets the parameters to minimize the noise function. We carefully select the parameters that satisfy the necessary conditions for managing the noise function effectively. We emphasize a program for parameter choice described in subsection 3.2. If the error quantity $\mathsf{noise} \leq \sigma\psi m\ell_{\mathsf{max}} Y + Y\ell_{\mathsf{max}}\psi \leq \frac{q}{2K}$, the decryption will produce the correct result i.e. $q \geq 2K(\sigma\psi m\ell_{\mathsf{max}} Y + Y\ell_{\mathsf{max}}\psi)$.

We now turn the attention to outline the implementation aspects used for our experimental validation of the uIPFE [1] and the challenges we found. Note that the performance of the protocol is mainly affected by the procedures TrapdoorGen and SamplePreimage. We implement the algorithms TrapdoorGen and SamplePreimage in C language and execute in the IBM power 9 server. Subsequently, we integrate these into our implementation of the uIPFE [1]. For our implementation, the trapdoor generation algorithm TrapdoorGen is adopted from the trapdoor generation algorithm of Micciancio et al. [14] and the Gaussian preimage sampling SamplePreimage is adopted from the works of Bert et al. [15] and Gur et al. [16]. We provide pseudocodes of

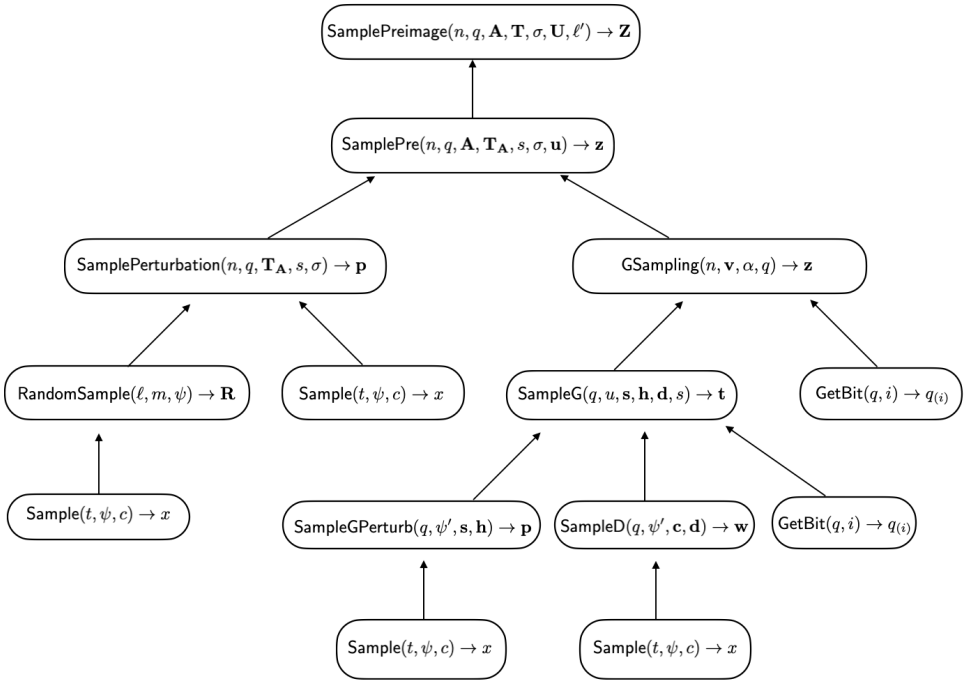

Fig. 3: Basic Overview of algorithm SamplePreimage

these algorithms and discuss the conditions needed to choose their parameters. Figures 2 and 3 depict a rough outline of the algorithms TrapdoorGen and SamplePreimage respectively. The algorithm $\mathsf{TrapdoorGen}(n, q, m, \psi) \to (\mathbf{A}, \mathbf{T})$ uses procedures $\mathsf{RandomSample}(\ell, m, \psi) \to \mathbf{R}$ and $\mathsf{GetBit}(q, i) \to q_{(i)}$ as subroutines to generate a random matrix $\mathbf{A} \in \mathbb{Z}_q^{n \times m}$ and a trapdoor basis $\mathbf{T} \in \mathbb{Z}^{m \times m}$ for lattice $\Lambda_q^{\perp}(\mathbf{A})$. The procedure $\mathsf{RandomSample}(\ell, m, \psi) \to \mathbf{R}$ generates random matrix $\mathbf{R}$ of order $\ell \times m$ sampled from discrete Gaussian distribution $\mathfrak{D}_{\mathbb{Z}^{\ell \times m}, \psi}$ using a Gaussian sampler $\mathsf{Sample}(t, \psi, c) \to x$ as subroutine. The procedure $\mathsf{Sample}(t, \psi, c) \to x$, adapted from the work of Ducas et al. [17] is sampler of discrete Gaussian distributions over $\mathbb{Z}$. The procedure $\mathsf{GetBit}(q, i) \to q_{(i)}$ returns i-th bit of a given integer q. The algorithm $\mathsf{SamplePreimage}(n, q, \mathbf{A}, \mathbf{T}, \sigma, \mathbf{U}, \ell') \to \mathbf{Z}$ generates a matrix $\mathbf{Z} \in \mathbb{Z}^{m \times \ell'}$ by running a procedure $\mathsf{SamplePre}(n, q, \mathbf{A}, \mathbf{T_A}, s, \sigma, \boldsymbol{u}) \to \boldsymbol{z}$ as subroutine to generate each column vector $\boldsymbol{z} \in \mathbb{Z}^m$ of the matrix $\mathbf{Z} \in \mathbb{Z}^{m \times \ell'}$ which can be run in parallel, given a matrix $\mathbf{A} \in \mathbb{Z}_q^{n \times m}$, its trapdoor basis $\mathbf{T} \in \mathbb{Z}^{m \times m}$ and a matrix $\mathbf{U} \in \mathbb{Z}^{n \times \ell'}$ such that $\mathbf{AZ} = \mathbf{U} \bmod q$. The procedure $\mathsf{SamplePre}(n, q, \mathbf{A}, \mathbf{T_A}, s, \sigma, \boldsymbol{u}) \to \boldsymbol{z}$ calls $\mathsf{SamplePerturbation}(n, q, \mathbf{T_A}, s, \sigma) \to \boldsymbol{p}$ and $\mathsf{GSampling}(n, \boldsymbol{v}, \alpha, q) \to \boldsymbol{z}$ as subroutines to generate each column vector $\boldsymbol{z} \in \mathbb{Z}^m$ of the matrix $\mathbf{Z} \in \mathbb{Z}^{m \times \ell'}$, given matrices $\mathbf{A} \in \mathbb{Z}_q^{n \times m}$, $\mathbf{T_A} \in \mathbb{Z}^{\overline{m} \times \omega}$ and column vector $\boldsymbol{u} \in \mathbb{Z}^n$ of matrix $\mathbf{U} \in \mathbb{Z}^{n \times \ell'}$ such that $\mathbf{A}\boldsymbol{z} = \boldsymbol{u} \bmod q$. The Perturbation sampling algorithm $\mathsf{SamplePerturbation}(n, q, \mathbf{T_A}, s, \sigma) \to \boldsymbol{p}$, aimed at generating vectors that adhere to a discrete Gaussian distribution on

$\mathbb{Z}^m$ with a covariance dependent on the trapdoor $\mathbf{T} \in \mathbb{Z}^{m \times m}$, was developed by Genise and Micciancio [18] and modified by Gur et al. [16]. On the other hand, $\mathsf{GSampling}(n, \boldsymbol{v}, \alpha, q) \rightarrow \boldsymbol{z}'$ computes a vector $\boldsymbol{z}' \in \mathbb{Z}^{\omega}$ following a discrete Gaussian distribution of parameter s over a coset $\Lambda_q^{\boldsymbol{v}}(\mathbf{G})$ of the G-lattice $\Lambda_q^{\perp}(\mathbf{G})$ given an n-dimensional target vector $\boldsymbol{v} \in \mathbb{Z}^m$ by using the procedure $\mathsf{SampleG}(q, u, \boldsymbol{s}, \boldsymbol{h}, \boldsymbol{d}, s) \rightarrow \boldsymbol{t}$. The procedure $\mathsf{SampleG}(q, u, \boldsymbol{u}, \boldsymbol{h}, \boldsymbol{d}, s) \rightarrow \boldsymbol{t}$ is designed to work with a single integer component $v_i \in \mathbb{Z}$ of the m-dimensional target vector $\boldsymbol{v} \in \mathbb{Z}^m$ instead of $\boldsymbol{v} \in \mathbb{Z}^m$ because each of the m integers can be sampled parallelly by running $\mathsf{SampleG}$ inside the function $\mathsf{GSampling}(n, \boldsymbol{v}, \alpha, q) \rightarrow \boldsymbol{z}'$ to generate the vector $\boldsymbol{z}' \in \mathbb{Z}^{\omega}$. More specifically, this $\mathsf{SampleG}$ uses a perturbation sampling procedure $\mathsf{SampleGPerturb}(q, \psi', \boldsymbol{s}, \boldsymbol{h}) \rightarrow \boldsymbol{p}$ and a discrete Gaussian sampling procedure $\mathsf{SampleD}(q, \psi', \boldsymbol{c}, \boldsymbol{d}) \rightarrow \boldsymbol{w}$. The procedures $\mathsf{SampleGPerturb}(q, \psi', \boldsymbol{s}, \boldsymbol{h}) \rightarrow \boldsymbol{p}$ generates a perturbation vector $\boldsymbol{p} \in \mathbb{Z}^l$ and $\mathsf{SampleD}(q, \psi', \boldsymbol{c}, \boldsymbol{d}) \rightarrow \boldsymbol{w}$ returns a sample $\boldsymbol{w} \in \mathbb{Z}^l$. Both the procedures $\mathsf{SampleD}$ and $\mathsf{SampleGPerturb}$ rely on an integer sampling subroutine $\mathsf{Sample}(t, \psi, c) \rightarrow x$ which yields a sample statistically approximating a one-dimensional discrete Gaussian with a tail-cut t, parameter ψ and center c. Our implementation supports multi-threaded execution for parallel computation of the procedure $\mathsf{SamplePre}$ inside the algorithm $\mathsf{SamplePreimage}$ and the procedure $\mathsf{SampleG}$ inside $\mathsf{GSampling}$. We experimentally evaluate multi-threaded performance of the uIPFE [1] on IBM Power9 server and provide the performance of the algorithms in section 5.

Our second contribution is construction of a strict unbounded inner product functional encryption scheme uIPFE #1. We propose a public-key uIPFE scheme based on LWE in the random oracle model with adaptively indistinguishability ($\mathsf{Adp\text{-}IND}$) security. Our scheme acheives $\mathsf{Adp\text{-}IND}$ security in ROM without using any trapdoor generation algorithm and pre-image sampling algorithm. Thus our construction not only reduces the time complexity but also makes it simple and easy to implement. In the setup phase, we choose a matrix $\mathbf{A} \in \mathbb{Z}_q^{n \times m}$ uniformly at random and sample a matrix $\mathbf{Z} \in \mathfrak{D}_{\mathbb{Z}^{m \times \ell_{\mathsf{max}}}, \sigma}$ from discrete Gaussian distribution with parameter σ and compute $\mathbf{W} = \mathbf{AZ} \in \mathbb{Z}_q^{n \times \ell_{\mathsf{max}}}$ where ℓ_{max} is the maximum length of the message vectors and secret key vectors. Then we set $\mathbf{A}$, $\mathbf{W}$ as master public key and $\mathbf{Z}$ as master secret key. We employ a full domain hash function, represented as a random oracle, $\mathcal{H} : \mathbb{N} \rightarrow \{-1, 1\}^{\ell_{\mathsf{max}}}$. The indices of the message vector $\boldsymbol{x} = (x_i)_{i \in I_{\boldsymbol{x}}} \in \mathbb{Z}_q^{|I_{\boldsymbol{x}}|}$ are used to construct a matrix $\mathbf{U} = [\boldsymbol{u}_1 \| \boldsymbol{u}_2 \| \dots \| \boldsymbol{u}_{|I_{\boldsymbol{x}}|}] \in \mathbb{Z}_q^{\ell_{\mathsf{max}} \times |I_{\boldsymbol{x}}|}$ for encryption where $\boldsymbol{u}_i = \mathcal{H}(I_{\boldsymbol{x}}[i]) \in \mathbb{Z}_q^n$ and $I_{\boldsymbol{x}}[i]$ is the i-th element of the index set $I_{\boldsymbol{x}} \subset \mathbb{N}$ of $\boldsymbol{x}$. A column vector $\boldsymbol{c}_1 = \mathbf{A}^{\top} \boldsymbol{s} + \boldsymbol{e}_1 \in \mathbb{Z}_q^m$ is set as the first component of the ciphertext where $\boldsymbol{s}$ is uniformly random column vector in $\mathbb{Z}_q^n$ and $\boldsymbol{e}_1$ is selected at random from discrete Gaussian distribution $\mathfrak{D}_{\mathbb{Z}^m, \psi}$ with ψ is Gaussian parameter. A column vector $\boldsymbol{c}_2 = \mathbf{U}^{\top} \mathbf{W}^{\top} \boldsymbol{s} + \left\lfloor \frac{q}{K} \right\rfloor \boldsymbol{x} + \boldsymbol{e}_2 \in \mathbb{Z}_q^{|I_{\boldsymbol{x}}|}$ is set as the second component of the ciphertext where K is specified as polynomially constrained in the security parameter λ. The decryption key related to secret key column vector $\boldsymbol{y} = (y_i)_{i \in I_{\boldsymbol{y}}} \in \mathbb{Z}_q^{|I_{\boldsymbol{y}}|}$ is set as $\mathsf{Dk}_{\boldsymbol{y}} = (\boldsymbol{y}^{\top}, \boldsymbol{y}^{\top} \mathbf{U}'^{\top} \mathbf{Z}^{\top}) \in \mathbb{Z}_q^{1 \times |I_{\boldsymbol{y}}|} \times \mathbb{Z}_q^{1 \times m}$ where $\mathbf{Z} \in \mathfrak{D}_{\mathbb{Z}^{m \times |I_{\boldsymbol{y}}|}, \sigma}$ with $\mathbf{U}' = [\boldsymbol{u}_1' \| \boldsymbol{u}_2' \| \dots \| \boldsymbol{u}_{|I_{\boldsymbol{y}}|}'] \in \mathbb{Z}^{\ell_{\mathsf{max}} \times |I_{\boldsymbol{y}}|}$, $\boldsymbol{u}_i' = \mathcal{H}(I_{\boldsymbol{y}}[i]) \in \mathbb{Z}_q^{\ell_{\mathsf{max}}}$, $I_{\boldsymbol{y}}[i]$ is the i-th element of the index set $I_{\boldsymbol{y}} \subset \mathbb{N}$ of $\boldsymbol{y}$. The decryptor can recover the inner product $\langle \boldsymbol{x}, \boldsymbol{y} \rangle$ from the ciphertext $\mathsf{Ct}_{\boldsymbol{x}} = (\boldsymbol{c}_1, \boldsymbol{c}_2)$ using the decryption key

$\mathsf{Dk}_{\boldsymbol{y}} = (\boldsymbol{y}^\top, \boldsymbol{y}^\top \mathbf{U}'^\top \mathbf{Z}^\top) \in \mathbb{Z}_q^{1\times|I_{\boldsymbol{y}}|} \times \mathbb{Z}_q^{1\times m}$ if $I_{\boldsymbol{x}} = I_{\boldsymbol{y}}$ by computing

$$\mu' = \boldsymbol{y}^\top \boldsymbol{c}_2 - \boldsymbol{y}^\top \mathbf{U}'^\top \mathbf{Z}^\top \boldsymbol{c}_1 \approx \langle \boldsymbol{x}, \boldsymbol{y} \rangle + \mathsf{noise}.$$

In order to reduce the noise, we effectively set the parameters. The decryption will yield accurate result if the error magnitude $\mathsf{noise} \leq \sigma\psi mY\ell_{\mathsf{max}}^2 + Y\ell_{\mathsf{max}}\psi \leq \frac{q}{2K}$ i.e. $q \geq 2K(\sigma\psi mY\ell_{\mathsf{max}}^2 + Y\ell_{\mathsf{max}}\psi)$. For further information, see Section 4 (construction) and Theorem 9 (security).

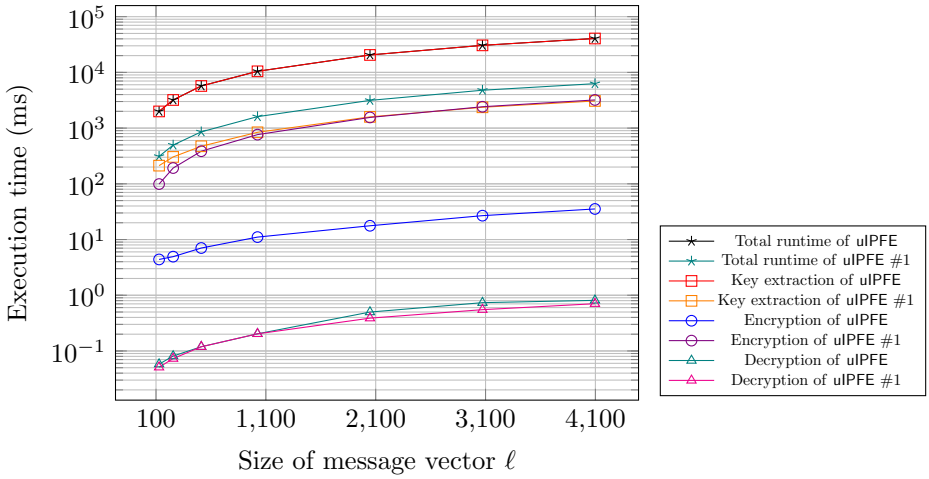

Fig. 4: Execution time comparison of uIPFE.KeyExtract, uIPFE.Encrypt, uIPFE.Decrypt of uIPFE [1] and uIPFE #1 for $\ell = 128, 256, 512, 1024, 2048, 3072, 4096$ with the choice of parameter as $n = 5$, $q \approx 2^{49}$ (uIPFE), $q \approx 2^{60}$ (uIPFE #1), $m = 490$ (uIPFE), $m = 600$ (uIPFE #1), $\ell_{\mathsf{max}} = 4096$, $X = Y = 4$, $\psi = 2.236$, $\alpha = 4.495 \times 10^{-14}$ (uIPFE), $\alpha = 1.069 \times 10^{-17}$ (uIPFE #1), $\sigma = 83.29$

Figure 4 illustrates the comparison of run times of uIPFE.KeyExtract, uIPFE.Encrypt and uIPFE.Decrypt of uIPFE and uIPFE #1 with different values of ℓ based on the particular choices of parameter. We can clearly observe that the total runtime of our uIPFE #1 is quiet less than that of uIPFE [1]. In particular, our key extraction and decryption algorithms achieve notably lower execution times, whereas the encryption algorithm in [1] exhibits marginally better performance than ours.

Organization. The subsequent portions of the paper are organized as follows. Section 2 presents a comprehensive introduction to the essential principles that are relevant to our implementation and construction. The algorithms those are needed to implement the scheme uIPFE [1] are discussed in subsections 2.4 and 3.1. Section 4 provides the construction and security analysis of our proposed scheme uIPFE #1. Section 5 is dedicated for analyzing the performance of the scheme uIPFE [1] and uIPFE #1. Section 6 serves as the concluding section of the paper.

2 Preliminaries

Notations.
We denote matrices and vectors by upper-case bold letters and lower-case bold letters respectively. We use notation λ to represent the security parameter, while $\mathsf{poly}(\lambda)$ represents a polynomial in λ. The set of integers and set of natural numbers are denoted by $\mathbb{Z}$ and $\mathbb{N}$ respectively. The field $\mathbb{Z}/q\mathbb{Z}$ is denoted by $\mathbb{Z}_q$ where q is a prime. For $m \in \mathbb{N}$, $[m]$ is used to represent the set $\{1, 2, \ldots, m\}$. For any set I, $I[i]$ denotes the i-th element of the set. For any real number r, $\mathsf{frac}(r)$ denotes the fractional part of the real number. The notation $s \xleftarrow{\$} S$ indicates that s is uniformly sampled from the set S and '$\mathbf{A}\|\mathbf{B}$' indicates the concatenation of $\mathbf{A}$ and $\mathbf{B}$.

Each matrix $\mathbf{M}$ of order $m \times n$ can be represented as $\mathbf{M} = [\boldsymbol{m}_1\|\boldsymbol{m}_2\|\ldots\|\boldsymbol{m}_n]$ where $\boldsymbol{m}_i$ denotes the i-th column of the matrix $\mathbf{M}$ and $m_{i,j}$ denotes the element of $\mathbf{M}$ at the i-th row and j-th column. The Gram-Schmidt orthogonalization of the columns of a matrix $\mathbf{T}$ is denoted as $\widetilde{\mathbf{T}}$. Transpose of a matrix $\mathbf{M}$ is denoted by $\mathbf{M}^\top$ and the identity matrix of order m is denoted by $\mathbf{I}_m$. Tensor product of two matrices $\mathbf{M}$ of order $m \times n$ and $\mathbf{N}$ of order $r \times s$ is defined by $\mathbf{M} \otimes \mathbf{N} = (m_{i,j}\mathbf{N})_{i\in[m],j\in[n]}$ where $\mathbf{M} = (m_{i,j})_{i\in[m],j\in[n]}$. The ℓ_2-norm of a vector $\boldsymbol{u} = (u_1, u_2, \ldots, u_n)$ is defined as $\|\boldsymbol{u}\| = \sqrt{u_1^2 + u_2^2 + \ldots + u_n^2}$. The max norm of a matrix $\mathbf{M}$ of order $m \times n$ is defined as $\|\mathbf{M}\| = \mathsf{max}_{1\le i\le n}\ \|\boldsymbol{m}_i\|$ where $\|\boldsymbol{m}_i\|$ denotes the ℓ_2-norm of the i-th column of $\mathbf{M}$. For a matrix $\mathbf{W} = [\boldsymbol{w}_1\|\boldsymbol{w}_2\|\ldots\|\boldsymbol{w}_m]$ of order $n \times m$, the matrix 2-norm is defined as $\|\mathbf{W}\|_2 = \mathsf{sup}_{\|\boldsymbol{x}\|=1}\|\mathbf{W}\boldsymbol{x}\|$ with the column vector $\boldsymbol{x} \in \mathbb{Z}_q^m$. We indicate a vector $\boldsymbol{u} = (u_1, u_2, \ldots, u_n) \in \mathbb{Z}^n$ as $\boldsymbol{u} = (u_i)_{i\in I_{\boldsymbol{u}}}$ where $u_i \neq 0, \forall i \in I_{\boldsymbol{u}}$, $I_{\boldsymbol{u}} \subset \mathbb{N}$ being the index set of $\boldsymbol{u}$. The inner product of any two vectors $\boldsymbol{u} = (u_i)_{i\in I_{\boldsymbol{u}}}$ and $\boldsymbol{v} = (v_j)_{j\in I_{\boldsymbol{v}}}$ is defined as

$$\langle \boldsymbol{u}, \boldsymbol{v} \rangle = \sum_{i \in I_{\boldsymbol{u}} \cap I_{\boldsymbol{v}}} u_i v_i$$

Note that $u_i = 0$ for $i \notin I_{\boldsymbol{u}}$ and $v_i = 0$ for $i \notin I_{\boldsymbol{v}}$. The notation $\mathbf{A} \approx_s \mathbf{B}$ denotes $\mathbf{A}$ and $\mathbf{B}$ are statistically close.

Definition 1 (Negligible Function) A function $\mathsf{negl} : \mathbb{N} \to \mathbb{R}$ is said to be negligible if for every $c \in \mathbb{N}$, there exists an integer $N \in \mathbb{N}$ such that $\mathsf{negl}(n) \le n^{-c}\ \forall\ n \ge N$.

Definition 2 (PPT Algorithm) An algorithm $\mathcal{A}$ is referred to as a *probabilistic polynomial time* (PPT) algorithm if it functions as a probabilistic model executed by a Turing machine within $\mathsf{poly}(\lambda)$ time.

Definition 3 (Statistically Close) Assume that two random variables, A and B, have values in a finite set C. The *statistical distance* is defined as

$$\Delta(A, B) = \frac{1}{2}\sum_{c\in C} |\mathsf{Pr}[A = c] - \mathsf{Pr}[Y = c]|$$

If $\Delta(A(\lambda), B(\lambda))$ is negligible function of λ, then two emsembles of random variables $A(\lambda)$ and $B(\lambda)$ are considered to be *statistically close* .

2.1 Strict Unbounded Inner Product Functional Encryption

A strict unbounded inner product functional enctyption (uIPFE) scheme, defined for a message space $\mathcal{X}$ and a key space $\mathcal{Y}$ with strict relation $\mathcal{R}_s : \mathcal{X} \times \mathcal{Y} \to \{0,1\}$ defined by $\mathcal{R}_s = \{(\boldsymbol{x}, \boldsymbol{y}) \in \mathcal{X} \times \mathcal{Y} | I_{\boldsymbol{x}} = I_{\boldsymbol{y}}\}$, comprises four PPT algorithms $\mathsf{uIPFE} = (\mathsf{Setup}, \mathsf{KeyExtract}, \mathsf{Encrypt}, \mathsf{Decrypt})$ that fulfill the following requirements:

– $\mathsf{Setup}(1^\lambda) \to (\mathsf{Mpk}, \mathsf{Msk})$: A trusted authority produces the master public key Mpk and the master secret key Msk on input a security parameter λ. The Mpk is made publicly available whereas the Msk is kept hidden to the authority itself.
– $\mathsf{KeyExtract}(\mathsf{Mpk}, \mathsf{Msk}, \boldsymbol{y}) \to \mathsf{Dk}_{\boldsymbol{y}}$: The algorithm is executed by the trusted authority using Mpk, Msk and a secret key vector $\boldsymbol{y} = (y_i)_{i \in I_{\boldsymbol{y}}} \in \mathcal{Y}$ as input to produce a decryption key $\mathsf{Dk}_{\boldsymbol{y}}$ for $\boldsymbol{y}$ where $I_{\boldsymbol{y}} \subseteq \mathbb{N}$ is the finite index set of $\boldsymbol{y}$.
– $\mathsf{Encrypt}(\mathsf{Mpk}, \boldsymbol{x}) \to \mathsf{Ct}_{\boldsymbol{x}}$: The encryptor executes this algorithm with Mpk as input along with a message vector $\boldsymbol{x} = (x_i)_{i \in I_{\boldsymbol{x}}} \in \mathcal{X}$, producing a ciphertext $\mathsf{Ct}_{\boldsymbol{x}}$ for $\boldsymbol{x}$ where $I \subseteq \mathbb{N}$ represents the finite index set of $\boldsymbol{x}$.
– $\mathsf{Decrypt}(\mathsf{Mpk}, \mathsf{Dk}_{\boldsymbol{y}}, \mathsf{Ct}_{\boldsymbol{x}}) \to \mu = \langle \boldsymbol{x}, \boldsymbol{y} \rangle$ or $\perp$: The decryptor uses $\mathsf{Dk}_{\boldsymbol{y}}$ to decrypt $\mathsf{Ct}_{\boldsymbol{x}}$ using Mpk, producing either the inner product $\mu = \langle \boldsymbol{x}, \boldsymbol{y} \rangle$ if $\mathcal{R}_s(\boldsymbol{x}, \boldsymbol{y}) = 1$ (i.e. $I_{\boldsymbol{x}} = I_{\boldsymbol{y}}$) or a specific symbol $\perp$ to indicate failure.

Correctness: An unbounded inner product functional encryption scheme $\mathsf{uIPFE} = (\mathsf{Setup}, \mathsf{KeyExtract}, \mathsf{Encrypt}, \mathsf{Decrypt})$ is said to be *correct* if, for all $\lambda \in \mathbb{N}$ and for all $\boldsymbol{x} = (x_i)_{i \in I_{\boldsymbol{x}}}, \boldsymbol{y} = (y_i)_{i \in I_{\boldsymbol{y}}}$ with $\mathcal{R}_s(\boldsymbol{x}, \boldsymbol{y}) = 1$, i.e. $I_{\boldsymbol{x}} = I_{\boldsymbol{y}} = I$ (say), the following holds:

$$\Pr\left[\mu = \langle \boldsymbol{x}, \boldsymbol{y} \rangle = \sum_{i \in I} x_i y_i \;\middle|\; \begin{array}{l} (\mathsf{Mpk}, \mathsf{Msk}) \leftarrow \mathsf{uIPFE.Setup}(1^\lambda); \\ \mathsf{Dk}_{\boldsymbol{y}} \leftarrow \mathsf{uIPFE.KeyExtract}(\mathsf{Mpk}, \mathsf{Msk}, \boldsymbol{y}); \\ \mathsf{Ct}_{\boldsymbol{x}} \leftarrow \mathsf{uIPFE.Encrypt}(\mathsf{Mpk}, \boldsymbol{x}); \\ \mu \leftarrow \mathsf{uIPFE.Decrypt}(\mathsf{Mpk}, \mathsf{Dk}_{\boldsymbol{y}}, \mathsf{Ct}_{\boldsymbol{x}}) \end{array}\right] = 1$$

Definition 4 (Adaptive indistinguishable ($\mathsf{Adp\text{-}IND}$)) An unbounded inner product functional encryption scheme $\mathsf{uIPFE} = (\mathsf{Setup}, \mathsf{KeyExtract}, \mathsf{Encrypt}, \mathsf{Decrypt})$ is said to be *adaptive indistinguishable* ($\mathsf{Adp\text{-}IND}$) secure if, for every PPT adversary $\mathcal{A}$ and for any $\lambda \in \mathbb{N}$, there exists a negligible function $\mathsf{negl}(\cdot)$ such that

$$\mathsf{Adv}^{\mathsf{uIPFE}}_{\mathcal{A},\mathsf{Adp\text{-}IND}}(\lambda) = |\Pr[\mathsf{Expt}^{\mathsf{uIPFE}}_{\mathcal{A},\mathsf{Adp\text{-}IND}}(\lambda) = 1] - \frac{1}{2}| \leq \mathsf{negl}(\lambda)$$

where the experiment $\mathsf{Expt}^{\mathsf{uIPFE}}_{\mathcal{A},\mathsf{Adp\text{-}IND}}(\lambda)$ between a challenger $\mathcal{C}$ and the adversary $\mathcal{A}$ is given below.

1. **Setup**: The challenger $\mathcal{C}$ generates the master public-secret key pair $(\mathsf{Mpk}, \mathsf{Msk}) \leftarrow \mathsf{uIPFE.Setup}(1^\lambda)$ and transmits Mpk to the adversary $\mathcal{A}$.
2. **Phase I**: The adversary can make polynomial number of secret key query to the oracle $\mathcal{O}_{\mathsf{KE}}(\mathsf{Mpk}, \mathsf{Msk}, \cdot)$. This oracle on query a secret key vector $\boldsymbol{y} \in \mathcal{Y}$ by $\mathcal{A}$, returns a secret key $\mathsf{Dk}_{\boldsymbol{y}}$ to $\mathcal{A}$. It takes input a key vector $\boldsymbol{y} = (y_i)_{i \in I_{\boldsymbol{y}}}$ satisfying $\langle \boldsymbol{x}^{(0)}, \boldsymbol{y} \rangle = \langle \boldsymbol{x}^{(1)}, \boldsymbol{y} \rangle$ whenever $I_{\boldsymbol{y}} = I^*$ and outputs the secret decryption key $\mathsf{Dk}_{\boldsymbol{y}} \leftarrow \mathsf{uIPFE.KeyExtract}(\mathsf{Mpk}, \mathsf{Msk}, \boldsymbol{y})$. The adversary $\mathcal{A}$ sends a pair of messages $(\boldsymbol{x}^{(0)}, \boldsymbol{x}^{(1)}) \leftarrow \mathcal{A}^{\mathcal{O}_{\mathsf{KE}}(\mathsf{Mpk},\mathsf{Msk},\cdot)}(\mathsf{Mpk})$

to $\mathcal{C}$ where $\boldsymbol{x}^{(0)} = (x_i^{(0)})_{i \in I_{\boldsymbol{x}^{(0)}}} \in \mathcal{X}, \boldsymbol{x}^{(1)} = (x_i^{(1)})_{i \in I_{\boldsymbol{x}^{(1)}}} \in \mathcal{X}$ with $I_{\boldsymbol{x}^{(0)}} = I_{\boldsymbol{x}^{(1)}} = I^*$ (say).

3. **Challenge**: The challenger $\mathcal{C}$ picks uniformly at random $\beta \xleftarrow{\$} \{0,1\}$, computes the challenge ciphertext $\mathsf{Ct}_{\boldsymbol{x}^{(\beta)}} \leftarrow \mathsf{uIPFE.Encrypt}(\mathsf{Mpk}, \boldsymbol{x}^{(\beta)})$ and sends $\mathsf{Ct}_{\boldsymbol{x}^{(\beta)}}$ to $\mathcal{A}$.
4. **Phase II**: The adversary $\mathcal{A}$ can make polynomial number of query to $\mathcal{O}_{\mathsf{KE}}(\mathsf{Mpk}, \mathsf{Msk}, \cdot)$ as in **Phase I**.
5. **Guess**: The adversary outputs a guess bit $\beta' \leftarrow \mathcal{A}^{\mathcal{O}_{\mathsf{KE}}(\mathsf{Mpk},\mathsf{Msk},\cdot)}(\mathsf{Mpk}, \mathsf{Ct}_{\boldsymbol{x}^{(\beta)}})$ after making a polynomial number of secret key query to the oracle $\mathcal{O}_{\mathsf{KE}(\mathsf{Mpk},\mathsf{Msk},\cdot)}$ and sends β' to $\mathcal{C}$.
6. The challenger $\mathcal{C}$ output 1 if $\beta = \beta'$, else 0. The experiment outputs 1 if $\mathcal{C}$ outputs 1, else 0.

2.2 Background Related to Lattices

Lattices[19, 20] Let $\mathbf{B} = [\boldsymbol{b}_1 \| \boldsymbol{b}_2 \| \dots \| \boldsymbol{b}_m] \in \mathbb{R}^{m \times m}$ be a matrix formed by m linearly independent column vectors $\boldsymbol{b}_1, \boldsymbol{b}_2, \dots, \boldsymbol{b}_m \in \mathbb{R}^m$. The lattice $\Lambda \subseteq \mathbb{R}^m$ generated by $\mathbf{B}$ which is of full rank m, is defined as:

$$\Lambda = \mathcal{L}(\mathbf{B}) = \{\boldsymbol{y} \in \mathbb{R}^m | \exists\ \boldsymbol{u} = (u_1, u_2, \dots, u_m) \in \mathbb{Z}^m, \boldsymbol{y} = \mathbf{B}\boldsymbol{u} = \sum_{i=1}^{m} u_i \boldsymbol{b}_i\}$$

The lattice Λ is an *integer lattice* when it is contained in $\mathbb{Z}^m$ instead of $\mathbb{R}^m$.
For prime modulus q, given a matrix $\mathbf{A} \in \mathbb{Z}_q^{n \times m}$ and a column vector $\boldsymbol{u} \in \mathbb{Z}_q^n$, we define

$$\begin{aligned}
\Lambda_q(\mathbf{A}) &= \{\boldsymbol{r} \in \mathbb{Z}^m \mid \exists\ \boldsymbol{v} \in \mathbb{Z}_q^n \text{ such that } \mathbf{A}^\top \boldsymbol{v} = \boldsymbol{r} \bmod q\} \\
\Lambda_q^\perp(\mathbf{A}) &= \{\boldsymbol{r} \in \mathbb{Z}^m \mid \mathbf{A}\boldsymbol{r} = \mathbf{0} \bmod q\} \\
\Lambda_q^{\boldsymbol{u}}(\mathbf{A}) &= \{\boldsymbol{r} \in \mathbb{Z}^m \mid \mathbf{A}\boldsymbol{r} = \boldsymbol{u} \bmod q\}.
\end{aligned}$$

For any matrix $\mathbf{R} \in \mathbb{Z}_q^{n \times \ell}$, we define

$$\Lambda_q^{\mathbf{R}}(\mathbf{A}) = \{\mathbf{U} \in \mathbb{Z}_q^{m \times \ell} \mid \mathbf{A}\mathbf{U} = \mathbf{R} \bmod q\}$$

Observe that if $\boldsymbol{r}' \in \Lambda_q^{\boldsymbol{u}}(\mathbf{A})$ then $\Lambda_q^{\boldsymbol{u}}(\mathbf{A}) = \Lambda_q^\perp(\mathbf{A}) + \boldsymbol{r}'$ and $\Lambda_q^{\boldsymbol{u}}(\mathbf{A})$ is shift coset to $\Lambda_q^\perp(\mathbf{A})$.

Discrete Gaussian: [20] Let $\Lambda \subset \mathbb{Z}^m$ denote an integer lattice. The *Gaussian distribution* over $\mathbb{R}^m$, represented as $\rho_{\psi,\boldsymbol{c}}$ with centre $\boldsymbol{c} \in \mathbb{R}^m$ and a positive parameter $\psi \in \mathbb{R}$, is defined as

$$\rho_{\psi,\boldsymbol{c}}(\boldsymbol{x}) = e^{-\pi \frac{||\boldsymbol{x}-\boldsymbol{c}||^2}{\psi^2}}$$

for $\boldsymbol{x} \in \Lambda$ and $\rho_{\psi,\boldsymbol{c}}(\Lambda) = \sum_{\boldsymbol{x} \in \Lambda} \rho_{\psi,\boldsymbol{c}}(\boldsymbol{x})$ which is perpetually converging.

The *discrete Gaussian distribution* over Λ, represented as $\mathfrak{D}_{\Lambda,\psi,\boldsymbol{c}}$, is characterized by parameters ψ and $\boldsymbol{c}$, is specified as

$$\mathfrak{D}_{\Lambda,\psi,\boldsymbol{c}}(\boldsymbol{y}) = \frac{\rho_{\psi,\boldsymbol{c}}(\boldsymbol{y})}{\rho_{\psi,\boldsymbol{c}}(\Lambda)} \quad \text{for} \quad \boldsymbol{y} \in \Lambda.$$

We abbreviate $\rho_{\psi,\mathbf{0}}$ and $\mathfrak{D}_{\Lambda,\psi,\mathbf{0}}$ as ρ_ψ and $\mathfrak{D}_{\Lambda,\psi}$ respectively. The distribution $\mathfrak{D}_{\Lambda,\psi,\boldsymbol{c}}$ is typically considered over the lattice $\Lambda = \Lambda_q^{\perp}(\mathbf{A})$ where $\mathbf{A} \in \mathbb{Z}_q^{n\times m}$ or over one of its cosets of the form $\Lambda = \boldsymbol{r}' + \Lambda_q^{\perp}(\mathbf{A})$ with $\boldsymbol{r}' \in \mathbb{Z}^m$.

For any matrix $\mathbf{M} \in \mathbb{Z}^{m\times n}$, we denote $\mathbf{M} \xleftarrow{\$} \mathfrak{D}_{\mathbb{Z}^{m\times n},\psi}$ with parameter ψ to represent that each of the n columns of the matrix $\mathbf{M}$ is distributed according to the distribution $\mathfrak{D}_{\mathbb{Z}^m,\psi}$.

Definition 5 (Smoothing Parameter) [21] The smoothing parameter $\eta_\epsilon(\Lambda)$ is described as the least value of s such that $\rho_{1/\psi}(\Lambda^* - \{\mathbf{0}\}) \leq \epsilon$ for a lattice Λ of dimension n and positive real number $\epsilon > 0$ where $\rho_{1/\psi}$ is a Gaussian function centered at $\mathbf{0}$ with parameter $1/\psi > 0$ defined above.

For a lattice Λ with a basis $\mathbf{B} \in \mathbb{Z}^{n\times n}$, we define $\mathsf{span}(\Lambda) = \mathsf{span}(\mathbf{B}) = \{\mathbf{B}\boldsymbol{v} : \boldsymbol{v} \in \mathbb{R}^n\}$.

Lemma 1 (Bounding Gaussian Noise) *[22] Consider a lattice $\Lambda \subset \mathbb{Z}^n$ with dimension n, centre $\boldsymbol{c} \in \mathsf{span}(\Lambda)$. Let $\epsilon \in (0,1) \cap \mathbb{R}$ and $\psi \in \eta_\epsilon(\Lambda)$ where $\eta_\epsilon(\Lambda)$ represents the smoothing parameter of Λ. Then*

$$\Pr\left[\|\boldsymbol{x} - \boldsymbol{c}\| > \psi\sqrt{n}\right] \leq \left(\frac{1+\epsilon}{1-\epsilon}\right)\frac{1}{2^n} \quad \forall \boldsymbol{x} \xleftarrow{\$} \mathfrak{D}_{\Lambda,\psi,\boldsymbol{c}} \ .$$

Specifically, the following holds for any $\epsilon \in (0, \frac{1}{2})$, any $\psi \in \eta_\epsilon(\mathbb{Z})$ and any $t \in \omega(\sqrt{\log n})$ when integers are sampled

$$\Pr\left[|x - c| > \psi t\right] \leq \mathsf{negl}(n) \quad \forall x \xleftarrow{\$} \mathfrak{D}_{\mathbb{Z},\psi,c} \ .$$

Lemma 2 (Lemma 2.4 [23]) *Let $\mathbf{Z}$ be sampled from $\mathfrak{D}_{\mathbb{Z}^{m\times m},\sigma}$, then for any matrix $\mathbf{U}$ chosen uniformly at random from $\{-1,1\}^{m\times m}$, the distribution of $\mathbf{ZU}$ is statistically close to $\mathfrak{D}_{\mathbb{Z}^{m\times m},\sigma\sqrt{m}}$.*

Definition 6 (Learning with errors (LWE)) [24]: For a prime q, a positive real value α and $\boldsymbol{s} \xleftarrow{\$} \mathbb{Z}_q^n$, we define the oracles $\mathcal{O}_{\boldsymbol{s}}$ and $\mathcal{O}_{\$}$ as follows:

- Oracle $\mathcal{O}_{\boldsymbol{s}}$: Samples fresh and independently $\boldsymbol{a} \xleftarrow{\$} \mathbb{Z}_q^n$, $x \xleftarrow{\$} \mathfrak{D}_{\mathbb{Z}_q,\alpha q}$ and generates samples $(\boldsymbol{a}, \langle \boldsymbol{a}, \boldsymbol{s}\rangle + x) \in \mathbb{Z}_q^n \times \mathbb{Z}_q$.
- Oracle $\mathcal{O}_{\$}$: Generates elements of $\mathbb{Z}_q^n \times \mathbb{Z}_q$ that are distributed uniformly at random.

Let $\mathcal{O}$ be an oracle which is either $\mathcal{O}_{\boldsymbol{s}}$ or $\mathcal{O}_{\$}$ for all invocations. The $\mathsf{LWE}_{q,\alpha,n}$ problem distinguishes with non-negligible probability whether a given input $(\boldsymbol{a}, b) \in \mathbb{Z}_q^n \times \mathbb{Z}_q$ of the oracle $\mathcal{O}$ corresponds to $\mathcal{O}_{\boldsymbol{s}}$ or $\mathcal{O}_{\$}$ by utilizing the oracle $\mathcal{O}$.

Definition 7 (Decisional Approximate Shortest Vector Problem (GapSVP_γ) [25]) Given an arbitrary basis of an n dimensional lattice $\Lambda = \mathcal{L}(\mathbf{B})$, the GapSVP_γ problem is to determine whether $\lambda_1(\Lambda) \leq 1$ or $\lambda_1(\Lambda) > \gamma(n)$ where $\lambda_1(\Lambda) = \mathsf{min}_{\boldsymbol{v} \in \Lambda \setminus \{\mathbf{0}\}} \|\boldsymbol{v}\|$ and $\gamma(n) \geq 1$ is approximation factor.

Definition 8 (Shortest Independent Vector Problem (SIVP_γ) [25]) Given a basis $\mathbf{B}$ of an n-dimensional lattice $\Lambda = \mathcal{L}(\mathbf{B})$, the SIVP_γ problem asks for a set of linearly independent lattice vectors $\{\boldsymbol{s}_1, \ldots, \boldsymbol{s}_n\} \subset \Lambda$ satisfying $\|\boldsymbol{s}_i\| \leq \gamma(n) \cdot \lambda_n(\Lambda)$ where λ_n denotes the n-th successive minimum of Λ. In other words, λ_n denotes the smallest k such that Λ contains n linearly independent vectors $\{\boldsymbol{s}_1, \ldots, \boldsymbol{s}_n\}$ with norms bounded by k and $\gamma(n) \geq 1$ denotes the approximation factor.

Lemma 3 *[24] Consider a positive integer n, a prime number $q = q(n)$ and a real number $\alpha = \alpha(n) \in (0,1)$ where $\alpha q > 2\sqrt{n}$. Suppose there is an algorithm capable of efficiently solving the $\mathsf{LWE}_{q,\alpha,n}$ problem. Then a quantum algorithm can efficiently approximate the shortest independent vectors problem $(\mathsf{SIVP})_\gamma$ and the decisional variant of the shortest vector problem $(\mathsf{GapSVP})_\gamma$ in the ℓ_2 norm for approximation factor $\gamma(n) \geq 1$. This holds in the worst-case scenario with accuracy of $\mathcal{O}(n/\alpha)$ factors.*

Lemma 4 *[13],[6]: Following result holds for the two PPT algorithms $\mathsf{TrapdoorGen}$ and $\mathsf{SamplePreimage}$ presented below for any positive integer n, any prime $q = \mathsf{poly}(n)$, any $m \geq 2n \log q$ and any $\sigma \geq L\omega(\sqrt{\log n})$ with $L = m^{2.5}$:*

- $\mathsf{TrapdoorGen}(n, q, m, \psi) \rightarrow (\mathbf{A}, \mathbf{T})$*: On input the parameters n, m, q and a real number ψ, the algorithm outputs a random matrix $\mathbf{A} \in \mathbb{Z}_q^{n \times m}$ that is statistically close to uniform along with a good basis $\mathbf{T} (\in \mathbb{Z}^{m \times m}) \subset \Lambda_q^{\perp}(\mathbf{A})$ with $\|\widetilde{\mathbf{T}}\| \leq L$. The basis $\mathbf{T}$ which satisfying $\mathbf{AT} = \mathbf{0}$ mod q serves as a trapdoor for the matrix $\mathbf{A}$. The basis $\mathbf{T}$ is "good" trapdoor basis of Λ means that the Gram-Schmidt orthogonalized column vectors of $\mathbf{T}$ are relatively short. The i-th column $\boldsymbol{t}_i$ of $\mathbf{T}$ satisfies $\|\boldsymbol{t}_i\| \leq \psi\sqrt{m}$.*
- $\mathsf{SamplePreimage}(n, q, \mathbf{A}, \mathbf{T}, \sigma, \mathbf{U}, \ell) \rightarrow \mathbf{Z}$*: This algorithm takes as input the parameters n, q, a matrix $\mathbf{A} \in \mathbb{Z}_q^{n \times m}$ and a trapdoor $\mathbf{T} \in \mathbb{Z}^{m \times m}$ generated by $\mathsf{TrapdoorGen}$, a matrix $\mathbf{U} \in \mathbb{Z}^{n \times \ell}$ chosen at random and $\sigma \geq L \cdot \omega(\sqrt{\log n})$ and generates a matrix $\mathbf{Z} \in \mathbb{Z}^{m \times \ell}$ satisfying $\mathbf{AZ} = \mathbf{U}$. Furthermore, the following two distributions are statistically close*

$$\left\{ (\mathbf{A}, \mathbf{Z}, \mathbf{U}) \;\middle|\; \begin{array}{l} (\mathbf{A}, \mathbf{T}) \leftarrow \mathsf{TrapdoorGen}(n, q, m, \psi), \\ \mathbf{Z} \leftarrow \mathsf{SamplePreimage}(n, q, \mathbf{A}, \mathbf{T}, \sigma, \mathbf{U}, \ell) \\ \text{where } \mathbf{U} \xleftarrow{\$} \mathbb{Z}^{n \times \ell} \end{array} \right\} \approx_s \left\{ (\mathbf{A}, \mathbf{Z}, \mathbf{AZ}) \;\middle|\; \begin{array}{l} \mathbf{A} \xleftarrow{\$} \mathbb{Z}_q^{n \times m}, \mathbf{Z} \xleftarrow{\$} \mathfrak{D}_{\mathbb{Z}^{m \times \ell}, \sigma} \\ \|\boldsymbol{z}_i\| \leq \sigma\sqrt{m} \;\forall i \in [\ell] \text{ where } \boldsymbol{z}_i \\ \text{denotes the } i\text{-th column of } \mathbf{Z} \end{array} \right\}$$

Definition 9 (Gadget Matrix [26]) For any modulus $q \geq 2$ with $l = \lceil \log q \rceil$, $\omega = nl$, the gadget matrix is defined as $\mathbf{G} = \mathbf{I}_n \otimes \boldsymbol{g} \in \mathbb{Z}_q^{n \times \omega}$ where $\boldsymbol{g} = (1, 2, \ldots, 2^{l-1}) \in \mathbb{Z}_q^l$ and $\mathbf{I}_n$ denotes the identity matrix of order n.
Moreover, the deterministic inversion (bit decomposition) function is defined as $\mathbf{G}^{-1} : \mathbb{Z}_q^{n \times \overline{m}} \rightarrow \{0,1\}^{\omega \times \overline{m}}$ where $\overline{m} \in \mathbb{Z}$ that decomposes each element of a matrix $\mathbf{A} \in \mathbb{Z}_q^{n \times \overline{m}}$ into its bit representation satisfying $\mathbf{G}\mathbf{G}^{-1}(\mathbf{A}) = \mathbf{A}$ for any matrix $\mathbf{A} \in \mathbb{Z}_q^{n \times \overline{m}}$.

2.3 ALS-IPFE [2]

Below, we go over a modified variation of ALS-IPFE [2] proposed by Abdalla et al. [6]. The distribution with a complex parameter configuration is used to select the secret matrix $\mathbf{Z}$ in the original ALS-IPFE of Agrawal et al. [2]. For message space $\mathcal{X} = \{0, 1, \ldots, X(\lambda) - 1\}^{\ell}$ and the decryption keys space $\mathcal{Y} = \{0, 1, \ldots, Y(\lambda) - 1\}^{\ell}$ where λ is the security parameter and ℓ represents the size of both the message vector and the decryption key vector, the security proof presented in [6] allows the selection of $\mathbf{Z}$ from discrete Gaussian distribution. Furthermore, it is assumed that the inner product value falls inside the range $\{0, 1, \ldots, K - 1\}$ i.e., $\langle \boldsymbol{x}, \boldsymbol{y} \rangle < K \ \forall \boldsymbol{x} \in \mathcal{X}$ and $\boldsymbol{y} \in \mathcal{Y}$ where $K = \ell XY$, $X = X(\lambda), Y = Y(\lambda)$.

– $\mathsf{Setup}(1^{\lambda}, 1^{\ell}) \rightarrow (\mathsf{Mpk}, \mathsf{Msk})$: On input λ and the size of input vector ℓ, the trusted authority first chooses the parameters (as given below in parameter setting) and runs the setup algorithm (see Algorithm 1) to produce the master public key $\mathsf{Mpk} = (\mathsf{pp}, \mathbf{A}_{\mathsf{ALS}}, \mathbf{D}_{\mathsf{ALS}})$ and master secret key $\mathsf{Msk} = \mathbf{Z}_{\mathsf{ALS}}$. Msk is kept secret and Mpk is made public.

Algorithm 1 Setup Algorithm

Input : λ;
Output : $\mathsf{Mpk} = (\mathsf{pp}, \mathbf{A}_{\mathsf{ALS}}, \mathbf{D}_{\mathsf{ALS}})$ with $\mathsf{pp} = (n, m, \ell, X, Y, K, q, \psi, \alpha, \sigma)$; $\mathsf{Msk} = \mathbf{Z}_{\mathsf{ALS}}$;

1: $n = n(\lambda) \in \mathbb{Z}, m = 2n \log q \in \mathbb{Z}$;
2: $X = X(\lambda) \in \mathbb{Z}, Y = Y(\lambda) \in \mathbb{Z}, K = \ell XY \in \mathbb{Z}$;
3: $q = q(\lambda) \in \mathbb{Z}$, $\psi = \psi(\lambda) \in \mathbb{R}, \alpha = \alpha(\lambda) \in \mathbb{R}, \sigma = \sigma(\lambda) \in \mathbb{R}$;
4: $\mathbf{A}_{\mathsf{ALS}} \xleftarrow{\$} \mathbb{Z}_q^{n \times m}, \mathbf{Z}_{\mathsf{ALS}} \xleftarrow{\$} \mathfrak{D}_{\mathbb{Z}^{m \times \ell}, \sigma}$;
5: $\mathbf{D}_{\mathsf{ALS}} = \mathbf{A}_{\mathsf{ALS}} \mathbf{Z}_{\mathsf{ALS}} \in \mathbb{Z}_q^{n \times \ell}$
6: **return** $(\mathsf{pp} = (n, m, \ell, X, Y, K, q, \psi, \alpha, \sigma), \mathsf{Mpk} = (\mathsf{pp}, \mathbf{A}_{\mathsf{ALS}}, \mathbf{D}_{\mathsf{ALS}}), \mathsf{Msk} = \mathbf{Z}_{\mathsf{ALS}})$

– $\mathsf{KeyExtract}(\mathsf{Mpk}, \mathsf{Msk}, \boldsymbol{y}) \rightarrow \mathsf{Dk}_{\boldsymbol{y}}$: The trusted authority takes Mpk, Msk and $\boldsymbol{y} \in \mathcal{Y}$ to generate the decryption key $\mathsf{Dk}_{\boldsymbol{y}} = (\boldsymbol{k}_1, \boldsymbol{k}_2)$ by running Algorithm 2 below. The trusted authority sends $\mathsf{Dk}_{\boldsymbol{y}}$ to the decryptor through a secure channel.

Algorithm 2 Key Extraction Algorithm

Input : $\mathsf{Mpk} = (\mathsf{pp}, \mathbf{A}_{\mathsf{ALS}}, \mathbf{D}_{\mathsf{ALS}})$; $\mathsf{Msk} = \mathbf{Z}_{\mathsf{ALS}}$; $\boldsymbol{y}$;
Output : $\mathsf{Dk}_{\boldsymbol{y}} = (\boldsymbol{k}_1, \boldsymbol{k}_2)$;

1: $\boldsymbol{k}_1 = \boldsymbol{y}^{\top}, \boldsymbol{k}_2 = \boldsymbol{y}^{\top} \mathbf{Z}_{\mathsf{ALS}}^{\top}$;
2: **return** $(\boldsymbol{k}_1, \boldsymbol{k}_2)$

– $\mathsf{Encrypt}(\mathsf{Mpk}, \boldsymbol{x}) \rightarrow \mathsf{Ct}_{\boldsymbol{x}}$: The encryptor intakes Mpk and a message vector $\boldsymbol{x} \in \mathcal{X}$ to compute the ciphertext $\mathsf{Ct}_{\boldsymbol{x}}$ using Algorithm 3 presented below and sends $\mathsf{Ct}_{\boldsymbol{x}}$ to the decryptor over a public channel.

Algorithm 3 Encryption Algorithm

Input : $\mathsf{Mpk} = (\mathsf{pp}, \mathbf{A}_{\mathsf{ALS}}, \mathbf{D}_{\mathsf{ALS}})$; $\boldsymbol{x}$;
Output : $\mathsf{Ct}_{\boldsymbol{x}} = (\boldsymbol{c}_1, \boldsymbol{c}_2)$;

1: $\boldsymbol{s} \xleftarrow{\$} \mathbb{Z}_q^n$, $\boldsymbol{e}_1 \xleftarrow{\$} \mathfrak{D}_{\mathbb{Z}^m,\psi}$, $\boldsymbol{e}_2 \xleftarrow{\$} \mathfrak{D}_{\mathbb{Z}^\ell,\psi}$;
2: $\boldsymbol{c}_1 = \mathbf{A}_{\mathsf{ALS}}^\top \boldsymbol{s} + \boldsymbol{e}_1 \in \mathbb{Z}_q^m$;
3: $\boldsymbol{c}_2 = \mathbf{D}_{\mathsf{ALS}}^\top \boldsymbol{s} + \lfloor \frac{q}{K} \rfloor \boldsymbol{x} + \boldsymbol{e}_2 \in \mathbb{Z}_q^\ell$;
4: **return** $(\boldsymbol{c}_1, \boldsymbol{c}_2)$

– $\mathsf{Decrypt}(\mathsf{Mpk}, \mathsf{Dk}_{\boldsymbol{y}}, \mathsf{Ct}_{\boldsymbol{x}}) \to \mu \vee \perp$: On receiving the ciphertext $\mathsf{Ct}_{\boldsymbol{x}}$ from an encryptor, a decryptor uses Mpk and its own secret key $\mathsf{Dk}_{\boldsymbol{y}}$ corresponding to its secret key vector $\boldsymbol{y} \in \mathcal{Y}$ and executes the decryption algorithm as displayed in Algorithm 4.

Algorithm 4 Decryption Algorithm

Input : $\mathsf{Mpk} = (\mathsf{pp}, \mathbf{A}_{\mathsf{ALS}}, \mathbf{D}_{\mathsf{ALS}})$; $\mathsf{Dk}_{\boldsymbol{y}} = (\boldsymbol{k}_1, \boldsymbol{k}_2)$; $\mathsf{Ct}_{\boldsymbol{x}} = (\boldsymbol{c}_1, \boldsymbol{c}_2)$;
Output : μ;

1: $\mu' = \langle \boldsymbol{k}_1^\top, \boldsymbol{c}_2 \rangle - \langle \boldsymbol{k}_2^\top, \boldsymbol{c}_1 \rangle$;
2: **for** $\mu = 0$ to $K - 1$ **do**
3: **if** $(|\lfloor \frac{q}{K} \rfloor \cdot \mu - \mu'| < (\sigma\psi m\ell Y + Y\ell\psi))$ **then**
4: **break**;
5: **end if**
6: **end for**
7: **return** μ

Lemma 5 (Correctness of ALS-IPFE [2],[6]) *If* $\psi, \sigma > \omega(\sqrt{\log n})$ *and* $q > 2\ell K\sqrt{\ell}Y\omega(\log^2 n)$, *then the scheme* ALS-IPFE *is correct.*

Parameter Settings

The parameter configuration for the ALS-IPFE which relies on the hardness of $\mathsf{LWE}_{q,\alpha,n}$ is presented as $m \geq 2n \log q$, $\psi \geq \omega(\sqrt{\log n}), \sigma \geq \omega(\sqrt{\log n})$, $\alpha q \geq 2\sqrt{n}$, $\alpha \in (0,1) \cap \mathbb{R}$, $\alpha \leq \frac{\psi}{2C_1 q(\sqrt{m}+\sqrt{n}+\sqrt{\ell})}$ where C_1 is a constant, $q \geq 2\ell K\sqrt{\ell}Y\omega(\log^2 n)$ with $K = \ell XY$ and αq represents the standard deviation of the noise distribution in the underlying LWE problem.

Lemma 6 (Security [2],[6]) *Let* $q \geq 2\ell K\sqrt{\ell}Y\omega(\log^2 n)$ *is prime,* $m \geq 2n \log q$, $\psi \geq \omega(\sqrt{\log n}), \sigma \geq \omega(\sqrt{\log n})$, *and* $\alpha \leq \frac{\psi}{2C_1 q(\sqrt{m}+\sqrt{n}+\sqrt{\ell})}$ *where* C_1 *is a constant. Then the scheme* ALS-IPFE *is considered* Adp-IND *(see Definition 4) secure based on the assumption of the hardness of* $\mathsf{LWE}_{q,\alpha,n}$ *(see Definition 6).*

2.4 uIPFE Scheme of Dutta et al.[1]

In this section we discuss the *unbounded inner product functional encryption* scheme uIPFE of Dutta et al. [1] below. The co-ordinates of the message vectors are from $\mathcal{X} = \{0, 1, \ldots, X(\lambda)-1\}$ and those of the secret key vectors are from $\mathcal{Y} = \{0, 1, \ldots, Y(\lambda)-1\}$. The length of the message vector and the decryption key vector are less than ℓ_{max}. The inner product of a message vector $\boldsymbol{x} \in \mathcal{X}^{\ell_{\mathsf{max}}}$ and the associated decryption key vector $\boldsymbol{y} \in \mathcal{Y}^{\ell_{\mathsf{max}}}$ lies between $\{0, 1, \ldots, K-1\}$ i.e. $\langle \boldsymbol{x}, \boldsymbol{y} \rangle < K$, where $K = XY\ell_{\mathsf{max}}$.

– $\mathsf{uIPFE.Setup}(1^\lambda) \rightarrow (\mathsf{Mpk}, \mathsf{Msk})$: On input a security parameter λ, a trusted authority executes the steps as described in Algorithm 5 below choosing the parameters as given in parameter settings below and generates master public-key Mpk and master secret-key Msk. This algorithm makes use of the PPT algorithm $\mathsf{TrapdoorGen}(n, q, m, \psi) \rightarrow (\mathbf{A}, \mathbf{T})$ which generates a random matrix $\mathbf{A} \in \mathbb{Z}_q^{n \times m}$ along with its trapdoor basis $\mathbf{T} (\in \mathbb{Z}^{m \times m}) \subseteq \Lambda_q^{\perp}(\mathbf{A})$ such that $\mathbf{AT} = \mathbf{0} \pmod q$, $\|\widetilde{\mathbf{T}}\| \leq L$ with $L = m^{2.5}$, $\|\boldsymbol{t}_i\| \leq \psi\sqrt{m}$, $\boldsymbol{t}_i$ being the i-th column of $\mathbf{T}$ (see Lemma 4). Mpk is published, Msk is kept secret to the trusted authority.

Algorithm 5 $\mathsf{uIPFE.Setup}(1^\lambda) \rightarrow (\mathsf{Mpk}, \mathsf{Msk})$

Input : λ;
Output : $\mathsf{Mpk} = (\mathsf{pp}, \mathbf{A})$ with $\mathsf{pp} = (n, q, m, \ell_{\mathsf{max}}, X, Y, K, \psi, \alpha, \sigma, \mathcal{H})$; $\mathsf{Msk} = \mathbf{T}$;

1: $n = n(\lambda) \in \mathbb{Z}, q = q(\lambda) \in \mathbb{Z}$, $m = 2n \log_2 q \in \mathbb{Z}$;
2: $\ell_{\mathsf{max}} = \ell_{\mathsf{max}}(\lambda) \in \mathbb{Z}, X = X(\lambda) \in \mathbb{Z}, Y = Y(\lambda) \in \mathbb{Z}$, $K = XY\ell_{\mathsf{max}} \in \mathbb{Z}$;
3: $\psi = \psi(\lambda) \in \mathbb{R}$, $\alpha = \alpha(\lambda) \in \mathbb{R}$, $\sigma = \sigma(\lambda) \in \mathbb{R}$;
4: Choose a hash function $\mathcal{H} : \mathbb{N} \rightarrow \mathbb{Z}_q^n$ which is modelled as random oracle;
5: $(\mathbf{A}, \mathbf{T}) \leftarrow \mathsf{TrapdoorGen}(n, q, m, \psi)$; $\mathbf{A} \in \mathbb{Z}_q^{n \times m}$, $\mathbf{T} \in \mathbb{Z}^{m \times m}$;
6: **return** $(\mathsf{pp} = (n, q, m, \ell_{\mathsf{max}}, X, Y, K, \psi, \alpha, \sigma, \mathcal{H}), \mathsf{Mpk} = (\mathsf{pp}, \mathbf{A}), \mathsf{Msk} = \mathbf{T})$

– $\mathsf{uIPFE.KeyExtract}(\mathsf{Mpk}, \mathsf{Msk}, \boldsymbol{y}) \rightarrow \mathsf{Dk}_{\boldsymbol{y}}$: On input the Mpk, Msk and a secret key vector $\boldsymbol{y} = (y_i)_{i \in I_{\boldsymbol{y}}} \in \mathcal{Y}^{\ell'}$ received from a user, the trusted authority performs the steps as shown in Algorithm 6 below where $\ell' = |I_{\boldsymbol{y}}|$, $I_{\boldsymbol{y}}$ being the index set of $\boldsymbol{y}$. This algorithm makes use of the PPT algorithm $\mathsf{SamplePreimage}(n, q, \mathbf{A}, \mathbf{T}, \sigma, \mathbf{U}', \ell') \rightarrow \mathbf{Z}$ which generates a matrix $\mathbf{Z} \in \mathcal{D}_{\mathbb{Z}^{n \times \ell'}, \sigma}$ satisfying $\mathbf{AZ} = \mathbf{U}' \pmod q$.

Algorithm 6 $\mathsf{uIPFE.KeyExtract}(\mathsf{Mpk}, \mathsf{Msk}, \boldsymbol{y}) \rightarrow \mathsf{Dk}_{\boldsymbol{y}}$

Input : $\mathsf{Mpk} = (\mathsf{pp}, \mathbf{A})$ with $\mathsf{pp} = (n, q, m, \ell_{\mathsf{max}}, X, Y, K, \psi, \alpha, \sigma, \mathcal{H})$; $\mathsf{Msk} = \mathbf{T}; \boldsymbol{y} = (y_i)_{i \in I_{\boldsymbol{y}}}$ with $|I_{\boldsymbol{y}}| = \ell' \leq \ell_{max}$;
Output : $\mathsf{Dk}_{\boldsymbol{y}} = (\boldsymbol{y}^\top, \boldsymbol{y}^\top \mathbf{Z}^\top)$;
1: **for** $i = 1$ to ℓ' **do**
2: $\quad \boldsymbol{u}'_i \leftarrow \mathcal{H}(I_{\boldsymbol{y}}[i])$;
3: **end for**
4: Set $\mathbf{U}' = [\boldsymbol{u}'_1 \| \boldsymbol{u}'_2 \| \ldots \| \boldsymbol{u}'_{\ell'}] \in \mathbb{Z}_q^{n \times \ell'}$;
5: $\mathsf{SamplePreimage}(n, q, \mathbf{A}, \mathbf{T}, \sigma, \mathbf{U}', \ell') \rightarrow \mathbf{Z} \in \mathfrak{D}_{\mathbb{Z}^{m \times \ell'}, \sigma}$ satisfying $\mathbf{AZ} = \mathbf{U}' (\mathrm{mod}\ q)$;
6: **return** $\mathsf{Dk}_{\boldsymbol{y}} = (\boldsymbol{y}^\top, \boldsymbol{y}^\top \mathbf{Z}^\top)$

– $\mathsf{uIPFE.Encrypt}(\mathsf{Mpk}, \boldsymbol{x}) \rightarrow \mathsf{Ct}_{\boldsymbol{x}}$: The encryptor takes as input the Mpk and a message vector $\boldsymbol{x} = (x_i)_{i \in I_{\boldsymbol{x}}} \in \mathcal{X}^\ell$ where $\ell = |I_{\boldsymbol{x}}|$, $I_{\boldsymbol{x}} \subseteq \mathbb{N}$ is the non-empty index set of $\boldsymbol{x}$ and generates the ciphertext $\mathsf{Ct}_{\boldsymbol{x}}$ following the steps presented in Algorithm 7 below and sends $\mathsf{Ct}_{\boldsymbol{x}}$ over a public channel to the decryptor.

Algorithm 7 $\mathsf{uIPFE.Encrypt}(\mathsf{Mpk}, \boldsymbol{x}) \rightarrow \mathsf{Ct}_{\boldsymbol{x}}$

Input : $\mathsf{Mpk} = (\mathsf{pp}, \mathbf{A})$ with $\mathsf{pp} = (n, q, m, \ell_{\mathsf{max}}, X, Y, K, \psi, \alpha, \sigma, \mathcal{H})$; $\boldsymbol{x} = (x_i)_{i \in I_{\boldsymbol{x}}}$ with $|I_{\boldsymbol{x}}| = \ell \leq \ell_{\mathsf{max}}$;
Output : $\mathsf{Ct}_{\boldsymbol{x}} = (\boldsymbol{c}_1, \boldsymbol{c}_2)$;
1: **for** $i = 1$ to ℓ **do**
2: $\quad \boldsymbol{u}_i \leftarrow \mathcal{H}(I_{\boldsymbol{x}}[i])$;
3: **end for**
4: Set $\mathbf{U} = [\boldsymbol{u}_1 \| \boldsymbol{u}_2 \| \ldots \| \boldsymbol{u}_\ell] \in \mathbb{Z}_q^{n \times \ell}$;
5: $\boldsymbol{s} \xleftarrow{\$} \mathbb{Z}_q^n, \boldsymbol{e}_1 \xleftarrow{\$} \mathfrak{D}_{\mathbb{Z}^m, \psi}$, $\boldsymbol{e}_2 \xleftarrow{\$} \mathfrak{D}_{\mathbb{Z}^\ell, \psi}$;
6: $\boldsymbol{c}_1 = \mathbf{A}^\top \boldsymbol{s} + \boldsymbol{e}_1 \in \mathbb{Z}_q^m$;
7: $\boldsymbol{c}_2 = \mathbf{U}^\top \boldsymbol{s} + \lfloor \frac{q}{K} \rfloor \boldsymbol{x} + \boldsymbol{e}_2 \in \mathbb{Z}_q^\ell$;
8: **return** $\mathsf{Ct}_{\boldsymbol{x}} = (\boldsymbol{c}_1, \boldsymbol{c}_2)$

– $\mathsf{uIPFE.Decrypt}(\mathsf{Mpk}, \mathsf{Dk}_{\boldsymbol{y}}, \mathsf{Ct}_{\boldsymbol{x}}) \rightarrow \mu \vee \perp$: On input the Mpk, a decryptor uses its decryption key $\mathsf{Dk}_{\boldsymbol{y}}$ corresponding to its secret key vector $\boldsymbol{y} \in \mathcal{Y}^{\ell'}$ and a ciphertext $\mathsf{Ct}_{\boldsymbol{x}}$, the decryptor executes the steps as in Algorithm 8 below.

Algorithm 8 $\mathsf{uIPFE.Decrypt}(\mathsf{Mpk}, \mathsf{Dk}_{\boldsymbol{y}}, \mathsf{Ct}_{\boldsymbol{x}}) \rightarrow \mu$ or $\perp$

Input : $\mathsf{Mpk} = (\mathsf{pp}, \mathbf{A})$ with $\mathsf{pp} = (n, q, m, \ell_{\mathsf{max}}, X, Y, K, \psi, \alpha, \sigma, \mathcal{H})$; $\mathsf{Dk}_{\boldsymbol{y}} = (\boldsymbol{y}^\top, \boldsymbol{y}^\top \mathbf{Z}^\top)$; $\mathsf{Ct}_{\boldsymbol{x}} = (\boldsymbol{c}_1, \boldsymbol{c}_2)$;
Output : μ;

$\mu' = \boldsymbol{y}^\top \boldsymbol{c}_2 - \boldsymbol{y}^\top \mathbf{Z}^\top \boldsymbol{c}_1$;
for ($\mu = 0$ to $K - 1$) **do**
  if ($|\lfloor \frac{q}{K} \rfloor \mu - \mu'| < \sigma\psi m\ell_{\mathsf{max}} Y + Y\ell_{\mathsf{max}}\psi$) **then**
    break;
  end if
end for
return μ

Parameter Settings

The parameter setting of the uIPFE of Dutta et al. [1] which is based on $\mathsf{LWE}_{q,\alpha,n}$ is presented as $m \geq 2n \log q$, $L = m^{2.5}, \sigma \geq L\omega(\sqrt{\log n})$, $\psi \geq \omega(\sqrt{\log n})$, $\alpha q \geq 2\sqrt{n}$, $\alpha \in (0,1) \cap \mathbb{R}$, $\alpha \leq \frac{\psi}{2C_1 q(\sqrt{m}+\sqrt{n}+\sqrt{\ell_{\mathsf{max}}})}$ where C_1 is a constant, $q \geq 2K(\sigma\psi m\ell_{\mathsf{max}} Y + Y\ell_{\mathsf{max}}\psi)$ where $K = \ell_{\mathsf{max}} XY$ and αq represents the standard deviation of LWE's noise distribution.

Lemma 7 (Correctness of uIPFE of Dutta et al. [1], [11]) *If $m \geq 2n \log q$, $\psi \geq \omega(\sqrt{\log n})$, $L = m^{2.5}$, $\sigma \geq L\omega(\sqrt{\log n})$ and $q \geq 2K(\sigma\psi m\ell_{max} Y + Y\ell_{max}\psi)$, then the above scheme uIPFE is correct.*

Lemma 8 (Security [1], [11]) *The scheme uIPFE discussed above is considered Adp-IND secure (see Definition 4) in random oracle model (ROM) under the hardness of $\mathsf{LWE}_{q,\alpha,n}$ problem (see Definition 6) where $m \geq 2n \log q, L = m^{2.5}, \sigma \geq L\omega(\sqrt{\log n}), \psi \geq \omega(\sqrt{\log n}), \alpha q \geq 2\sqrt{n}, \alpha \leq \frac{\psi}{2C_1 q(\sqrt{m}+\sqrt{n}+\sqrt{\ell_{\mathsf{max}}})}$ and $q \geq 2K(\sigma\psi m\ell_{\mathsf{max}} Y + Y\ell_{\mathsf{max}}\psi)$.*

3 Implementation of the uIPFE of Dutta et al. [1]

3.1 Implementation Overview

The scheme uIPFE = (uIPFE.Setup, uIPFE.Extract, uIPFE.Encrypt, uIPFE.Decrypt) is a trapdoor-based scheme as the setup algorithm uIPFE.Setup (Algorithm 5) invokes the trapdoor generation algorithm TrapdoorGen and the key extraction algorithm uIPFE.KeyExtract (Algorithm 6) calls the Gaussian preimage sampling algorithm SamplePreimage. Thus the largest part of the running time of uIPFE depends on the execution time of TrapdoorGen and SamplePreimage. We provide below the details of an efficient Gaussian sampler RandomSample which serves as a building block of both the algorithms TrapdoorGen and SamplePreimage in our implementation. In both algorithms uIPFE.KeyExtract (see Algorithm 6) and uIPFE.Encrypt (Algorithm 7), we instantiate SHAKE-256 as a hash function, which relies on the Keccak permutation.

- The procedure RandomSample described in Algorithm 9 samples a matrix $\mathbf{R}$ such that each entry of the matrix $\mathbf{R}$ is sampled according to the Gaussian distribution. It takes parameter ψ of the discrete Gaussian distribution and dimensions ℓ, m of the matrix $\mathbf{R}$ as input, executes the procedure Sample described in Algorithm 10 as a subroutine to sample each entry of $\mathbf{R}$ and finally outputs the matrix $\mathbf{R}$ of order $\ell \times m$ from discrete Gaussian distribution $\mathfrak{D}_{\mathbb{Z}^{\ell\times m},\psi}$.

Algorithm 9 RandomSample$(\ell, m, \psi) \rightarrow \mathbf{R}$

Input : ℓ, m, ψ;
Output : $\mathbf{R} = (r_{i,j})_{i\in[\ell],j\in[m]} \in \mathbb{Z}^{\ell\times m}$;
1: **for** $(i = 1$ to $\ell)$ **do**
2: **for** $(j = 1$ to $m)$ **do**
3: $r_{i,j} \leftarrow$ Sample$(1.0, \psi, 0.0)$;
4: **end for**
5: **end for**
6: **return R**

Algorithm 10 Sample$(t, \psi, c) \rightarrow x$

Input : $t \in \mathsf{FP}_{m_1}, \psi \in \mathsf{FP}_{m_1}, c \in \mathsf{FP}_{m_1}$;
Output : $x \in \mathbb{Z}$;
1: $h = -\pi/\psi^2 \in \mathsf{FP}_{m_1}$;
2: **do**
3: $x \xleftarrow{\$} [c - t\psi, c + t\psi] \cap \mathbb{Z}$;
4: $r \xleftarrow{\$} [0, 1) \subset \mathsf{FP}_{m_1}$
5: $p = e^{h(x-c)^2} \in \mathsf{FP}_{m_1}$
6: **while** $(r < p)$;
7: **return** x

- The subprocedure Sample adapts the rejection sampling algorithm for Discrete Gaussian on $\mathbb{Z}$ of [17] that takes tail-cut t, parameter ψ of discrete Gaussian distribution and centre c as input and outputs an integer x sampled from $\mathfrak{D}_{\mathbb{Z},\psi}$ where $t \in \mathsf{FP}_{m_1}$, $\psi \in \mathsf{FP}_{m_1}$ and $c(= 0) \in \mathsf{FP}_{m_1}$. Here, FP_{m_1} denotes the set of floating point numbers with mantissa size m_1 bits. A floating point number $f \in \mathsf{FP}_{m_1}$ can be expressed as a triplet $f = (f_1, f_2, f_3)$ which represents the real number $f = (-1)^{f_1} \cdot 2^{f_2-m_1} \cdot f_3 \in \mathbb{R}$ where $f_1 \in \{0, 1\}$, $f_2 \in \mathbb{Z}$ and $f_3 \in \mathbb{N}_{2^{m_1}-1}$. Floating-point numbers have explicitly determined precisions based on their data types such as 32-bit single-precision and 64-bit double-precision which offer 23-bit and 52-bit mantissa respectively. We have used Gaussian sampler that samples over $\mathbb{Z}$ as this sampler can produce samples from $\mathbb{Z}$ in a constant time regardless of the output value, center or Gaussian parameter. The running time of this sampler of discrete Gaussian over $\mathbb{Z}$ depend on neither the input values nor on the output values. One can replace this sampler with any other sampler if required.
- The trapdoor generation algorithm TrapdoorGen presented in Algorithm 11 generates a matrix $\mathbf{A} \in \mathbb{Z}_q^{n\times m}$ and its corresponding trapdoor $\mathbf{T} \in \mathbb{Z}^{m\times m}$ on input n, q, m and parameter ψ. It sets $l, \omega, \overline{m}$ as shown in the Algorithm 11, samples a matrix $\overline{\mathbf{A}}$ randomly from $\mathbb{Z}_q^{n\times\overline{m}}$. We set the tag $\mathbf{H} \in \mathbb{Z}_q^{n\times n}$ as an identity matrix of order n. The procedure RandomSample is invoked to generate a matrix $\mathbf{T}_{\mathbf{A}} \in \mathbb{Z}^{\overline{m}\times\omega}$ sampled from the discrete Gaussian distribution $\mathfrak{D}_{\mathbb{Z}^{\overline{m}\times\omega},\psi}$. It then generates the gadget matrix $\mathbf{G} \in \mathbb{Z}_q^{n\times\omega}$ by computing tensor product of identity matrix $\mathbf{I}_n$ of order n and the gadget vector $\boldsymbol{g} = (1, 2, 2^2, \ldots, 2^{l-1}) \in \mathbb{Z}_q^l$. Finally, it outputs the matrix

$\mathbf{A} = [\overline{\mathbf{A}} \| \mathbf{HG} - \overline{\mathbf{A}}\mathbf{T_A}] \in \mathbb{Z}_q^{n \times m}$ and its trapdoor $\mathbf{T} = \begin{bmatrix} \mathbf{I}_{\overline{m}} + \mathbf{T_A}\mathbf{W} & \mathbf{T_A}\mathbf{S} \\ \mathbf{W} & \mathbf{S} \end{bmatrix} \in \mathbb{Z}^{m \times m}$ where $\mathbf{W} = -\mathbf{G}^{-1}(\mathbf{A}) \in \mathbb{Z}^{\omega \times \overline{m}}$ and $\mathbf{S} \in \mathbb{Z}^{\omega \times \omega}$ is a basis for $\Lambda^{\perp}(\mathbf{G})$ as shown in the Algorithm 11. This procedure need to run a procedure GetBit described in Algorithm 12 for generating the matrix $\mathbf{S} \in \mathbb{Z}^{\omega \times \omega}$.

Algorithm 11 TrapdoorGen $(n, q, m, \psi) \rightarrow (\mathbf{A}, \mathbf{T})$

Input : $n \in \mathbb{Z}, m \in \mathbb{Z}, q \in \mathbb{Z}, \psi \in \mathbb{R}$;
Output : $\mathbf{A} \in \mathbb{Z}_q^{n \times m}, \mathbf{T} \in \mathbb{Z}^{m \times m}$;

1: $l = \lceil \log_2(q) \rceil \in \mathbb{Z}$, $\omega = n.l \in \mathbb{Z}$, $\overline{m} = (m - \omega) \in \mathbb{Z}$;
2: $\overline{\mathbf{A}} \xleftarrow{\$} \mathbb{Z}_q^{n \times \overline{m}}, \mathbf{H} = \mathbf{I}_n \in \mathbb{Z}_q^{n \times n}$;
3: $\mathbf{T_A} \leftarrow \mathsf{RandomSample}(\overline{m}, \omega, \psi)$; $\mathbf{T_A} \in \mathbb{Z}^{\overline{m} \times \omega}$;
4: $\boldsymbol{g} = (1, 2, 2^2, \ldots, 2^{l-1}) \in \mathbb{Z}_q^l$;
5: $\mathbf{G} = \mathbf{I}_n \otimes \boldsymbol{g} \in \mathbb{Z}_q^{n \times \omega}$;
6: $\mathbf{A} = [\overline{\mathbf{A}} \| \mathbf{H} \cdot \mathbf{G} - \overline{\mathbf{A}} \cdot \mathbf{T_A}] \in \mathbb{Z}_q^{m \times n}$;
7: **for** $(i = 1$ to $l)$ **do**
8: $\quad q_{(i)} = \mathsf{GetBit}(q, i)$;
9: **end for**
10: Set $\mathbf{S}_l = \begin{bmatrix} 2 & 0 & 0 \ldots & 0 & q_{(1)} \\ -1 & 2 & 0 \ldots & 0 & q_{(2)} \\ 0 & -1 & 2 \ldots & 0 & q_{(3)} \\ \cdot & \cdot & \cdot \ldots & \cdot & \cdot \\ 0 & 0 & 0 \ldots & -1 & q_{(l)} \end{bmatrix} \in \mathbb{Z}^{l \times l}$; $\mathbf{S} = \begin{bmatrix} \mathbf{S}_l & \mathbf{O}_l & \ldots & \mathbf{O}_l \\ \mathbf{O}_l & \mathbf{S}_l & \ldots & \mathbf{O}_l \\ \cdot & \cdot & \ldots & \cdot \\ \mathbf{O}_l & \mathbf{O}_l & \ldots & \mathbf{S}_l \end{bmatrix} \in \mathbb{Z}^{\omega \times \omega}$;
11: $\mathbf{W} = -\mathbf{G}^{-1}(\overline{\mathbf{A}}) \in \mathbb{Z}^{\omega \times \overline{m}}$;
12: $\mathbf{T} = \begin{bmatrix} \mathbf{I}_{\overline{m}} & \mathbf{T_A} \\ \mathbf{O} & \mathbf{I}_\omega \end{bmatrix} \cdot \begin{bmatrix} \mathbf{I}_{\overline{m}} & \mathbf{O} \\ \mathbf{W} & \mathbf{S} \end{bmatrix} = \begin{bmatrix} \mathbf{I}_{\overline{m}} + \mathbf{T_A}\mathbf{W} & \mathbf{T_A}\mathbf{S} \\ \mathbf{W} & \mathbf{S} \end{bmatrix} \in \mathbb{Z}^{m \times m}$;
13: **return** $(\mathbf{A}, \mathbf{T})$

- The procedure GetBit takes two integers q and i and returns the i-th bit $q_{(i)}$ of the given integer q. We denote 'a RightShift b' to represent the right-shifting of an integer 'a', 'b' bits which is equivalent to dividing a with 2^b and 'a And b' represents the bit-by-bit and operator.

Algorithm 12 GetBit$(q, i) \rightarrow q_{(i)}$

Input : $q \in \mathbb{Z}, i \in \mathbb{Z}$;
Output : $q_{(i)} \in \mathbb{Z}$;
1: $q_{(i)} \leftarrow ((q \text{ RightShift } i) \text{ And } 1)$;
2: **return** $q_{(i)}$

- The Gaussian preimage sampling algorithm SamplePreimage provided in algorithm 13 takes as input n, q, a matrix $\mathbf{A} \in \mathbb{Z}_q^{n \times m}$, its trapdoor $\mathbf{T} \in \mathbb{Z}^{m \times m}$, Gaussian parameter $\sigma \in \mathbb{R}$, a matrix $\mathbf{U} \in \mathbb{Z}_q^{n \times \ell'}$ and number of columns ℓ' of $\mathbf{U}$ and generates a matrix $\mathbf{Z} \in \mathbb{Z}^{m \times \ell'}$ such that $\mathbf{AZ} = \mathsf{U}$.

Algorithm 13 SamplePreimage $(n, q, \mathbf{A}, \mathbf{T}, \sigma, \mathbf{U}, \ell') \rightarrow \mathbf{Z}$

Input : $n \in \mathbb{Z}, q \in \mathbb{Z}, \mathbf{A} \in \mathbb{Z}_q^{n \times m}, \mathbf{T} \in \mathbb{Z}^{m \times m}, \sigma \in \mathbb{R}, \mathbf{U} \in \mathbb{Z}_q^{n \times \ell}, \ell' \in \mathbb{Z}$;
Output : $\mathbf{Z} \in \mathbb{Z}^{m \times \ell'}$
1: $l = \log_2 q$; Choose $s \in \mathbb{R}$ where s is discrete Gaussian parameter such that $s \geq 10\sqrt{\log_2(2l(1+\frac{1}{\epsilon}))/\pi}$, $0 < \epsilon \leq \frac{1}{2}$;
2: Parse $\mathbf{T} = \begin{bmatrix} \mathbf{T}_1 & \mathbf{T}_2 \\ \mathbf{T}_3 & \mathbf{T}_4 \end{bmatrix}$ as $\begin{bmatrix} \mathbf{I}_{\overline{m}} + \mathbf{T_A W} & \mathbf{T_A S} \\ \mathbf{W} & \mathbf{S} \end{bmatrix}$ where $\mathbf{T}_1 = \mathbf{I}_{\overline{m}} + \mathbf{T_A W} \in \mathbb{Z}^{\overline{m} \times \overline{m}}$, $\mathbf{T}_2 = \mathbf{T_A S} \in \mathbb{Z}^{\overline{m} \times \omega}$, $\mathbf{T}_3 = \mathbf{W} \in \mathbb{Z}^{\omega \times \overline{m}}$ and $\mathbf{T}_4 = \mathbf{S} \in \mathbb{Z}^{\omega \times \omega}$;
3: $\mathbf{T_A} = \mathbf{T}_2 \cdot \mathbf{T}_4^{-1} \in \mathbb{Z}^{\overline{m} \times \omega}$;
4: **for** $(i = 1$ to $\ell')$ **do**
5: $\quad \boldsymbol{z}_i \leftarrow \mathsf{SamplePre}(n, q, \mathbf{A}, \mathbf{T_A}, s, \sigma, \boldsymbol{u}_i)$;$\boldsymbol{z}_i \in \mathbb{Z}^n$; $\boldsymbol{u}_i$ is the i-th column of $\mathbf{U}$;
6: **end for**// This for loop can be executed parallelly;
7: Set $\mathbf{Z} = [\boldsymbol{z}_1 \| \boldsymbol{z}_2 \| \ldots \| \boldsymbol{z}_{\ell'}]$; $\mathbf{Z} \in \mathfrak{D}_{\mathbb{Z}^{m \times \ell'}, \sigma}$;
8: **return Z**

- The algorithm employs the procedure SamplePre for sampling the columns $\{\boldsymbol{z}_i\}_{1 \leq i \leq \ell'}$ of the matrix $\mathbf{Z}$ which in turn executes SamplePerturbation and GSampling described in Algorithm 15 and Algorithm 16 respectively as subroutines.
- The procedure SamplePerturbation takes matrix $\mathbf{T_A} \in \mathbb{Z}^{\overline{m} \times \omega}$ along with the parameters σ and s as input and runs the procedures RandomSample (Algorithm 9) and Sample (Algorithm 10) as subroutines to output a perturbation vector $\boldsymbol{p} \in \mathbb{Z}^m$. The vector $\boldsymbol{p}$ is drawn from a centered discrete Gaussian distribution over $\mathbb{Z}^m$ with covariance $\mathbf{S} \in \mathbb{R}^{\overline{m} \times \overline{m}}$ as shown in Algorithm 15. Following the method proposed by Bert et al. [15], the covariance matrix is decomposed into blocks of varying dimensions at each iteration, with the center updated correspondingly to iteratively sample the individual perturbation components $p_i \in \mathbb{Z}$.
- On the other hand, the procedure GSampling on input n, a vector $\boldsymbol{v} \in \mathbb{Z}^n$, modulus q and α as variance invokes the procedures SampleG and GetBit described in Algorithm

Algorithm 14 SamplePre $(n, q, \mathbf{A}, \mathbf{T_A}, s, \sigma, \boldsymbol{u}) \rightarrow \boldsymbol{z}$

Input : $n \in \mathbb{Z}, q \in \mathbb{Z}, \mathbf{A} \in \mathbb{Z}_q^{n \times m}, \mathbf{T_A} \in \mathbb{Z}^{\overline{m} \times \omega}, s \in \mathbb{R}, \sigma \in \mathbb{R}, \boldsymbol{u} \in \mathbb{Z}_q^n$;
Output : $\boldsymbol{z} \in \mathbb{Z}^m$;

1: $l = \lceil \log_2 q \rceil$; $\omega = n.l$;
2: $\boldsymbol{p} \leftarrow \mathsf{SamplePerturbation}(n, q, \mathbf{T_A}, s, \sigma)$; $\boldsymbol{p} \in \mathbb{Z}^m$;
3: $\boldsymbol{v} = \boldsymbol{u} - \mathbf{A} \cdot \boldsymbol{p} \in \mathbb{Z}_q^n$;
4: $\boldsymbol{z'} \leftarrow \mathsf{GSampling}(n, \boldsymbol{v}, s, q)$; $\boldsymbol{z'} \in \mathbb{Z}^\omega$;
5: $\boldsymbol{z} = \boldsymbol{p} + \begin{bmatrix} \mathbf{T_A} \\ \\ \mathbf{I}_\omega \end{bmatrix} \cdot \boldsymbol{z'}$; $\mathbf{I}_\omega$ is identity matrix of order ω; $\boldsymbol{z} \in \mathbb{Z}^m$
6: **return** $\boldsymbol{z}$

Algorithm 15 SamplePerturbation$(n, q, \mathbf{T_A}, s, \sigma) \rightarrow \boldsymbol{p}$

Input : $n \in \mathbb{Z}, q \in \mathbb{Z}, \mathbf{T_A} \in \mathbb{Z}^{\overline{m} \times \omega}, s \in \mathbb{R}, \sigma \in \mathbb{R}$;
Output : $\boldsymbol{p} \in \mathbb{Z}^m$;

1: $\omega = n\lceil \log_2 q \rceil \in \mathbb{Z}$; $\overline{m} = m - \omega \in \mathbb{Z}$;
2: $\boldsymbol{r} \leftarrow \mathsf{RandomSample}(\omega, 1, \sqrt{\sigma^2 - s^2})$; $\boldsymbol{r} \in \mathbb{Z}^\omega$;
3: $\boldsymbol{c} = \frac{s^2}{\sigma^2 - s^2} \cdot \mathbf{T_A} \cdot \boldsymbol{r} \in \mathbb{R}^{\overline{m}}$;
4: $\mathbf{S} = \sigma^2 \cdot \mathbf{I}_{\overline{m}} - (s^{-2} - \sigma^{-2})^{-1} \cdot \mathbf{T_A} \cdot \mathbf{T_A}^\top \in \mathbb{R}^{\overline{m} \times \overline{m}}$;
5: **for** $(i = \overline{m}$ to $1)$ **do**
6: Split matrix $\mathbf{S} = \begin{bmatrix} \mathbf{A} & \mathbf{B} \\ \\ \mathbf{B}^\top & f \end{bmatrix}$ into block matrices;
7: where $\mathbf{A} \in \mathbb{R}^{i \times i}$, $\mathbf{B} \in \mathbb{R}^{i \times 1}$, $\mathbf{B}^\top \in \mathbb{R}^{1 \times i}$ and $f \in \mathbb{R}^{1 \times 1}$;
8: $\boldsymbol{c} = (\boldsymbol{c'}, c_i)$; where $\boldsymbol{c'} \in \mathbb{R}^{\overline{m}}$, $c_i \in \mathbb{R}$;
9: $p_i \leftarrow \mathsf{Sample}(1, \sqrt{f}, c_i)$ by running Algorithm 10; $p_i \in \mathbb{R}$;
10: $\boldsymbol{c} = \boldsymbol{c'} + f^{-1} \cdot (p_i - c_i) \cdot \mathbf{B} \in \mathbb{R}^{\overline{m}}$;
11: $\mathbf{S} = \mathbf{A} - f^{-1} \cdot \mathbf{B} \cdot \mathbf{B}^\top \in \mathbb{R}^{i \times i}$;
12: **end for**
13: $\boldsymbol{p} = (p_1, \ldots, p_{\overline{m}}, \boldsymbol{r}) \in \mathbb{Z}^m$;
14: **return** $\boldsymbol{p}$

17 and Algorithm 12 as subroutines and outputs a vector $\boldsymbol{z'} \in \mathbb{Z}^\omega$. To generate the vector $\boldsymbol{z'} \in \mathbb{Z}^\omega$ from the coset $\Lambda_q^{\boldsymbol{v}}(\mathbf{G})$ of the $\mathbf{G}$-lattice $\Lambda_q(\mathbf{G})$, n column vectors are sampled independently from the lattice $\Lambda_q^\perp(\boldsymbol{g})$ and arranged one after another to form the vector $\boldsymbol{z'} \in \mathbb{Z}^\omega$.

- The procedure SampleG in turn calls three procedures SampleGPerturb, SampleD and GetBit presented in Algorithm 18, Algorithm 19 and Algorithm 12 as subroutines and outputs a vector $\boldsymbol{t} \in \mathbb{Z}^l$.

Algorithm 16 $\mathsf{GSampling}(n, \boldsymbol{v}, \alpha, q) \rightarrow \boldsymbol{z}'$

Input : $n \in \mathbb{Z}, \boldsymbol{v} \in \mathbb{Z}^m, \alpha \in \mathbb{R}, q \in \mathbb{Z}$;
Output : $\boldsymbol{z}' \in \mathbb{Z}^\omega$;

$l = \lceil \log_2(q) \rceil \in \mathbb{Z}$; $\omega = n \cdot l \in \mathbb{Z}$;
$d_1 = \mathsf{GetBit}(q, 1)/2 \in \mathbb{R}$, $s_1 = \sqrt{2(1 + 1/l) + 1} \in \mathbb{R}$, $h_1 = 0 \in \mathbb{R}$;
for $(i = 2$ to $l)$ **do**
    $d_i = (d_{i-1} + \mathsf{GetBit}(q, i))/2 \in \mathbb{R}$; // d_i is the i-th component of $\boldsymbol{d} \in \mathbb{R}^l$
    $s_i = \sqrt{2(1 + 1/(l - i))} \in \mathbb{R}$; // s_i is the i-th component of $\boldsymbol{s} \in \mathbb{R}^l$
    $h_i = \sqrt{2(1 - 1/\{l - (i - 1)\})} \in \mathbb{R}$; // h_i is the i-th component of $\boldsymbol{h} \in \mathbb{R}^l$
end for
for $(i = 1$ to $n)$ **do**
    $\boldsymbol{t} \leftarrow \mathsf{SampleG}(q, v_i, \boldsymbol{s}, \boldsymbol{h}, \boldsymbol{d}, s)$; where $\boldsymbol{t} \in \mathbb{Z}^n$;
    for $(j = 1$ to $l)$ **do**
        $z'_{(i-1)\cdot l+j} = t_j$; // t_j is the j-th component of $\boldsymbol{t} \in \mathbb{Z}^l$, z'_j is the j-th component of $\boldsymbol{z}' \in \mathbb{Z}^\omega$
    end for
end for// This for loop can be executed parallelly;
return $\boldsymbol{z}'$

Algorithm 17 $\mathsf{SampleG}(q, u, \boldsymbol{s}, \boldsymbol{h}, \boldsymbol{d}, s) \rightarrow \boldsymbol{t}$

Input : $q \in \mathbb{Z}, u \in \mathbb{Z}, \boldsymbol{s} \in \mathbb{R}^l, \boldsymbol{h} \in \mathbb{R}^l, \boldsymbol{d} \in \mathbb{R}^l, s \in \mathbb{R}$;
Output : $\boldsymbol{t} \in \mathbb{Z}^l$;

$\psi' = s/3 \in \mathbb{R}, l = \lceil \log_2(q) \rceil \in \mathbb{Z}$;
$\boldsymbol{p} \leftarrow \mathsf{SampleGPerturb}(q, \psi', \boldsymbol{s}, \boldsymbol{h})$; where $\boldsymbol{p} \in \mathbb{Z}^l$;
$c_1 = (\mathsf{GetBit}(u, 1) - p_1)/2 \in \mathbb{R}$;
for $(i = 2$ to $l)$ **do**
    $c_i = (c_{i-1} + \mathsf{GetBit}(u, i) - p_i)/2 \in \mathbb{R}$;
end for
$\boldsymbol{w} \leftarrow \mathsf{SampleD}(q, \psi', \boldsymbol{c}, \boldsymbol{d})$; where $\boldsymbol{w} \in \mathbb{Z}^l$;
$t_1 = 2 \cdot w_1 + \mathsf{GetBit}(q, 1) \cdot w_l + \mathsf{GetBit}(u, 1) \in \mathbb{Z}$;
for $(i = 2$ to $l - 1)$ **do**
    $t_i = 2 \cdot w_i - w_{i-1} + \mathsf{GetBit}(q, i) \cdot w_{l-1} + \mathsf{GetBit}(u, i) \in \mathbb{Z}$;// t_i is the i-th component of $\boldsymbol{t} \in \mathbb{Z}^l$;
end for
$t_l = \mathsf{GetBit}(q, l) \cdot w_l - w_{l-1} + \mathsf{GetBit}(u, l) \in \mathbb{Z}$;
return $\boldsymbol{t}$

- The procedure $\mathsf{SampleGPerturb}$ outputs a perturbation vector $\boldsymbol{p} \in \mathbb{Z}^m$ and procedure $\mathsf{SampleD}$ produces a vector $\boldsymbol{w} \in \mathbb{Z}^l$ using the rejection sampling algorithm Sample described as Algorithm 10.

Algorithm 18 $\mathsf{SampleGPerturb}(q, \psi', \boldsymbol{s}, \boldsymbol{h}) \rightarrow \boldsymbol{p}$

Input : $q \in \mathbb{Z}, \psi' \in \mathbb{R}, \boldsymbol{s} \in \mathbb{R}^l, \boldsymbol{h} \in \mathbb{R}^l$;
Output : $\boldsymbol{p} \in \mathbb{Z}^l$;

1: $\beta = 0 \in \mathbb{R}$, $z_1'' = 0 \in \mathbb{Z}$, $l = \log_2(q)$;
2: **for** $(i = 1$ to $l)$ **do**
3: $c_i = \beta / s_i \in \mathbb{R}$, $\mathsf{frac}(c_i) = c_i - \lfloor c_i \rfloor \in [0, 1)$, $\psi_i' = \psi' / s_i \in \mathbb{R}$;// s_i is the i-th component of $\boldsymbol{s} \in \mathbb{R}^l$
4: $z_i'' \leftarrow \lfloor c_i \rfloor + \mathsf{Sample}(1, \psi_i', \mathsf{frac}(c_i)) \in \mathbb{Z}$; using Algorithm 10; // z_i'' is the i-th component of $\boldsymbol{z}'' \in \mathbb{Z}^l$
5: $\beta = -z_i'' \cdot h_i \in \mathbb{R}$;// h_i is the i-th component of $\boldsymbol{h} \in \mathbb{R}^l$
6: **end for**
7: $p_1 = 5 \cdot z_1'' + 2 \cdot z_2'' \in \mathbb{Z}$;
8: **for** $(i = 2$ to $l - 1)$ **do**
9: $p_i = 2(z_{i-1}'' + 2 \cdot z_i'' + z_{i+1}'')$;// p_i is the i-th component of $\boldsymbol{p} \in \mathbb{Z}^l$
10: **end for**
11: $p_l = 2(z_{l-1}'' + 2 \cdot z_l'')$;
12: **return** $\boldsymbol{p}$

Algorithm 19 $\mathsf{SampleD}(q, \psi', \boldsymbol{c}, \boldsymbol{d}) \rightarrow \boldsymbol{w}$

Input : $q \in \mathbb{Z}, \psi' \in \mathbb{R}, \boldsymbol{c} \in \mathbb{R}^l, \boldsymbol{d} \in \mathbb{R}^l$;
Output : $\boldsymbol{w} \in \mathbb{Z}^l$;

1: $l = \log_2 q$;
2: $w_l \leftarrow \mathsf{Sample}(1, \psi'/d_l, -c_l/d_l)$ using Algorithm 10; where $w_l \in \mathbb{Z}$;
3: $\boldsymbol{c} = \boldsymbol{c} - w_l \cdot \boldsymbol{d} \in \mathbb{R}^l$;
4: **for** $(i = 1$ to $l - 1)$ **do**
5: $w_i \leftarrow \mathsf{Sample}(1, \psi', -c_i)$; // w_i is the i-th component of $\boldsymbol{w} \in \mathbb{Z}^l$
6: **end for**
7: **return** $\boldsymbol{w}$

3.2 Parameter Choice

We employ a procedure Param for the choice of parameters for the uIPFE [1]. This procedure generates the parameters which satisfy the constraints given in parameter settings in subsection 2.3.

- The procedure Param on input $n, \ell_{\mathsf{max}}, \sigma, X, Y$ generates q, m, K, ψ, α by using the procedure PrimeGen as a subroutine.
- The procedure PrimeGen takes an integer a as input and generates a prime number p greater than a. It calls a procedure isPrime as subroutine.

Algorithm 20 $\mathsf{Param}(n, \ell_{\mathsf{max}}, \sigma, X, Y) \rightarrow (q, m, K, \psi, \alpha)$

Input : $n \in \mathbb{Z}, \ell_{\mathsf{max}} \in \mathbb{Z}, \sigma \in \mathbb{R}, X \in \mathbb{Z}, Y \in \mathbb{Z}$;
Output : $q \in \mathbb{Z}, m \in \mathbb{Z}, K \in \mathbb{Z}, \psi \in \mathbb{R}, \alpha \in \mathbb{R}$;
1: $a = 2 \in \mathbb{Z}, i = 0 \in \mathbb{Z}, j = 2 \in \mathbb{Z}$;
2: $K = XY\ell_{\mathsf{max}} \in \mathbb{Z}$; $\psi = \sqrt{n} \in \mathbb{R}$;
3: **do**
4: $\quad l = \lceil \log_2 a \rceil \in \mathbb{Z}$; $m = 2 \cdot n \cdot l \in \mathbb{Z}$;
5: $\quad \mathsf{rhs} = 2K(\sigma\psi m\ell_{\mathsf{max}}Y + Y\ell_{\mathsf{max}}\psi) \in \mathbb{R}$;
6: $\quad a = 2^j$; $j = j + 1$;
7: **while** $(a < \mathsf{rhs})$;
8: $q = \mathsf{PrimeGen}(a) \in \mathbb{Z}$;
9: $l = \lceil \log_2 q \rceil \in \mathbb{Z}$; $m = 2 \cdot n \cdot l \in \mathbb{Z}$;
10: **do**
11: $\quad \alpha = \frac{\psi}{2 \cdot q \cdot 10^{-i} \cdot (\sqrt{m} + \sqrt{n} + \sqrt{\ell_{\mathsf{max}}})} \in \mathbb{R}$;
12: $\quad i = i + 1$;
13: **while** $(\alpha \cdot q < 2 \cdot \sqrt{n})$;
14: **return** q, m, K, ψ, α

Algorithm 21 $\mathsf{PrimeGen}(a) \rightarrow p$

Input : $a \in \mathbb{Z}$;
Output : $p \in \mathbb{Z}$;
1: $p = a + 1$;
2: **while** (1) **do**
3: $\quad$ **if** $(\mathsf{isPrime}(p))$ **then**
4: $\quad\quad$ **break**;
5: $\quad$ **end if**
6: $\quad p = p + 1$;
7: **end while**
8: **return** p

Algorithm 22 $\mathsf{isPrime}(p) \rightarrow 1 \vee 0$

Input : $p \in \mathbb{Z}$;
Output : $1 \vee 0$;
1: **for** $(i = 2$ to $\sqrt{p})$ **do**
2: $\quad$ **if** $((p$ MODULO $i\ ) = 0)$ **then return** 0
3: $\quad\quad$ **break**;
4: $\quad$ **end if**
5: **end for**
6: **return** 1

- The procedure isPrime takes an integer p and returns 1 if p is a prime integer and 0 otherwise. We denote 'a MODULO b' to represent the modulo operator which yields the remainder when the integer a is divided by the integer b.

4 Our scheme uIPFE #1

This section presents our *unbounded inner product functional encryption* scheme uIPFE scheme #1. The co-ordinates of the message vectors are from $\mathcal{X} = \{0, 1, \ldots, X(\lambda)-1\}$ and those of the secret key vectors are from $\mathcal{Y} = \{0, 1, \ldots, Y(\lambda) - 1\}$. The length of the message vector and the decryption key vector are less than ℓ_{max}. The inner product between the message vector $\boldsymbol{x} \in \mathcal{X}^{\ell_{\mathsf{max}}}$ and the associated decryption key vector $\boldsymbol{y} \in \mathcal{Y}^{\ell_{\mathsf{max}}}$ lies between $\{0, 1, \ldots, K-1\}$, that is, $\langle \boldsymbol{x}, \boldsymbol{y} \rangle < K$ where $K = XY\ell_{\mathsf{max}}$.

– $\mathsf{uIPFE.Setup}(1^\lambda) \to (\mathsf{Mpk}, \mathsf{Msk})$: On input a security parameter λ, a trusted authority executes the steps as described in Algorithm 23 below choosing the parameters as in Table 2 and generates master public-key Mpk and master secret-key Msk. Mpk is published and Msk is kept secret to the trusted authority.

Algorithm 23 $\mathsf{uIPFE.Setup}(1^\lambda) \to (\mathsf{Mpk}, \mathsf{Msk})$

Input : λ;
Output : $\mathsf{Mpk} = (\mathsf{pp}, \mathbf{A}, \mathbf{W})$ with $\mathsf{pp} = (n, q, m, \ell_{\mathsf{max}}, X, Y, K, \psi, \alpha, \sigma, \mathcal{H})$; $\mathsf{Msk} = \mathbf{Z}$;
1: $n = n(\lambda) \in \mathbb{Z}, q = q(\lambda) \in \mathbb{Z}$, $m = 2n \log_2 q \in \mathbb{Z}$;
2: $\ell_{\mathsf{max}} = \ell_{\mathsf{max}}(\lambda) \in \mathbb{Z}, X = X(\lambda) \in \mathbb{Z}, Y = Y(\lambda) \in \mathbb{Z}$, $K = XY\ell_{\mathsf{max}} \in \mathbb{Z}$;
3: $\psi = \psi(\lambda) \in \mathbb{R}$, $\alpha = \alpha(\lambda) \in \mathbb{R}$, $\sigma = \sigma(\lambda) \in \mathbb{R}$;
4: Choose a hash function $\mathcal{H} : \mathbb{N} \to \{-1, 1\}^{\ell_{\mathsf{max}}}$ which is modelled as random oracle;
5: $\mathbf{A} \xleftarrow{\$} \mathbb{Z}_q^{n \times m}$;
6: $\mathbf{Z} \leftarrow \mathsf{RandomSample}(m, \ell_{\mathsf{max}}, \sigma)$; $\mathbf{Z} \in \mathfrak{D}_{\mathbb{Z}^{m \times \ell_{\mathsf{max}}}, \sigma}$;
7: Compute $\mathbf{W} = \mathbf{AZ} \pmod q \in \mathbb{Z}_q^{n \times \ell_{\mathsf{max}}}$;
8: **return** $(\mathsf{pp} = (n, q, m, \ell_{\mathsf{max}}, X, Y, K, \psi, \alpha, \sigma, \mathcal{H}), \mathsf{Mpk} = (\mathsf{pp}, \mathbf{A}, \mathbf{W}), \mathsf{Msk} = \mathbf{Z})$

– $\mathsf{uIPFE.KeyExtract}(\mathsf{Mpk}, \mathsf{Msk}, \boldsymbol{y}) \to \mathsf{Dk}_{\boldsymbol{y}}$: On input the master public key Mpk, master secret key Msk and a secret key vector $\boldsymbol{y} = (y_i)_{i \in I_{\boldsymbol{y}}} \in \mathcal{Y}^{\ell'}$ received from a user, the trusted authority performs the steps as shown in Algorithm 24 below where $\ell' = |I_{\boldsymbol{y}}|$, $I_{\boldsymbol{y}}$ being the index set of $\boldsymbol{y}$. The decryption key $\mathsf{Dk}_{\boldsymbol{y}}$ generated by this algorithm is sent to the user through a secure communication channel.

Algorithm 24 $\mathsf{uIPFE.KeyExtract}(\mathsf{Mpk}, \mathsf{Msk}, \boldsymbol{y}) \to \mathsf{Dk}_{\boldsymbol{y}}$

Input : $\mathsf{Mpk} = (\mathsf{pp}, \mathbf{A}, \mathbf{W})$ with $\mathsf{pp} = (n, q, m, \ell_{\mathsf{max}}, X, Y, K, \psi, \alpha, \sigma, \mathcal{H})$; $\mathsf{Msk} = \mathbf{Z}$; $\boldsymbol{y} = (y_i)_{i \in I_{\boldsymbol{y}}}$ with $|I_{\boldsymbol{y}}| = \ell' \leq \ell_{\mathsf{max}}$;
Output : $\mathsf{Dk}_{\boldsymbol{y}} = (\boldsymbol{y}^\top, \boldsymbol{y}^\top \mathbf{U}'^\top \mathbf{Z}^\top)$;
1: **for** $i = 1$ to ℓ' **do**
2: $\quad \boldsymbol{u}'_i \leftarrow \mathcal{H}(I_{\boldsymbol{y}}[i])$;
3: **end for**
4: Set $\mathbf{U}' = [\boldsymbol{u}'_1 \| \boldsymbol{u}'_2 \| \ldots \| \boldsymbol{u}'_{\ell'}] \in \{-1, 1\}^{\ell_{\mathsf{max}} \times \ell'}$;
5: **return** $\mathsf{Dk}_{\boldsymbol{y}} = (\boldsymbol{y}^\top, \boldsymbol{y}^\top \mathbf{U}'^\top \mathbf{Z}^\top)$

– $\mathsf{uIPFE.Encrypt}(\mathsf{Mpk}, \boldsymbol{x}) \to \mathsf{Ct}_{\boldsymbol{x}}$: The encryptor takes as input the Mpk and a message vector $\boldsymbol{x} = (x_i)_{i \in I_{\boldsymbol{x}}} \in \mathcal{X}^\ell$ where $\ell = |I_{\boldsymbol{x}}|, I_{\boldsymbol{x}} \subseteq \mathbb{N}$ is the non-empty index set of $\boldsymbol{x}$ and generates the ciphertext $\mathsf{Ct}_{\boldsymbol{x}}$ following the steps presented in Algorithm 25 below and sends $\mathsf{Ct}_{\boldsymbol{x}}$ over a public channel to the decryptor.

Algorithm 25 $\mathsf{uIPFE.Encrypt}(\mathsf{Mpk}, \boldsymbol{x}) \rightarrow \mathsf{Ct}_{\boldsymbol{x}}$

Input : $\mathsf{Mpk} = (\mathsf{pp}, \mathbf{A}, \mathbf{W})$ with $\mathsf{pp} = (n, q, m, \ell_{\mathsf{max}}, X, Y, K, \psi, \alpha, \sigma, \mathcal{H})$; $\boldsymbol{x} = (x_i)_{i \in I_{\boldsymbol{x}}}$ with $|I_{\boldsymbol{x}}| = \ell \leq \ell_{\mathsf{max}}$;
Output : $\mathsf{Ct}_{\boldsymbol{x}} = (\boldsymbol{c}_1, \boldsymbol{c}_2)$;
1: **for** $i = 1$ to ℓ **do**
2: $\quad \boldsymbol{u}_i \leftarrow \mathcal{H}(I_{\boldsymbol{x}}[i])$;
3: **end for**
4: Set $\mathbf{U} = [\boldsymbol{u}_1 \| \boldsymbol{u}_2 \| \ldots \| \boldsymbol{u}_\ell] \in \{-1, 1\}^{\ell_{\mathsf{max}} \times \ell}$;
5: $\boldsymbol{s} \xleftarrow{\$} \mathbb{Z}_q^n, \boldsymbol{e}_1 \xleftarrow{\$} \mathfrak{D}_{\mathbb{Z}^m, \psi}$, $\boldsymbol{e}_2 \xleftarrow{\$} \mathfrak{D}_{\mathbb{Z}^\ell, \psi}$;
6: $\boldsymbol{c}_1 = \mathbf{A}^\top \boldsymbol{s} + \boldsymbol{e}_1 \in \mathbb{Z}_q^m$;
7: $\boldsymbol{c}_2 = \mathbf{U}^\top \mathbf{W}^\top \boldsymbol{s} + \lfloor \frac{q}{K} \rfloor \boldsymbol{x} + \boldsymbol{e}_2 \in \mathbb{Z}_q^\ell$;
8: **return** $\mathsf{Ct}_{\boldsymbol{x}} = (\boldsymbol{c}_1, \boldsymbol{c}_2)$

– $\mathsf{uIPFE.Decrypt}(\mathsf{Mpk}, \mathsf{Dk}_{\boldsymbol{y}}, \mathsf{Ct}_{\boldsymbol{x}}) \rightarrow \mu \vee \perp$: On input the Mpk, a decryptor uses its decryption key $\mathsf{Dk}_{\boldsymbol{y}}$ corresponding to its secret key vector $\boldsymbol{y} \in \mathcal{Y}^{\ell'}$ and a ciphertext $\mathsf{Ct}_{\boldsymbol{x}}$, the decryptor executes the steps as in Algorithm 26 below.

Algorithm 26 $\mathsf{uIPFE.Decrypt}(\mathsf{Mpk}, \mathsf{Dk}_{\boldsymbol{y}}, \mathsf{Ct}_{\boldsymbol{x}}) \rightarrow \mu$ or $\perp$

Input : $\mathsf{Mpk} = (\mathsf{pp}, \mathbf{A}, \mathbf{W})$ with $\mathsf{pp} = (n, q, m, \ell_{\mathsf{max}}, X, Y, K, \psi, \alpha, \sigma, \mathcal{H})$; $\mathsf{Dk}_{\boldsymbol{y}} = (\boldsymbol{y}^\top, \boldsymbol{y}^\top \mathbf{U}'^\top \mathbf{Z}^\top)$, $\mathsf{Ct}_{\boldsymbol{x}} = (\boldsymbol{c}_1, \boldsymbol{c}_2)$;
Output : $\mu = \langle \boldsymbol{x}, \boldsymbol{y} \rangle$;
1: $\mu' = \boldsymbol{y}^\top \boldsymbol{c}_2 - \boldsymbol{y}^\top \mathbf{U}'^\top \mathbf{Z}^\top \boldsymbol{c}_1$;
2: **for** ($\mu = 0$ to $K - 1$) **do**
3: $\quad$ **if** ($|\lfloor \frac{q}{K} \rfloor \mu - \mu'| < \sigma \psi m Y \ell_{\mathsf{max}}^2 + Y \ell_{max} \psi$) **then**
4: $\quad\quad$ **break**;
5: $\quad$ **end if**
6: **end for**
7: **return** μ

Parameter Setting

The parameter setting of our scheme uIPFE #1 which is based on $\mathsf{LWE}_{q,\alpha,n}$ is presented below in Table 2 where αq represents the standard deviation of LWE's noise distribution.

Correctness

Correctness of our scheme uIPFE #1= $(\mathsf{Setup}, \mathsf{KeyExtract}, \mathsf{Encrypt}, \mathsf{Decrypt})$ follows from the following observation. The decryption succeed if $I_{\boldsymbol{x}} = I_{\boldsymbol{y}}$ (i.e., $\ell = \ell' \leq \ell_{\mathsf{max}}$), we have

$$\begin{aligned} \mu' &= \boldsymbol{y}^\top \boldsymbol{c}_2 - \boldsymbol{y}^\top \mathbf{U}'^\top \mathbf{Z}^\top \boldsymbol{c}_1 \\ &= \boldsymbol{y}^\top (\mathbf{U}^\top \mathbf{W}^\top \boldsymbol{s} + \left\lfloor \frac{q}{K} \right\rfloor \boldsymbol{x} + \boldsymbol{e}_2) - \boldsymbol{y}^\top \mathbf{U}'^\top \mathbf{Z}^\top (\mathbf{A}^\top \boldsymbol{s} + \boldsymbol{e}_1) \end{aligned}$$

Table 2: Parameters for our uIPFE

Parameters	Constraints
m, n	$m \geq 2n \log q$
ψ, σ	$\psi \geq \omega(\sqrt{\log n}), \sigma \geq \omega(\sqrt{\log n})$
α, q, n	$\alpha q \geq 2\sqrt{n}$ where $\alpha \in (0,1) \cap \mathbb{R}$
$\alpha, q, m, n, \ell_{\mathsf{max}}$	$\alpha \leq \frac{\psi}{2C_1 q(\sqrt{m}+\sqrt{n}+\sqrt{\ell_{\mathsf{max}}})}$ where C_1 is a constant
$q, \sigma, \psi, \ell_{\mathsf{max}}, X, Y$	$q \geq 2K(\sigma\psi mY\ell_{\mathsf{max}}^2 + Y\ell_{\mathsf{max}}\psi)$ where $K = \ell_{\mathsf{max}}XY$

$$= \left\lfloor \frac{q}{K} \right\rfloor \langle \boldsymbol{x}, \boldsymbol{y} \rangle + \underbrace{\boldsymbol{y}^\top \boldsymbol{e}_2 - \boldsymbol{y}^\top \mathbf{U}'^\top \mathbf{Z}^\top \boldsymbol{e}_1}_{\mathsf{noise}}.$$

Decryption will give correct output if the magnitude of the error quantity $\mathsf{noise} = \boldsymbol{y}^\top \boldsymbol{e}_2 - (\boldsymbol{y}^\top \mathbf{U}'^\top \mathbf{Z}^\top)\boldsymbol{e}_1$ remains small. Since $\mathbb{Z} \in \mathfrak{D}_{\mathbb{Z}^{m \times \ell_{\mathsf{max}}}, \sigma}$, every column $\boldsymbol{z}_i$ of $\mathbf{Z} \in \mathbb{Z}^{m \times \ell_{\mathsf{max}}}$ satisfies $\|\boldsymbol{z}_i\| \leq \sigma\sqrt{m}$ yielding $\|\mathbf{Z}\| \leq \sigma\sqrt{m\ell_{\mathsf{max}}}$ and since $\mathbf{U}' \in \{-1, 1\}^{\ell_{\mathsf{max}} \times \ell}$, $\|\mathbf{U}'\| \leq \sqrt{\ell_{\mathsf{max}}\ell}$. Also $\|\boldsymbol{y}\| \leq Y\sqrt{\ell}$ as $\boldsymbol{y} \in \mathcal{Y}^{|I_{\boldsymbol{y}}|} = \{0, 1, \ldots, Y-1\}^{|I_{\boldsymbol{y}}|}$ with $|I_{\boldsymbol{y}}| = \ell' = \ell$, $\boldsymbol{e}_1 \xleftarrow{\$} \mathfrak{D}_{\mathbb{Z}^m, \psi}$ and $\boldsymbol{e}_2 \xleftarrow{\$} \mathfrak{D}_{\mathbb{Z}^\ell, \psi}$ implies $\|\boldsymbol{e}_1\| \leq \sqrt{m}\psi$ and $\|\boldsymbol{e}_2\| \leq \sqrt{\ell}\psi$ which in turn implies that

$$\|(\boldsymbol{y}^\top \mathbf{U}'^\top \mathbf{Z}^\top)\boldsymbol{e}_1\| \leq \|\boldsymbol{y}\|\|\mathbf{U}'\|\|\mathbf{Z}\|\|\boldsymbol{e}_1\| \leq Y\sqrt{\ell}\sqrt{\ell_{\mathsf{max}}\ell}\sigma\sqrt{m\ell_{\mathsf{max}}}\psi\sqrt{m} \leq \sigma\psi m\ell Y\ell_{\mathsf{max}}$$

by choosing $\psi \geq 2C_1\alpha q(\sqrt{m} + \sqrt{n} + \sqrt{\ell})$ where C_1 is a constant. Therefore, $|\mathsf{noise}| \leq \sigma\psi m\ell Y\ell_{\mathsf{max}} + Y\ell\psi$ with overwhelming probability. To guarantee the decryption's accuracy, it is crucial to make sure that the amount of noise is smaller than $\frac{q}{2K}$ which can be acheived by setting $q \geq 2K(\sigma\psi mY\ell_{\mathsf{max}}^2 + Y\ell_{\mathsf{max}}\psi)$ where ℓ_{max} represents the maximum magnitude of the message vectors and the decryption key vectors.

Theorem 9 *Assume that the modified variation of ALS-IPFE discussed in subsection 2.3 is Adp-IND secure according to Definition 4 depending upon the validity of the $\mathsf{LWE}_{q,\alpha,n}$ assumption (see Definition 6) for the condition $q \geq 2K\sigma\psi m\sqrt{|I|}Y + 2KY|I|\psi$ where $n, m, K, X, Y, \sigma, \psi$ are public parameters as described above in parameter settings of subsection 2.3. Then our scheme uIPFE #1 = (uIPFE.Setup, uIPFE.KeyExtract, uIPFE.Encrypt, uIPFE.Decrypt) presented above is Adp-IND secure according to Definition 4 in the random oracle model (ROM) under the hardness of $\mathsf{LWE}_{q,\alpha,n}$ problem where $m \geq 2n \log q, \sigma \geq \omega(\sqrt{\log n}), \psi \geq \omega(\sqrt{\log n}), \alpha q \geq 2\sqrt{n}, \alpha \leq \frac{\psi}{2C_1 q(\sqrt{m}+\sqrt{n}+\sqrt{\ell_{\mathsf{max}}})}$ and $q \geq 2K(\sigma\psi mY\ell_{\mathsf{max}}^2 + Y\ell_{\mathsf{max}}\psi)$.*

Proof To demonstrate the security of our uIPFE #1, we conduct a sequences of hybrid games involving a challenger $\mathcal{B}$ and a PPT adversary $\mathcal{A}$. Assume that $\mathsf{Adv}_{\mathcal{A},\mathrm{Game}_i}(\lambda)$ represents $\mathcal{A}$'s advantage in Game i. We consider without loss of generality that when $\mathcal{A}$ makes the j-th decryption key query for the secret key on a vector $\boldsymbol{y}^{(j)} = (y_i^{(j)})_{i \in I}$, $\mathcal{A}$ initially makes

query to a random oracle (RO) $\mathcal{H}$ for the finite index set $I \subseteq \mathbb{N}$ where $|I| \leq \ell_{\mathsf{max}}$ with ℓ_{max} denoting the maximum magnitude of message vectors and the secret key vectors.
Game 0: $\mathcal{B}$ produces $(\mathsf{Mpk}, \mathsf{Msk}) \leftarrow \mathsf{uIPFE.Setup}(1^\lambda)$ with $\mathsf{Mpk} = (\mathsf{pp}, \mathbf{A}, \mathbf{W})$, $\mathsf{Msk} = \mathbf{Z}$ and transmits Mpk to $\mathcal{A}$. We have, $\mathbf{A} \in \mathbb{Z}_q^{n \times m}, \mathbf{W} \in \mathbb{Z}_q^{n \times \ell_{\mathsf{max}}}, \mathbf{Z} \in \mathcal{D}_{\mathbb{Z}^{m \times \ell_{\mathsf{max}}}, \sigma}$, pp includes $m = m(\lambda) \in \mathbb{Z}, n = n(\lambda) \in \mathbb{Z}, X = X(\lambda) \in \mathbb{Z}_q, Y = Y(\lambda) \in \mathbb{Z}_q, \psi(\lambda), \alpha(\lambda), \sigma(\lambda) \in \mathbb{R}^+$ and $K = \ell_{\mathsf{max}} XY \in \mathbb{Z}$ is polynomially constrained in λ. After receiving pp, $\mathcal{A}$ makes a polynomial number of decryption key queries. For the j-th decryption key query on the vector $\boldsymbol{y}^{(j)} = (y_i^{(j)})_{i \in I}$, $\mathcal{B}$ then calculates the secret key $\mathsf{sk}_{\boldsymbol{y}^{(j)}} \leftarrow \mathsf{uIPFE.KeyExtract}(\mathsf{Mpk}, \mathsf{Msk}, \boldsymbol{y}^{(j)})$ and provides the secret key back to $\mathcal{A}$. $\mathcal{A}$ provides two different challenge vectors $\boldsymbol{x}^{(0)} = (x_i^{(0)})_{i \in I_0^*}$ and $\boldsymbol{x}^{(1)} = (x_i^{(1)})_{i \in I_1^*}$ with $|I_0^*| = |I_1^*| = \ell$ to $\mathcal{B}$ such that $\langle \boldsymbol{x}^{(0)}, \boldsymbol{y}^{(j)} \rangle = \langle \boldsymbol{x}^{(1)}, \boldsymbol{y}^{(j)} \rangle$ is satisfied whenever $I_0^* = I_1^* = I \; \forall \; \boldsymbol{y}^{(j)} = (y_i^{(j)})_{i \in I}$ for which decryption key query has already been made. Then $\mathcal{B}$ selects a bit $\beta \xleftarrow{\$} \{0, 1\}$ uniformly at random and transmit the challenge ciphertext $\mathsf{Ct}_{\boldsymbol{x}^{(\beta)}} = (\boldsymbol{c}_1, \boldsymbol{c}_2) \leftarrow \mathsf{uIPFE.Encrypt}(\mathsf{Mpk}, \boldsymbol{x}^{(\beta)})$ to $\mathcal{A}$ where $\boldsymbol{c}_1 = \mathbf{A}^\top \boldsymbol{s} + \boldsymbol{e}_1 \in \mathbb{Z}_q^m, \boldsymbol{c}_2 = \mathbf{U}^\top \mathbf{W}^\top \boldsymbol{s} + \lfloor \frac{q}{K} \rfloor \boldsymbol{x}^{(\beta)} + \boldsymbol{e}_2 \in \mathbb{Z}_q^\ell$ and $\boldsymbol{s} \xleftarrow{\$} \mathbb{Z}_q^n, \boldsymbol{e}_1 \xleftarrow{\$} \mathfrak{D}_{\mathbb{Z}^m, \psi}, \boldsymbol{e}_2 \xleftarrow{\$} \mathfrak{D}_{\mathbb{Z}^\ell, \psi}, \mathbf{U} = [\boldsymbol{u}_1 \| \boldsymbol{u}_2 \| , \ldots, \| \boldsymbol{u}_{|I_\beta^*|}] \in \mathbb{Z}_q^{\ell_{\mathsf{max}} \times \ell}$ with $\mathcal{H}(I_\beta^*[i]) = \boldsymbol{u}_i$ for all $I_\beta^*[i] \in I_\beta^*$. The two real numbers σ, ψ are selected in a way that $\sigma \geq \omega(\sqrt{\log n})$ and $\psi \geq 2C_1 \alpha q(\sqrt{m} + \sqrt{n} + \sqrt{\ell_{\mathsf{max}}})$ where q is a large prime and $\alpha \in (0, 1) \cap \mathbb{R}$ satisfying $q \geq 2K(\sigma \psi m Y \ell_{\mathsf{max}}^2 + Y \ell_{\mathsf{max}} \psi)$.
Game 1: $\mathcal{A}$ tries to predict the index set I_0^* or I_1^* that $\mathcal{B}$ uses to generate the challenger ciphertext $\mathsf{Ct}_{\boldsymbol{x}^{(\beta)}}$ linked to the vectors $\boldsymbol{x}^{(0)}$ and $\boldsymbol{x}^{(1)}$ respectively. An exponential security loss for $\mathcal{A}$ would result from guessing the directly index set I_0^* or I_1^* used to generate the challenge ciphertext $\mathsf{Ct}_{\boldsymbol{x}^{(\beta)}}$. Rather, $\mathcal{A}$ selects the index of RO queries on I_0^* and I_1^* to obtain the matrices $\mathbf{U}_0 \in \mathbb{Z}_q^{\ell_{\mathsf{max}} \times |I_0^*|}$ and $\mathbf{U}_1 \in \mathbb{Z}_q^{\ell_{\mathsf{max}} \times |I_1^*|}$ respectively. For each index set query, it takes $\frac{1}{Q_{\mathsf{ro}}^2}$ security loss where Q_{ro} is the total number of queries sent to the RO $\mathcal{H}$.

The following two scenarios are taken into consideration by $\mathcal{B}$ based on the challenge index sets I_0^* and I_1^*.

- **Case I** ($I_0^* = I_1^*$), $\mathcal{B}$ performs the $\mathsf{ALS\text{-}IPFE}$ reduction to demonstrate the security of our scheme approach.
- **Case II** ($I_0^* \neq I_1^*$), $\mathcal{B}$ employs the LWE reduction directly.

Case I ($I_0^* = I_1^*$)

$\mathcal{B}$ shows that the challenge ciphertext which is encrypting either message $\boldsymbol{x}^{(0)}$ or message $\boldsymbol{x}^{(1)}$ is indistinguishable by utilizing $\mathsf{ALS\text{-}IPFE}$'s security as described in subsection 2.3. When interacting with $\mathsf{ALS\text{-}IPFE}$ challenger $\mathcal{C}$, $\mathcal{B}$ is regarded as an intermediate adversary. We demonstrate that this game is reduced to the $\mathsf{Adp\text{-}IND}$ security of $\mathsf{ALS\text{-}IPFE}$ with the challenger's public key $\mathsf{Mpk}_{\mathsf{ALS}} = (\mathsf{pp}_{\mathsf{ALS}}, \mathbf{A}_{\mathsf{ALS}}, \mathbf{D}_{\mathsf{ALS}})$ with $\mathsf{pp} = (n, q, m, \ell, X, Y, K, \psi, \alpha, \sigma)$, $\mathbf{A}_{\mathsf{ALS}} \in \mathbb{Z}_q^{n \times m}$ and $\mathbf{D}_{\mathsf{ALS}} \in \mathbb{Z}_q^{n \times \ell}$ along with $|I_0^*| = |I_0^*| = \ell$ fulfilling $\mathbf{D}_{\mathsf{ALS}} = \mathbf{A}_{\mathsf{ALS}} \mathbf{Z}_{\mathsf{ALS}}$ and $\mathbf{Z}_{\mathsf{ALS}} \in \mathfrak{D}_{\mathbb{Z}^{m \times \ell}, \sigma}$ is kept secret to $\mathcal{C}$. $\mathcal{B}$ simulates $\mathcal{A}$ in the following manner using the index sets $I_0^* = I_1^* = I^*$(say).

Setup

$\mathcal{B}$ sets $\mathbf{A} = \mathbf{A}_{\mathsf{ALS}}$ rather than selecting the matrix $\mathbf{A}$ from $\mathbb{Z}_q^{n \times m}$ as in Game 0. Every other parameter is formed just like in Game 0.

– After querying the RO for I with $|I| = \ell$, a random matrix $\mathbf{U}_I = [\boldsymbol{u}_1 \| \boldsymbol{u}_2 \| \dots \| \boldsymbol{u}_{|I|}] \xleftarrow{\$} \{-1,1\}^{\ell_{\mathsf{max}} \times \ell}$ is choosen by $\mathcal{B}$ and sets $\boldsymbol{u}_i = \mathcal{H}(I[i]) \in \{-1,1\}^{\ell_{\mathsf{max}}}$ to program the RO $\mathcal{H}$. The uIPFE #1 challenger $\mathcal{B}$ returns $\mathsf{sk}_{\boldsymbol{y}^{(j)}} = (\boldsymbol{y}^{(j)}, (\boldsymbol{y}^{(j)})^\top \mathbf{U}_I^\top \mathbf{Z}_I^\top)$ for the decryption key queries on the vector $\boldsymbol{y}^{(j)} = (y_i^{(j)})_{i \in I}$ with $|I| = \ell$. This distribution is identical to that of Game 0.

Challenge ciphertext

The challenge message vector pair $(\boldsymbol{x}^{(0)}, \boldsymbol{x}^{(1)})$ is sent by $\mathcal{A}$ to uIPFE #1 challenger $\mathcal{B}$ who then transmits it to ALS-IPFE challenger $\mathcal{C}$. Then $\mathcal{B}$ gives the challenge ciphertext $\mathsf{Ct}^{(\beta)} = (\boldsymbol{c}_1, \boldsymbol{c}_2)$ to $\mathcal{A}$ after receiving it from $\mathcal{C}$. Here $\boldsymbol{c}_1 = \mathbf{A}_{\mathsf{ALS}}^\top \boldsymbol{s} + \boldsymbol{e}_1 \in \mathbb{Z}_q^m, \boldsymbol{c}_2 = \mathbf{D}_{\mathsf{ALS}}^\top \boldsymbol{s} + \lfloor \frac{q}{K} \rfloor \boldsymbol{x}^{(\beta)} + \boldsymbol{e}_2$ where $\boldsymbol{e}_1 \xleftarrow{\$} \mathfrak{D}_{\mathbb{Z}^m, \psi}, \boldsymbol{e}_2 \xleftarrow{\$} \mathfrak{D}_{\mathbb{Z}^\ell, \psi}$.

Decryption keys

$\mathcal{B}$ sends the vector $\boldsymbol{y}^{(j)}$ to $\mathcal{C}$ for the j-th decryption key query on $\boldsymbol{y}^{(j)} = (y_i^{(j)})_{i \in I}$ with $I = I^*$(or $\neq I^*$). Then $\mathcal{C}$ generates the decryption key $\mathsf{sk}_{\boldsymbol{y}^{(j)}} = (\boldsymbol{y}^{(j)}, (\boldsymbol{y}^{(j)})^\top \mathbf{Z}_{\mathsf{ALS}}^\top)$ (here $\mathbf{Z}_{\mathsf{ALS}} \in \mathbb{Z}^{m \times \ell}$ is the master secret key of $\mathcal{C}$ satisfying $\mathbf{D}_{\mathsf{ALS}} = \mathbf{A}_{\mathsf{ALS}} \mathbf{Z}_{\mathsf{ALS}}$) and transmits it to $\mathcal{B}$ who then forwards it to $\mathcal{A}$. Consider the decryption key as $\mathsf{sk}_{\boldsymbol{y}^{(j)}} = (\boldsymbol{y}^{(j)}, (\boldsymbol{y}^{(j)})^\top \mathbf{U}_{I^*}^\top \mathbf{Z}^\top)$ where $\mathbf{A} \xleftarrow{\$} \mathbb{Z}_q^{n \times m}$, $\mathbf{U}_{I^*} = [\boldsymbol{u}_1 \| \boldsymbol{u}_2 \| \cdots \| \boldsymbol{u}_{|I^*|}] \in \mathbb{Z}_q^{\ell_{\mathsf{max}} \times \ell}$ with $\mathcal{H}(I^*[i]) = \boldsymbol{u}_i \in \{-1,1\}^{\ell_{\mathsf{max}}}$ for all $I^*[i] \in I^*$ and $\mathbf{Z} \in \mathfrak{D}_{\mathbb{Z}^{m \times \ell_{\mathsf{max}}}, \sigma}$ which has a distribution identical to that of Game 0. The decryption key $\mathsf{sk}_{\boldsymbol{y}^{(j)}} = (\boldsymbol{y}^{(j)}, (\boldsymbol{y}^{(j)})^\top \mathbf{Z}_{\mathsf{ALS}}^\top)$, on the other hand, has the same distribution as in Game 1 with $\mathbf{D}_{\mathsf{ALS}} = \mathbf{A}_{\mathsf{ALS}} \mathbf{Z}_{\mathsf{ALS}}$ where $\mathbf{D}_{\mathsf{ALS}} \in \mathbb{Z}_q^{n \times \ell}, \mathbf{A}_{\mathsf{ALS}} \in \mathbb{Z}_q^{n \times m}$ and $\mathbf{Z}_{\mathsf{ALS}} \in \mathfrak{D}_{\mathbb{Z}^{m \times \ell}, \sigma}$. In Game 1, $\mathcal{B}$ sets $\mathbf{A} = \mathbf{A}_{\mathsf{ALS}}$ and $\mathbf{W}\mathbf{U}_{I^*} = \mathbf{D}_{\mathsf{ALS}} = \mathbf{A}_{\mathsf{ALS}} \mathbf{Z}_{\mathsf{ALS}}$ for $I = I^*$ in Game 1 by implicitly setting $\mathbf{Z}\mathbf{U}_{I^*} = \mathbf{Z}_{\mathsf{ALS}}$. In both Games 0 and 1, the decryption keys are distributed in the same way. We can see that $\mathsf{Adv}_{\mathcal{A}, \mathrm{Game}_1}(\lambda) \leq \mathsf{Adv}^{\mathsf{Adp\text{-}IND}}_{\mathcal{B}, \mathsf{ALS\text{-}IPFE}}(\lambda)$ and from the following argument $|\mathsf{Adv}_{\mathcal{A}, \mathrm{Game}_1}(\lambda) - \mathsf{Adv}_{\mathcal{A}, \mathrm{Game}_0}(\lambda)| \leq \mathsf{Adv}^{\mathsf{Adp\text{-}IND}}_{\mathcal{B}, \mathsf{ALS\text{-}IPFE}}(\lambda)$.

• For $I = I^*$(or $\neq I^*$), we must show that the distribution of ALS-IPFE's public keys is statistically similar to that of honestly extracted keys, as in Game 0. While in Game 0, $\mathbf{W}$ takes the form $\mathbf{W} = \mathbf{A}\mathbf{Z}$ where $\mathbf{Z} \xleftarrow{\$} \mathfrak{D}_{\mathbb{Z}^{m \times \ell_{\mathsf{max}}}, \sigma}$ and $\mathbf{A} \xleftarrow{\$} \mathbb{Z}_q^{n \times m}$, the matrix the matrix $\mathbf{W}\mathbf{U}_{I^*} = \mathbf{D}_{\mathsf{ALS}}$ in ALS-IPFE takes the form $\mathbf{D}_{\mathsf{ALS}} = \mathbf{A}_{\mathsf{ALS}} \mathbf{Z}_{\mathsf{ALS}}$ where $\mathbf{Z}_{\mathsf{ALS}} \xleftarrow{\$} \mathfrak{D}_{\mathbb{Z}^{m \times \ell}, \sigma}$ and $\mathbf{A}_{\mathsf{ALS}} \xleftarrow{\$} \mathbb{Z}_q^{n \times m}$. Given $\mathbf{A} = \mathbf{A}_{\mathsf{ALS}}$ and $\mathbf{W}\mathbf{U}_{I^*} = \mathbf{D}_{\mathsf{ALS}}$, it is evident that the public keys of ALS-IPFE and Game 0 are connected. The RO $\mathcal{H}$ is programmed by $\mathcal{B}$ such that $\mathbf{W}\mathbf{U}_{I^*} = \mathbf{D}_{\mathsf{ALS}}$. The distribution of $\mathbf{Z}\mathbf{U}_I^*$ is equal to that of $\mathbf{Z}_{\mathsf{ALS}} \xleftarrow{\$} \mathfrak{D}_{\mathbb{Z}^{m \times \ell}, \sigma}$ according to Lemma 2 where σ of ALS-IPFE is set as σ of ALS-IPFE $= \sigma$ of uIPFE $\cdot \sqrt{\ell_{\mathsf{max}}}$. Consequently, the matrix $\mathbf{W}\mathbf{U}_{I^*}$ is not statistically close to uniform. Nevertheless, this is not a problem in our case because the distribution of $\mathbf{W}\mathbf{U}_{I^*}$ is uniform distributed given that it is in the form $\mathbf{A}\mathbf{Z}\mathbf{U}_{I^*} = \mathbf{A}_{\mathsf{ALS}} \mathbf{Z}_{\mathsf{ALS}}$ with $\mathbf{A}_{\mathsf{ALS}} \in \mathbb{Z}_q^{n \times m}$ and $\mathbf{Z}_{\mathsf{ALS}} \xleftarrow{\$} \mathfrak{D}_{\mathbb{Z}^{m \times \ell}, \sigma}$ ($\mathbf{Z}\mathbf{U}_{I^*}$ is implicitly set as $\mathbf{Z}_{\mathsf{ALS}}$). Therefore, we can conclude that $\mathsf{Dist}_0 = (\mathbf{A}_{\mathsf{ALS}}, \mathbf{Z}_{\mathsf{ALS}}, \mathbf{D}_{\mathsf{ALS}})$ is statistically close to $\mathsf{Dist}_1 = (\mathbf{A}, \mathbf{Z}\mathbf{U}_{I^*}, \mathbf{W}\mathbf{U}_{I^*})$ with uniform $\mathbf{U}_{I^*}$. Similar to the decryption keys in Game 0 corresponding to the vector $\boldsymbol{y}^{(j)} = (y_i^{(j)})_{i \in I}$, the keys in Game 1 (produced by $\mathcal{C}$) are distributed statistically close to honestly extracted keys.

Case II ($I_0^* \neq I_1^*$): $\mathcal{A}$ cannot get the decryption key for either the vector $(y_i^{(j)})_{i \in I_0^*}$ or $(y_i^{(j)})_{i \in I_1^*}$ in Adp-IND security model since these keys can readily discriminate between the encryption of $\boldsymbol{x}^{(0)} = (x_i^{(0)})_{i \in I_0^*}$ and $\boldsymbol{x}^{(1)} = (x_i^{(1)})_{i \in I_1^*}$. As a result, $\mathcal{A}$ is unable to query for

decryption keys associated with the index sets I_0^* and I_1^*. We show below that the challenger can immediately reduce the problem to LWE for the matrix

$$\mathbf{E} = \begin{bmatrix} \mathbf{A} & \mathbf{W}\mathbf{U}_0 & \mathbf{W}\mathbf{U}_1 \end{bmatrix} \in \mathbb{Z}_q^{n \times (m+\ell_0+\ell_1)}$$

where $|I_0^*| = \ell_0$ and $|I_1^*| = \ell_1$, $\mathbf{A} \xleftarrow{\$} \mathbb{Z}_q^{n \times m}$, $\mathbf{W} = \mathbf{A}\mathbf{Z} \in \mathbb{Z}_q^{n \times \ell_{\mathsf{max}}}$, $\mathbf{U}_0 = \left[\boldsymbol{u}_1 \| \boldsymbol{u}_2 \| \ldots \| \boldsymbol{u}_{\ell_0}\right] \in \{-1,1\}^{\ell_{\mathsf{max}} \times \ell_0}$ with $\mathcal{H}(I_0^*[i]) = \boldsymbol{u}_i \in \{-1,1\}^{\ell_{\mathsf{max}}}$ for all $I_0^*[i] \in I_0^*$ and $\mathbf{U}_1 = \left[\boldsymbol{u}_1' \| \boldsymbol{u}_2' \| \ldots \| \boldsymbol{u}_{\ell_0}'\right] \in \{-1,1\}^{\ell_{\mathsf{max}} \times \ell_1}$ with $\mathcal{H}(I_1^*[i]) = \boldsymbol{u}_i' \in \{-1,1\}^{\ell_{\mathsf{max}}}$ for all $I_1^*[i] \in I_1^*$. The LWE challenger selects column vector $\boldsymbol{d} \xleftarrow{\$} \mathbb{Z}_q^n, \boldsymbol{g} = (\boldsymbol{g}_1, \boldsymbol{g}_2^{(0)}, \boldsymbol{g}_2^{(1)}) \xleftarrow{\$} \mathfrak{D}_{\mathbb{Z}^{m+\ell_0+\ell_1},\psi}$ to produce the tuple $\boldsymbol{z} = (\boldsymbol{c}_1, \boldsymbol{c}_2^{(0)}, \boldsymbol{c}_2^{(1)})$ from the matrix $\mathbf{E}$. $\mathbf{E}$ is a random matrix as $\mathcal{H}$ is modelled as the RO.

Note that, $\mathbf{E}$ is a random matrix since $\mathcal{H}$ is modelled here as the RO.

– For any $I \notin \{I_0^*, I_1^*\}$ query to the RO $\mathcal{H}$, the challenger selects a matrix $\mathbf{U}_I \xleftarrow{\$} \{-1,1\}^{\ell_{\mathsf{max}} \times \ell}$ with $|I| = \ell$ and in order to program the RO $\mathcal{H}$ so that the output $\mathbf{U}_I$ is calculated as $\mathbf{U}_I = [\boldsymbol{u}_1 \| \boldsymbol{u}_2 \| \ldots \| \boldsymbol{u}_{|I|}]$ with $\boldsymbol{u}_i = \mathcal{H}(I[i]), \forall I[i] \in I$. The adversary obtains the decryption key $\mathsf{sk}_{\boldsymbol{y}^{(j)}} = (\boldsymbol{y}^{(j)}, (\boldsymbol{y}^{(j)})^\top \mathbf{U}_I^\top \mathbf{Z}^\top)$ corresponding to the vector $\boldsymbol{y}^{(j)} = (y_i^{(j)})_{i \in I}$. To response the challenge query for the vector $\boldsymbol{x}^{(\beta)} = (x_i^{(\beta)})_{i \in I_\beta^*}$, the challenger now selects a random bit $\beta \xleftarrow{\$} \{0,1\}$. $\mathcal{C}$ gives back $\mathsf{ct}_{\boldsymbol{x}^{(0)}} = (\boldsymbol{c}_1, \boldsymbol{c}_2^{(0)} + \lfloor \frac{q}{K} \rfloor \boldsymbol{x}^{(0)})$ for $\beta = 0$ and $\mathsf{ct}_{\boldsymbol{x}^{(1)}} = (\boldsymbol{c}_1, \boldsymbol{c}_2^{(1)} + \lfloor \frac{q}{K} \rfloor \boldsymbol{x}^{(1)})$ for $\beta = 1$.

The distribution of the challenge ciphertext is $\mathsf{Ct}_{\boldsymbol{x}^{(\beta)}} = (\boldsymbol{c}_1 = \mathbf{A}^\top \boldsymbol{d} + \boldsymbol{g}_1, \boldsymbol{c}_2 = \widetilde{\boldsymbol{c}} + \lfloor \frac{q}{K} \rfloor \boldsymbol{x}^{(\beta)})$ when the LWE oracle produces a random pair $(\mathbf{W}\mathbf{U}_\beta, \widetilde{\boldsymbol{c}})$ where $\mathbf{W} \in \mathbb{Z}_q^{n \times \ell_{\mathsf{max}}}$, $\mathbf{U}_\beta \xleftarrow{\$} \{-1,1\}^{\ell_{\mathsf{max}} \times \ell}$ and $\widetilde{\boldsymbol{c}} \xleftarrow{\$} \mathbb{Z}_q^\ell$. This is same as in Game 1. When the pair $(\mathbf{W}\mathbf{U}_\beta, \boldsymbol{c}_2^{(\beta)})$ is produced by the LWE oracle, with $\mathbf{W} \in \mathbb{Z}_q^{n \times \ell_{\mathsf{max}}}$, $\mathbf{U}_\beta \xleftarrow{\$} \{-1,1\}^{\ell_{\mathsf{max}} \times \ell}$ and $\boldsymbol{c}_2^{(\beta)} = \mathbf{W}\mathbf{U}_\beta \boldsymbol{d} + \boldsymbol{g}^{(\beta)}$ by calculating

$$\begin{aligned} \boldsymbol{c} = \mathbf{E}^\top \boldsymbol{d} + \boldsymbol{g} &= \begin{bmatrix} \mathbf{A} & \mathbf{W}\mathbf{U}_0 & \mathbf{W}\mathbf{U}_1 \end{bmatrix}^\top \boldsymbol{d} + (\boldsymbol{g}_1, \boldsymbol{g}_2^{(0)}, \boldsymbol{g}_2^{(1)}) \\ &= (\mathbf{A}^\top \boldsymbol{d} + \boldsymbol{g}_1, \mathbf{U}_0^\top \mathbf{W}^\top \boldsymbol{d} + \boldsymbol{g}_2^{(0)}, \mathbf{U}_1^\top \mathbf{W}^\top \boldsymbol{d} + \boldsymbol{g}_2^{(1)}) \\ &= (\boldsymbol{c}_1, \boldsymbol{c}_2^{(0)}, \boldsymbol{c}_2^{(1)}) \in \mathbb{Z}_q^m \times \mathbb{Z}_q^{\ell_0} \times \mathbb{Z}_q^{\ell_1}. \end{aligned}$$

The challenge ciphertext's distribution is $\mathsf{Ct}_{\boldsymbol{x}^{(\beta)}} = (\boldsymbol{c}_1 = \mathbf{A}^\top \boldsymbol{d} + \boldsymbol{g}_1, \boldsymbol{c}_2 = \boldsymbol{c}_2^{(\beta)} + \lfloor \frac{q}{K} \rfloor \boldsymbol{x}^{(\beta)}) \in \mathbb{Z}_q^m \times \mathbb{Z}_q^{\ell_\beta}$ which is same as in Game 0. Consequently, we may write as follows:

$$|\mathsf{Adv}_{\mathcal{A},\mathrm{Game}_1}(\lambda) - \mathsf{Adv}_{\mathcal{A},\mathrm{Game}_0}(\lambda)| \leq \mathsf{Adv}_{\mathcal{B}}^{\mathsf{LWE}_{q,\alpha,n}}(\lambda).$$

Consequently, it can be said that

$$\mathsf{Adv}_{\mathcal{A}}^{\mathsf{Adp\text{-}IND}}(\lambda) \leq \frac{1}{Q_{\mathsf{ro}}^2} \cdot \mathsf{Adv}_{\mathcal{B}}^{\mathsf{LWE}_{q,\alpha,n}}(\lambda)$$

where Q_{ro} is the total number of queries that RO has been asked. □

5 Implementation Result

We have implemented the uIPFE [1] and our uIPFE #1 in programming language C and run it on IBM Power9, model 9006-22P. Our trapdoor sampling implementation

is based on [13] and pre-image sampling implementation is based on [16] and are independent of any external sources like MPFR or GMP. Multi-threading is used to make the implementation fast. For instance, we run the procedure SamplePre parallelly using loop parallelizsation inside the procedure SamplePreimage that reduces the runtime. SamplePreimage takes almost the same time as a single execution of the procedure SamplePre while running on a machine with multi-core via multi-threading using OpenMP. More specifically, we have used OpenMP API for multi-threading on IBM POWER9 with 32 cores rated at 3.8GHz. The same concept of loop parallelization is used for the procedure GSampling. It is evident that the overall runtime of uIPFE is primarily influenced by the TrapdoorGen and SamplePreimage algorithms. In contrast, our proposed scheme, uIPFE #1, eliminates the need for both trapdoor generation and preimage sampling, resulting in a considerably faster execution and superior overall performance compared to uIPFE. However, the encryption phase in uIPFE #1 is marginally slower due to the presence of additional multiplication operations. The implementation code is available in https://github.com/uIPFE-implementation.

Table 3: Representative parameter sets for uIPFE and uIPFE #1

Parameter set	**I**	**II**	**III**	**IV**
n	5	5	10	20
m (uIPFE)	530	480	980	1760
m (uIPFE #1)	640	570	1160	2040
ℓ_{max}	5000	1000	800	100
X	8	8	8	8
Y	8	8	8	8
q (uIPFE)	$4503599627370517(\approx 2^{53})$	$140737488355333(\approx 2^{48})$	$281474976710677(\approx 2^{49})$	$8796093022237(\approx 2^{44})$
q (uIPFE #1)	$\approx 2^{64}$	$14411518807585581(\approx 2^{57})$	$288230376151711813(\approx 2^{58})$	$1125899906842679(\approx 2^{51})$
ψ	2.236	2.236	3.162	4.472
α (uIPFE)	2.587×10^{-15}	1.424×10^{-13}	8.952×10^{-14}	4.505×10^{-12}
α (uIPFE #1)	6.169×10^{-19}	1.344×10^{-16}	8.374×10^{-17}	3.330×10^{-14}
σ	83.29	83.29	83.29	83.29

Figures 5, 6, 7 and 8 illustrate the performance comparison between the uIPFE scheme proposed by Dutta et al. [1] and our proposed uIPFE #1, corresponding to the parameter sets I, II, III and IV, respectively, as listed in Table 3. From these figures 5 - 8, it is evident that the overall runtime of the scheme in [1] is higher than that of our uIPFE #1. In particular, our key extraction and decryption algorithms execute significantly faster, whereas the encryption algorithm of [1] performs slightly better than that of our implementation.

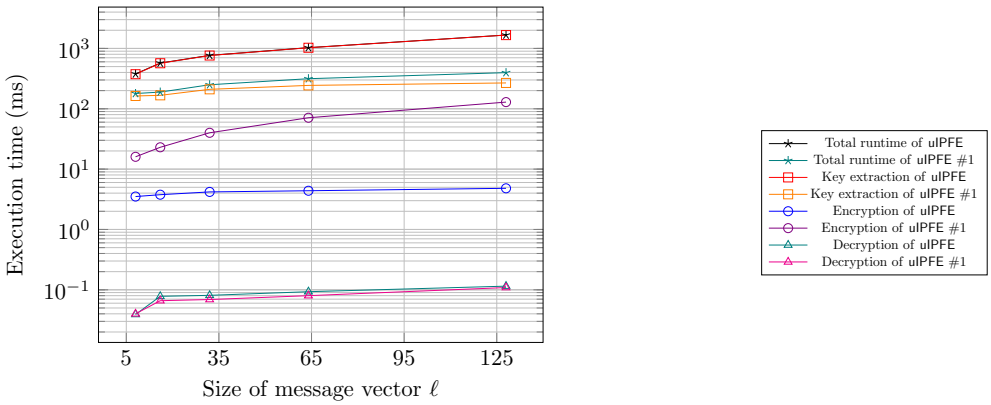

Fig. 5: Execution time comparison of uIPFE.KeyExtract, uIPFE.Encrypt, uIPFE.Decrypt and total runtime of uIPFE [1] and uIPFE #1 for $\ell = 8, 16, 32, 64, 128$ with the choice of parameter set **I** given in Table 3

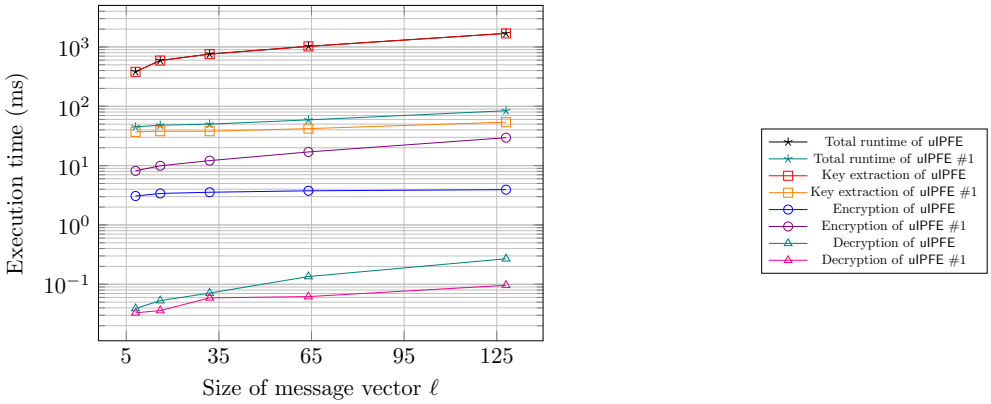

Fig. 6: Execution time comparison of uIPFE.KeyExtract, uIPFE.Encrypt, uIPFE.Decrypt and total runtime of uIPFE [1] and uIPFE #1 for $\ell = 8, 16, 32, 64, 128$ with the choice of parameter set **II** given in Table 3

6 Conclusion

This work presents the implementation of the scheme uIPFE of Dutta et al. [1] and construction of a new scheme strict uIPFE #1 without employing any pairing operations. Our scheme uIPFE #1 achieves post-quantum security against adaptive adversaries using a random oracle without using trapdoor generation and pre-image sampling algorithms. From a practical perspective, this primitive holds greater significance in scenarios where the vectors' length is not known beforehand. Thus, our implementation of the uIPFE of Dutta et al. [1] and our construction of uIPFE #1 are capable of encrypting or decrypting vectors of any size using the same public parameters. In

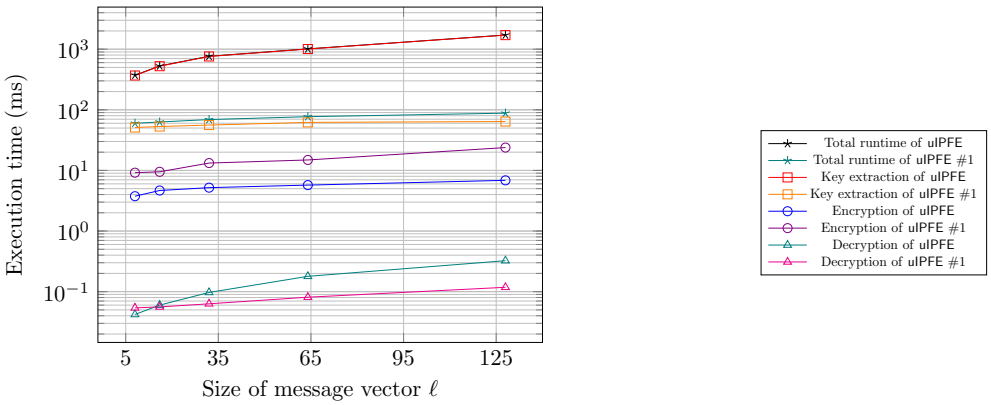

Fig. 7: Execution time comparison of uIPFE.KeyExtract, uIPFE.Encrypt, uIPFE.Decrypt and total runtime of uIPFE [1] and uIPFE #1 for $\ell = 8, 16, 32, 64, 128$ with the choice of parameter set **III** given in Table 3

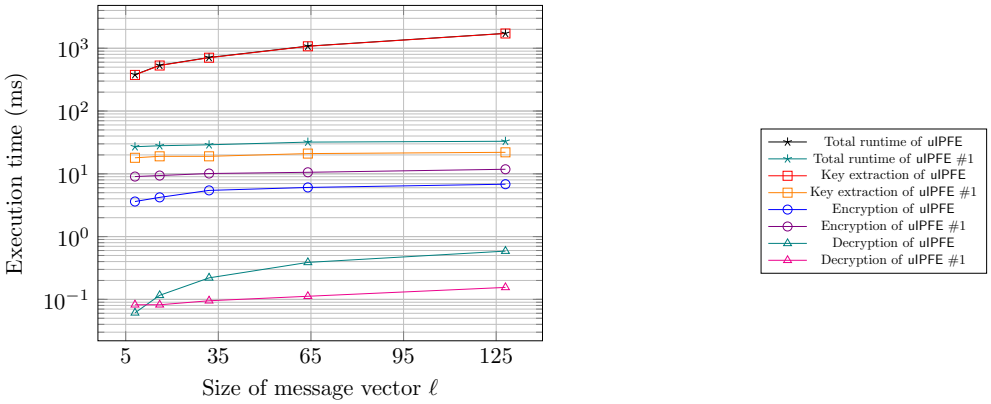

Fig. 8: Execution time comparison of uIPFE.KeyExtract, uIPFE.Encrypt, uIPFE.Decrypt and total runtime of uIPFE [1] and uIPFE #1 for $\ell = 8, 16, 32, 64, 128$ with the choice of parameter set **IV** given in Table 3

addition, we evaluate both the schemes uIPFE and uIPFE #1 based on their execution time and give the implementation outcome. However, constructing such a uIPFE for multi-user systems where multiple users may collide, remains an open problem. Additionally, designing the uIPFE within the standard model based on post-quantum assumptions may pose a significant challenge.

Acknowledgements. The work was partially supported by the Space Applications Centre (SAC), Indian Space Research Organisation (ISRO), (Project ID: STC0322) of the Government of India. We express our sincere gratitude to Subhranil Dutta and Jayashree Dey for their involvement on this work.

Declarations

- Funding : The work received partial financial support from the Space Applications Centre (SAC), Indian Space Research Organisation (ISRO), Government of India, under Project ID: STC0322.
- Conflict of interest/Competing interests : The authors confirm that there are no financial or personal relationships that could have inappropriately influenced the results or interpretations presented in this paper.
- Ethics approval and consent to participate : The authors further confirm that the results, data and figures presented in this manuscript are original, have not been published previously and are not under review by any other journal or publisher.
- Consent for publication : All authors have reviewed the manuscript and consented to its submission in the present form.
- Data availability : No data was utilized for the study outlined in the article.
- Materials availability : Not applicable
- Code availability : The implementation code is available in https://github.com/uIPFE-implementation.
- Author contribution : **Suprava Roy**: Writing - original draft, Writing review & editing, Conceptualization, Formal analysis, Methodology. **Ratna Dutta**: Writing review & editing, Supervision, Formal analysis, Methodology, Conceptualization.

References

[1] Dutta, S., Dutta, R., Mukhopadhyay, S.: Constructing pairing free unbounded inner product functional encryption schemes with unbounded inner product policy. In: International Conference on Information Technology and Communications Security, pp. 102–116 (2022). Springer

[2] Agrawal, S., Libert, B., Stehlé, D.: Fully secure functional encryption for inner products, from standard assumptions. In: Annual International Cryptology Conference, pp. 333–362 (2016). Springer

[3] Boneh, D., Sahai, A., Waters, B.: Functional encryption: Definitions and challenges. In: Theory of Cryptography Conference, pp. 253–273 (2011). Springer

[4] Abdalla, M., Bourse, F., De Caro, A., Pointcheval, D.: Simple functional encryption schemes for inner products. In: IACR International Workshop on Public Key Cryptography, pp. 733–751 (2015). Springer

[5] Benhamouda, F., Bourse, F., Lipmaa, H.: Cca-secure inner-product functional encryption from projective hash functions. In: Fehr, S. (ed.) Public-Key Cryptography – PKC 2017, pp. 36–66. Springer, Berlin, Heidelberg (2017)

[6] Abdalla, M., Catalano, D., Gay, R., Ursu, B.: Inner-product functional encryption with fine-grained access control. In: International Conference on the Theory and Application of Cryptology and Information Security, pp. 467–497 (2020). Springer

[7] Abdalla, M., Gay, R., Raykova, M., Wee, H.: Multi-input inner-product functional encryption from pairings. In: Annual International Conference on the Theory and Applications of Cryptographic Techniques, pp. 601–626 (2017). Springer

[8] Datta, P., Okamoto, T., Tomida, J.: Full-hiding (unbounded) multi-input inner product functional encryption from the k-linear assumption. In: Abdalla, M., Dahab, R. (eds.) Public-Key Cryptography – PKC 2018, pp. 245–277. Springer, Cham (2018)

[9] Tomida, J., Takashima, K.: Unbounded inner product functional encryption from bilinear maps. In: Advances in Cryptology–ASIACRYPT 2018: 24th International Conference on the Theory and Application of Cryptology and Information Security, Brisbane, QLD, Australia, December 2–6, 2018, Proceedings, Part II 24, pp. 609–639 (2018). Springer

[10] Dufour-Sans, E., Pointcheval, D.: Unbounded inner-product functional encryption with succinct keys. In: International Conference on Applied Cryptography and Network Security, pp. 426–441 (2019). Springer

[11] Dutta, S., Dutta, R., Mukhopadhyay, S.: Securing data in the cloud using pairing-free inner product functional encryption with unbounded vector size. Theoretical Computer Science **1031**, 115085 (2025)

[12] Dowerah, U., Dutta, S., Hartmann, F., Mitrokotsa, A., Mukherjee, S., Pal, T.: Sacfe: Secure access control in functional encryption with unbounded data. In: 2024 IEEE 9th European Symposium on Security and Privacy (EuroS&P), pp. 860–882 (2024). IEEE

[13] Gentry, C., Peikert, C., Vaikuntanathan, V.: Trapdoors for hard lattices and new cryptographic constructions. In: Proceedings of the Fortieth Annual ACM Symposium on Theory of Computing, pp. 197–206 (2008)

[14] Micciancio, D., Peikert, C.: Trapdoors for lattices: Simpler, tighter, faster, smaller. In: Annual International Conference on the Theory and Applications of Cryptographic Techniques, pp. 700–718 (2012). Springer

[15] Bert, P., Eberhart, G., Prabel, L., Roux-Langlois, A., Sabt, M.: Implementation of lattice trapdoors on modules and applications. In: Post-Quantum Cryptography: 12th International Workshop, PQCrypto 2021, Daejeon, South Korea, July 20–22, 2021, Proceedings 12, pp. 195–214 (2021). Springer

[16] Gür, K.D., Polyakov, Y., Rohloff, K., Ryan, G.W., Savas, E.: Implementation and evaluation of improved gaussian sampling for lattice trapdoors. In: Proceedings of the 6th Workshop on Encrypted Computing & Applied Homomorphic Cryptography, pp. 61–71 (2018)

[17] Ducas, L., Nguyen, P.Q.: Faster gaussian lattice sampling using lazy floating-point arithmetic. In: International Conference on the Theory and Application of Cryptology and Information Security, pp. 415–432 (2012). Springer

[18] Genise, N., Micciancio, D.: Faster gaussian sampling for trapdoor lattices with arbitrary modulus. In: Advances in Cryptology–EUROCRYPT 2018: 37th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Tel Aviv, Israel, April 29-May 3, 2018 Proceedings, Part I 37, pp. 174–203 (2018). Springer

[19] Agrawal, S., Boneh, D., Boyen, X.: Efficient lattice (h) ibe in the standard model. In: Annual International Conference on the Theory and Applications of Cryptographic Techniques, pp. 553–572 (2010). Springer

[20] Agrawal, S., Boneh, D., Boyen, X.: Lattice basis delegation in fixed dimension and shorter-ciphertext hierarchical ibe. In: Advances in Cryptology–CRYPTO 2010: 30th Annual Cryptology Conference, Santa Barbara, CA, USA, August 15-19, 2010. Proceedings 30, pp. 98–115 (2010). Springer

[21] Micciancio, D., Regev, O.: Worst-case to average-case reductions based on gaussian measures. SIAM journal on computing **37**(1), 267–302 (2007)

[22] Micciancio, D., Regev, O.: Worst-case to average-case reductions based on gaussian measures. SIAM Journal on Computing **37**(1), 267–302 (2007)

[23] Luo, F., Wang, F., Wang, K., Chen, K.: A more efficient leveled strongly-unforgeable fully homomorphic signature scheme. Information Sciences **480**, 70–89 (2019)

[24] Regev, O.: On lattices, learning with errors, random linear codes, and cryptography, 2005. In: STOC, pp. 84–93 (2005). ACM

[25] Peikert, C., *et al.*: A decade of lattice cryptography. Foundations and trends® in theoretical computer science **10**(4), 283–424 (2016)

[26] Zheng, Z., Tian, K., Liu, F.: Fully Homomorphic Encryption, pp. 143–174. Springer, Singapore (2023). https://doi.org/10.1007/978-981-19-7644-5_6