Chapter 2
Homomorphic Public Key Encryption Techniques

In order to attain privacy-preserving data aggregation in smart grid communications, we need to first understand the homomorphic public key encryption (HPKE) techniques [1–7]. Different from the general public key encryption algorithms [8, 9], HPKE further holds an additional "homomorphic" property, which makes the privacy-preserving data aggregation possible. As shown in Fig. 2.1, when we directly operate over two encrypted data $E(x)$ and $E(y)$ with some operation "$\bullet$", we can gain $E(x \circ y) = E(x) \bullet E(y)$. In most HPKE cases, the operations "$\bullet$" and "$\circ$" are respectively referred to the common multiplication "$\times$" and addition "$+$" operations. Because most of privacy-enhancing aggregation techniques illustrated in this monograph are based on Paillier public key encryption [2] and Boneh-Goh-Nissim (BGN) public key encryption [6], in this chapter, we first take a close look at these two popular homomorphic encryption techniques. Note that, both of them are randomized encryption algorithms [1], i.e., in addition to the plaintext as the input of encryption, a random number is also input for achieving semantic security.

2.1 Paillier Public Key Encryption

Paillier Public Key Encryption (PKE) was first proposed in 1999 [2]. Because of its nice "homomorphic" property, Paillier PKE has received considerable attention and has been widely applied in various privacy-preserving computations. In this section, we will briefly recall the famous homomorphic encryption technique. Before that, we first introduce some basic mathematical backgrounds briefly, which may help the reader digest the Paillier PKE better.

R. Lu, *Privacy-Enhancing Aggregation Techniques for Smart Grid Communications*, Wireless Networks, DOI 10.1007/978-3-319-32899-7_2

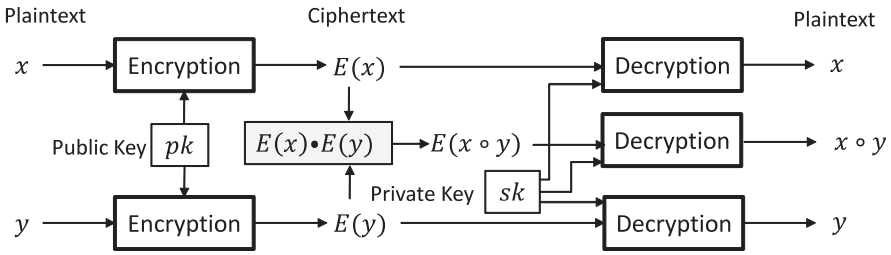

Fig. 2.1 Diagram of homomorphic public key encryption

2.1.1 Mathematical Background

Let $p = 2p' + 1$ and $q = 2q' + 1$ be two safe primes, where p' and q' are also two primes. Compute $n = pq$, we need to prove the following two results:

1. for any $x \in \mathbb{Z}_n$, we have $(1 + n)^x = 1 + x \cdot n \bmod n^2$
2. let $\lambda = lcm(p - 1, q - 1) = 2p'q'$ be the least common multiple of $p - 1$ and $q - 1$. For any $x \in \mathbb{Z}^*_{n^2}$, we have $x^{n\lambda} = 1 \bmod n^2$.

For the first result, we use the following theorem to prove it.

Theorem 2.1. *For any $x \in \mathbb{Z}_n$, we have $(1 + n)^x = 1 + x \cdot n \bmod n^2$.*

Proof. When $x = 0$, the result obviously holds. When $0 < x < n$, we have

$$\begin{aligned}(1+n)^x &= \sum_{i=0}^{x} \binom{x}{i} 1^{n-i} \cdot n^i \bmod n^2 \\ &= 1 + n \cdot x + \binom{x}{2} n^2 + \cdots + \binom{x}{x} n^x \bmod n^2 \\ &= 1 + n \cdot x \bmod n^2\end{aligned} \tag{2.1}$$

As a result, the theorem is correct. ■

For the second result, we should use some lemmas and theorem as below.

Definition 2.1 (Euler Totient Function). Let $P = \prod_i p_i^{l_i}$ with p_i pairwise different primes and $l_i > 0$. Then, Euler Totient Function is defined as

$$\phi(P) = P \cdot \prod_i (1 - \frac{1}{p_i}) \tag{2.2}$$

Lemma 2.1. $\phi(p) = p - 1, \phi(q) = q - 1, \phi(n) = (p - 1)(q - 1), \phi(p^2) = p\phi(p), \phi(q^2) = q\phi(q), \phi(n^2) = n\phi(n)$.

Proof. From the definition of the Euler Totient Function, this lemma can be easily proved. ■

According to the Euler Theorem, for any $x \in \mathbb{Z}_{n^2}^*$, we have $x^{\phi(n^2)} = x^{n\phi(n)} = x^{n\cdot 2\lambda} = 1 \bmod n^2$, but we still cannot determine whether $x^{n\lambda} = 1 \bmod n^2$. In order to obtain $x^{n\lambda} = 1 \bmod n^2$, we need to use the result of the Chinese Remainder Theorem [10].

Theorem 2.2 (Chinese Remainder Theorem). *Suppose that $m_1, m_2, \cdots, m_k$ are pairwise relatively prime positive integers, and let $a_1, a_2, \cdots, a_k$ be integers. Then, the system of congruences, $x \equiv a_i \bmod m_i$ for $1 \le i \le k$, has a unique solution modulo $M = m_1 \times m_2 \times \cdots \times m_k$, which is given by*

$$x \equiv a_1 M_1 y_1 + a_2 M_2 y_2 + \cdots + a_k M_k y_k \bmod M$$

where $M_i = \frac{M}{m_i}$ and $y_i \equiv \frac{1}{M_i} \bmod m_i$ for $1 \le i \le k$.

Let $x \in \mathbb{Z}_{n^2}^*$, from the Euler Theorem, we have

$$\begin{cases} x^{\phi(p^2)} = x^{p(p-1)} = x^{p\cdot 2p'} = 1 \bmod p^2 \Rightarrow x^{pq\cdot 2p'q'} = 1^{qq'} \bmod p^2 \Rightarrow x^{n\lambda} = 1 \bmod p^2 \\ x^{\phi(q^2)} = x^{q(q-1)} = x^{q\cdot 2q'} = 1 \bmod q^2 \Rightarrow x^{pq\cdot 2p'q'} = 1^{pp'} \bmod q^2 \Rightarrow x^{n\lambda} = 1 \bmod q^2 \end{cases} \tag{2.3}$$

Because $\gcd(p^2, q^2) = 1$, we can apply the Extended Euclidean Algorithm to find two integers s, t such that $s \cdot p^2 + t \cdot q^2 = 1$. Let $m_1 = p^2$, $m_2 = q^2$, then $M = m_1 m_2 = n^2$, $M_1 = q^2$, $M_2 = p^2$, $y_1 = t$, and $y_2 = s$. Based on the Chinese Remainder Theorem, where $a_1 = a_2 = 1$, we have

$$x^{n\lambda} = a_1 M_1 y_1 + a_2 M_2 y_2 \bmod M = 1 \cdot q^2 \cdot t + 1 \cdot p^2 \cdot s \bmod n^2 = 1 \bmod n^2 \tag{2.4}$$

Theorem 2.3. *For any $x \in \mathbb{Z}_{n^2}^*$, we have $x^{n\lambda} = 1 \bmod n^2$.*

Definition 2.2 (n-th Residues Modulo n^2). A number $y \in \mathbb{Z}_{n^2}^*$ is said to be an n-th residues modulo n^2 if there exists a number $x \in \mathbb{Z}_{n^2}^*$ such that $y = x^n \bmod n^2$.

Let **NR** be the set of n-th residues modulo n^2. It has been proved that the size of **NR** is exactly $\phi(n)$, i.e., $|\mathbf{NR}| = \phi(n)$. In addition, it has also been proved that "given $y \in \mathbb{Z}_{n^2}^*$, decide whether or not y is n-th residue modulo n^2" is a hard problem, i.e., there does not exist an algorithm that solves the problem in a polynomial time [10].

Let $\mathscr{G} = \{u \in \mathbb{Z}_{n^2}^* | ord(u) = kn, 1 \le k \le \lambda\}$. A special case $g = (1+n)^a \bmod n^2$ with $ord(g) = n$ for some random integer $a \ge 1$ belongs to $\mathscr{G}$, because $g^n = (1+n)^{an} = 1 \bmod n^2$. Define a function $f(m, r) = g^m r^n \bmod n^2$ over $\mathbb{Z}_n \times \mathbb{Z}_n^* \to \mathbb{Z}_{n^2}^*$, we can show that it is a bijective function. First, because $|\mathbb{Z}_n| = n$, $|\mathbb{Z}_n^*| = \phi(n)$, and $|\mathbb{Z}_{n^2}^*| = \phi(n^2) = n\phi(n)$, we have $|\mathbb{Z}_n \times \mathbb{Z}_n^*| = |\mathbb{Z}_{n^2}^*|$, and thus $\mathbb{Z}_n \times \mathbb{Z}_n^* \to \mathbb{Z}_{n^2}^*$ is injective. On the other hand, if for some $(m_0, r_0), (m_1, r_1) \in \mathbb{Z}_n \times \mathbb{Z}_n^*$ with $f(m_0, r_0) = f(m_1, r_1)$, we have

$$\begin{aligned} &g^{m_0} {r_0}^n = g^{m_1} {r_1}^n \bmod n^2 \Rightarrow g^{m_0-m_1} {r_0}^n = {r_1}^n \bmod n^2 \\ \Rightarrow &g^{(m_0-m_1)\lambda} {r_0}^{n\lambda} = {r_1}^{n\lambda} \bmod n^2 \Rightarrow g^{(m_0-m_1)\lambda} = 1 \bmod n^2 \end{aligned} \tag{2.5}$$

which means $ord(g)|(m_0-m_1)\lambda$. By choosing $g = (1+n)^a \bmod n^2$ with $ord(g) = n$, we have $n|(m_0 - m_1)\lambda$. Because $\gcd(n, \lambda) = 1$, we have $m_0 - m_1 = 0 \bmod n$. As a result, we have $m_0 = m_1 \bmod n \Rightarrow m_0 = m_1$. Once we have $m_0 = m_1$, we can further obtain ${r_0}^n = {r_1}^n \bmod n^2$ from $g^{m_0-m_1} {r_0}^n = {r_1}^n \bmod n^2$. It has been easily proved that $f(x) = x^n \bmod n^2$ over $\mathbb{Z}_n^* \rightarrow \mathbf{NR}$ is bijective. Therefore, given ${r_0}^n = {r_1}^n \bmod n^2$, we have $r_0 = r_1$. With this nice bijective function $f(m, r) = g^m r^n \bmod n^2$, the Paillier PKE was proposed [2]. In the following, we describe the details of Paillier PKE.

2.1.2 Description of Paillier PKE

The Paillier PKE can achieve the homomorphic properties, which is mainly comprised of three algorithms: key generation, encryption and decryption.

- *Key Generation:* Given the security parameter κ, two large prime numbers $p = 2p' + 1, q = 2q' + 1$ are first chosen, where $|p| = |q| = \kappa$ and p', q' are also both primes. Then, the RSA modulus $n = pq$ and $\lambda = lcm(p - 1, q - 1) = 2p'q'$ are computed. Define a function $L(u) = \frac{u-1}{n}$, after choosing a generator $g = (1 + n) \in \mathbb{Z}_{n^2}^*$, $\mu = (L(g^\lambda \bmod n^2))^{-1} \bmod n$ is further calculated. Then, the public key is $pk = (n, g)$, and the corresponding private key is $sk = (\lambda, \mu)$.
- *Encryption:* Given a message $m \in \mathbb{Z}_n$, choose a random number $r \in \mathbb{Z}_n^*$, and the ciphertext can be calculated as $c = E(m, r) = g^m \cdot r^n \bmod n^2$.
- *Decryption:* Given the ciphertext $c \in \mathbb{Z}_{n^2}^*$, the corresponding message can be recovered as $m = D(c) = L(c^\lambda \bmod n^2) \cdot \mu \bmod n$.

Correctness.

$$\begin{aligned} m = D(c) &= L(c^\lambda \bmod n^2) \cdot \mu \bmod n = \frac{L(c^\lambda \bmod n^2)}{L(g^\lambda \bmod n^2)} \bmod n \\ &= \frac{L((g^m \cdot r^n)^\lambda \bmod n^2)}{L(g^\lambda \bmod n^2)} \bmod n = \frac{L(g^{m\lambda} \cdot r^{n\lambda} \bmod n^2)}{L(g^\lambda \bmod n^2)} \bmod n \\ &= \frac{L(g^{m\lambda} \bmod n^2)}{L(g^\lambda \bmod n^2)} \bmod n = \frac{L((1+n)^{m\lambda} \bmod n^2)}{L((1+n)^\lambda \bmod n^2)} \bmod n \\ &= \frac{L(1 + m\lambda n \bmod n^2)}{L(1 + \lambda n \bmod n^2)} \bmod n = \frac{m\lambda}{\lambda} \bmod n = m \bmod n = m \end{aligned} \tag{2.6}$$

Security. The Paillier PKE is provably secure against chosen plaintext attack, the detailed security analysis can be referred to [2].

Homomorphic Properties.

1. **Addition.** $E(m_1, r_1) \cdot E(m_2, r_2) = E(m_1 + m_2, r_1 r_2)$

$$E(m_1, r_1) \cdot E(m_2, r_2) = g^{m_1} \cdot {r_1}^n \bmod n^2 \cdot g^{m_2} \cdot {r_2}^n \bmod n^2 = g^{m_1+m_2} \cdot (r_1 r_2)^n \bmod n^2 = E(m_1 + m_2, r_1 r_2)$$

2. **Multiplication.** $E(m_1, r_1)^{m_2} = E(m_1 \cdot m_2, r_1^{m_2})$

$$E(m_1, r_1)^{m_2} = (g^{m_1} \cdot {r_1}^n)^{m_2} \bmod n^2 = g^{m_1 m_2} \cdot {r_1}^{n m_2} \bmod n^2 = E(m_1 m_2, r_1^{m_2})$$

3. **Self-Blinding.** $E(m_1, r_1) \cdot r_2^n \bmod n^2 = E(m_1, r_1 r_2)$

$$E(m_1, r_1) \cdot r_2^n \bmod n^2 = g^{m_1} \cdot r_1 {r_2}^n \bmod n^2 = E(m_1, r_1 r_2)$$

Source Code. The sample java source code of Paillier PKE is available in Appendix 1 in this chapter.

2.2 Boneh-Goh-Nissim (BGN) Public Key Encryption

Paillier PKE is a very popular homomorphic encryption technique, but it does not support the homomorphic multiplication directly over two ciphertexts, which may limit its applications in some scenarios. In order to obtain homomorphic multiplication over two ciphertexts, we recall another famous homomorphic encryption technique - Boneh-Goh-Nissim (BGN) PKE [6] in this section. Before delving into the details, we first recall the bilinear pairing techniques, which serve as the basis of BGN PKE.

2.2.1 Bilinear Pairing Techniques

2.2.1.1 Bilinear Groups of Prime Order

Bilinear pairing is an important cryptographic primitive and has been widely adopted in many positive applications in cryptography [11, 12]. Let $\mathbb{G}$ be a cyclic additive group and $\mathbb{G}_T$ be a cyclic multiplicative group of the same prime order q. We assume that the discrete logarithm problems in both $\mathbb{G}$ and $\mathbb{G}_T$ are hard. A bilinear pairing is a mapping $e : \mathbb{G} \times \mathbb{G} \rightarrow \mathbb{G}_T$ which satisfies the following properties:

1. Bilinearity: for any $P, Q \in \mathbb{G}$ and $a, b \in \mathbb{Z}_q^*$, we have $e(aP, bQ) = e(P, Q)^{ab}$.
2. Non-degeneracy: there exists $P \in \mathbb{G}$ and $Q \in \mathbb{G}$ such that $e(P, Q) \neq 1_{\mathbb{G}_T}$.
3. Computability: there exists an efficient algorithm to compute $e(P, Q)$ for all $P, Q \in \mathbb{G}$.

From Reference [11], we note that such a bilinear pairing may be realized using the modified Weil pairing associated with supersingular elliptic curve.

Definition 2.3 (Bilinear Generator). A bilinear parameter generator $\mathcal{G}en$ is a probability algorithm that takes a security parameter κ as input and outputs a 5-tuple $(q, P, \mathbb{G}, \mathbb{G}_T, e)$, where q is a κ-bit prime number, $(\mathbb{G}, +)$ and $(\mathbb{G}_T, \times)$ are two groups with the same order q, $P \in \mathbb{G}$ is a generator, and $e : \mathbb{G} \times \mathbb{G} \to \mathbb{G}_T$ is an admissible bilinear map.

In the following, we define the quantitative notion of the complexity assumptions, including Computational Diffie-Hellman (CDH) Problem, Decisional Diffie-Hellman (DDH) Problem, and Bilinear Diffie-Hellman (BDH) Problem.

Definition 2.4 (Computational Diffie-Hellman (CDH) Problem). The Computational Diffie-Hellman (CDH) problem in $\mathbb{G}$ is defined as follows: Given P, aP, $bP \in \mathbb{G}$ for unknown $a, b \in \mathbb{Z}_q^*$, compute $abP \in \mathbb{G}$.

Definition 2.5 (CDH Assumption). Let $\mathscr{A}$ be an adversary that takes an input of $(P, aP, bP) \in \mathbb{G}$ for unknown $a, b \in \mathbb{Z}_q^*$, and returns abP. We consider the following random experiment.

$$
\begin{aligned}
&\texttt{Experiment } \mathbf{Exp}_{\mathscr{A}}^{\mathrm{CDH}} \\
&\quad a, b \overset{R}{\leftarrow} \mathbb{Z}_q, \alpha \leftarrow \mathscr{A}(P, aP, bP) \\
&\quad \texttt{if } \alpha = abP, \texttt{ then } \beta \leftarrow 1, \texttt{ else } \beta \leftarrow 0 \\
&\quad \texttt{return } \beta
\end{aligned}
$$

We define the corresponding success probability of $\mathscr{A}$ in solving the CDH problem via

$$\mathbf{Succ}_{\mathscr{A}}^{\mathrm{CDH}} = \Pr\left[\mathbf{Exp}_{\mathscr{A}}^{\mathrm{CDH}} = 1\right]$$

Let $\tau \in \mathbb{N}$ and $\epsilon \in [0, 1]$. We say that the CDH is (τ, ϵ)-secure if no polynomial algorithm $\mathscr{A}$ running in time τ has success $\mathbf{Succ}_{\mathscr{A}}^{\mathrm{CDH}} \geq \epsilon$.

Definition 2.6 (Decisional Diffie-Hellman (DDH) Problem). For $a, b, c \in \mathbb{Z}_q^*$, given P, aP, bP, $cP \in \mathbb{G}$, decide whether $c = ab \in \mathbb{Z}_q$. The DDH problem is easy in $\mathbb{G}$, since we can compute $e(aP, bP) = e(P, P)^{ab}$ and decide whether $e(P, P)^{ab} = e(P, P)^c$ [11].

Definition 2.7 (Bilinear Diffie-Hellman (BDH) Problem). The Bilinear Diffie-Hellman (BDH) problem in $\mathbb{G}$ is as follows: Given P, aP, bP, $cP \in \mathbb{G}$ for unknown $a, b, c \in \mathbb{Z}_q^*$, compute $e(P, P)^{abc} \in \mathbb{G}_T$.

Definition 2.8 (BDH Assumption). Let $\mathscr{A}$ be an adversary that takes an input of $(P, aP, bP, cP) \in \mathbb{G}$ for unknown $a, b, c \in \mathbb{Z}_q^*$, and returns $e(P,P)^{abc}$. We consider the following random experiment.

$$
\begin{aligned}
&\texttt{Experiment } \mathbf{Exp}_{\mathscr{A}}^{\mathrm{CDH}} \\
&\quad a, b, c \overset{R}{\leftarrow} \mathbb{Z}_q, \alpha \leftarrow \mathscr{A}\,(P, aP, bP, cP) \\
&\quad \texttt{if } \alpha = e(P,P)^{abc} \texttt{ then } \beta \leftarrow 1 \texttt{ else } \beta \leftarrow 0 \\
&\quad \texttt{return } \beta
\end{aligned}
$$

We define the corresponding success probability of $\mathscr{A}$ in solving the BDH problem via

$$\mathbf{Succ}_{\mathscr{A}}^{\mathrm{BDH}} = \Pr\left[\mathbf{Exp}_{\mathscr{A}}^{\mathrm{BDH}} = 1\right]$$

Let $\tau \in \mathbb{N}$ and $\epsilon \in [0,1]$. We say that the BDH is (τ, ϵ)-secure if no polynomial algorithm $\mathscr{A}$ running in time τ has success $\mathbf{Succ}_{\mathscr{A}}^{\mathrm{BDH}} \geq \epsilon$.

Definition 2.9 (Decisional Diffie-Hellman (DBDH) Problem). The Decisional Bilinear Diffie-Hellman (DBDH) problem in $\mathbb{G}$ is as follows: Given an element P of $\mathbb{G}$, a tuple (aP, bP, cP, T) for unknown $a, b, c \in \mathbb{Z}_q^*$ and $T \in \mathbb{G}_T$, decide whether $T = e(P,P)^{abc}$ or a random element R drawn from $\mathbb{G}_T$.

Definition 2.10 (DBDH Assumption). Let $\mathscr{A}$ be an adversary that takes an input of (aP, bP, cP, T) for unknown $a, b, c \in \mathbb{Z}_q^*$ and $T \in \mathbb{G}_T$, and returns a bit $\beta' \in \{0,1\}$. We consider the following random experiments.

$$
\begin{aligned}
&\texttt{Experiment } \mathbf{Exp}_{\mathscr{A}}^{\mathrm{DBDH}} \\
&\quad a, b, c \overset{R}{\leftarrow} \mathbb{Z}_q; R \overset{R}{\leftarrow} \mathbb{G}_T \\
&\quad \tilde{\beta} \leftarrow \{0,1\} \\
&\quad \text{if } \tilde{\beta} = 0, \text{ then } T = e(P,P)^{abc}; \text{ else if } \tilde{\beta} = 1 \text{ then } T = R \\
&\quad \tilde{\beta}' \leftarrow \mathscr{A}(aP, bP, cP, T) \\
&\quad \texttt{return } 1 \text{ if } \tilde{\beta}' = \tilde{\beta}, 0 \text{ otherwise}
\end{aligned}
$$

We then define the advantage of $\mathscr{A}$ via

$$
\begin{aligned}
\mathbf{Adv}_{\mathscr{A}}^{\mathrm{DBDH}} = \Big|&\Pr\left[\mathbf{Exp}_{\mathscr{A}}^{\mathrm{DBDH}} = 1 | \tilde{\beta} = 0\right] \\
&- \Pr\left[\mathbf{Exp}_{\mathscr{A}}^{\mathrm{DBDH}} = 1 | \tilde{\beta} = 1\right]\Big| \geq \epsilon
\end{aligned}
$$

Let $\tau \in \mathbb{N}$ and $\epsilon \in [0,1]$. We say that the DBDH is (τ, ϵ)-secure if no adversary $\mathscr{A}$ running in time τ has an advantage $\mathbf{Adv}_{\mathscr{A}}^{\mathrm{DBDH}} \geq \epsilon$.

2.2.1.2 Bilinear Groups of Composite Order

Let p, q be two distinct large primes, and $n = pq$. Groups $(\mathbb{G}, \mathbb{G}_T)$ of composite order n are called *bilinear map groups of composite order* if there is a mapping $e : \mathbb{G} \times \mathbb{G} \to \mathbb{G}_T$ with the following properties [6, 13]:

1. Bilinearity: $e(g^a, h^b) = e(g, h)^{ab}$ for any $(g, h) \in \mathbb{G}^2$ and $a, b \in \mathbb{Z}_n$.
2. Non-degeneracy: there exists $g \in \mathbb{G}$ such that $e(g, g)$ has order n in $\mathbb{G}_T$. In other words, $e(g, g)$ is a generator of $\mathbb{G}_T$, whereas g generates $\mathbb{G}$.
3. Computability: there exists an efficient algorithm to compute $e(g, h) \in \mathbb{G}_T$ for all $(g, h) \in \mathbb{G}$.

Note that 1) we use the multiplicative group to represent the group $\mathbb{G}$, which, however, can be instantiated by the elliptic curve addition group, i.e., the modified Weil pairing or Tate pairing [6, 13]; 2) the vast majority of cryptosystems based on pairings assume for simplicity that bilinear groups have prime order q. In composite order case, it is important that the pairing is defined over a group $\mathbb{G}$ containing $|\mathbb{G}| = n$ elements, where $n = pq$ has a (ostensibly hidden) factorization in two large primes, $p \neq q$; 3) those complexity assumptions above in bilinear group of prime order also hold in bilinear group of composite order.

Definition 2.11 (Composite Bilinear Generator). A composite bilinear parameter generator $\mathscr{CG}en$ is a probabilistic algorithm that takes a security parameter k as input, and outputs a 5-tuple $(n, g, \mathbb{G}, \mathbb{G}_T, e)$, where $n = pq$ and p, q are two k-bit prime numbers, $\mathbb{G}, \mathbb{G}_T$ are two groups with order n, $g \in \mathbb{G}$ is a generator, and $e : \mathbb{G} \times \mathbb{G} \to \mathbb{G}_T$ is a non-degenerated and efficiently computable bilinear map.

Let $\mathbf{g}$ be a generator of $\mathbb{G}$, then $g = \mathbf{g}^q \in \mathbb{G}$ can generate the subgroup $\mathbb{G}_p = \{g^0, g^1, \cdots, g^{p-1}\}$ of order p, and $g' = \mathbf{g}^p \in \mathbb{G}$ can generate the subgroup $\mathbb{G}_q = \{g'^0, g'^1, \cdots, g'^{q-1}\}$ of order q in $\mathbb{G}$. In the following, we define the quantitative notion of the complexity of the SubGroup Decision (SGD) Problem [6].

Definition 2.12 (SubGroup Decision (SGD) Problem). The SubGroup Decision (SGD) problem in $\mathbb{G}$ is as follows: Given a tuple $(e, \mathbb{G}, \mathbb{G}_T, n, h)$, where the element h is randomly drawn from either $\mathbb{G}$ or subgroup $\mathbb{G}_q$, decide whether or not $h \in \mathbb{G}_q$.

Definition 2.13 (SGD Assumption). Let $\mathscr{A}$ be an adversary that takes an input of h drawn from either $\mathbb{G}$ or subgroup $\mathbb{G}_q$, and returns a bit $b' \in \{0, 1\}$. We consider the following random experiments.

$$
\begin{aligned}
&\texttt{Experiment}\ \mathbf{Exp}^{\mathrm{SGD}}_{\mathscr{A}} \\
&\quad \tilde{b} \leftarrow \{0, 1\} \\
&\quad \text{if } \tilde{b} = 0, \text{ then } h \overset{R}{\leftarrow} \mathbb{G}_q; \text{ else if } \tilde{b} = 1 \text{ then } h \overset{R}{\leftarrow} \mathbb{G} \\
&\quad \tilde{b}' \leftarrow \mathscr{A}(e, \mathbb{G}, \mathbb{G}_T, n, h) \\
&\quad \texttt{return}\ 1 \text{ if } \tilde{b}' = \tilde{b}, 0 \text{ otherwise}
\end{aligned}
$$

We then define the advantage of $\mathscr{A}$ via

$$\begin{aligned}\mathbf{Adv}_{\mathscr{A}}^{\mathrm{SGD}} = &\left|\Pr\left[\mathbf{Exp}_{\mathscr{A}}^{\mathrm{SGD}} = 1|\tilde{b} = 0\right]\right. \\ &\left. - \Pr\left[\mathbf{Exp}_{\mathscr{A}}^{\mathrm{SGD}} = 1|\tilde{b} = 1\right]\right| \geq \epsilon\end{aligned}$$

Let $\tau \in \mathbb{N}$ and $\epsilon \in [0, 1]$. We say that the SGD is (τ, ϵ)-secure if no adversary $\mathscr{A}$ running in time τ has an advantage $\mathbf{Adv}_{\mathscr{A}}^{\mathrm{SGD}} \geq \epsilon$.

2.2.2 Description of BGN PKE

BGN PKE [6] can achieve one more homomorphic property in comparison to the Paillier PKE, which mainly consists of three algorithms: key generation, encryption, and decryption.

- *Key Generation:* Given the security parameter k, composite bilinear parameters $(n, g, \mathbb{G}, \mathbb{G}_T, e)$ are generated by $\mathscr{GG}en(k)$, where $n = pq$ and p, q are two k-bit prime numbers, and $g \in \mathbb{G}$ is a generator of order n. Set $h = g^q$, then h is a random generator of the subgroup of $\mathbb{G}$ of order p. The public key is $pk = (n, \mathbb{G}, \mathbb{G}_T, e, g, h)$, and the corresponding private key is $sk = p$.
- *Encryption:* We assume the message space consists of integers in the set $\{0, 1, \cdots, T\}$ with $T < q$. To encrypt a message m, we choose a random number $r \in \mathbb{Z}_n$, and compute the ciphertext $c = E(m, r) = g^m h^r \in \mathbb{G}$.
- *Decryption:* Given the ciphertext $c = E(m, r) = g^m h^r \in \mathbb{G}$, the corresponding message can be recovered by the private key p. Observe that $c^p = (g^m h^r)^p = (g^p)^m$. Let $\hat{g} = g^p$. To recover m, it suffices to compute the discrete log of c^p base $\hat{g}$. Since $0 \leq m \leq T$, the expected time is around $O(\sqrt{T})$ when using the Pollard's lambda method [14](p. 128).

Security. BGN PKE is provably secure against chosen plaintext attack based on the subgroup decision assumption, the detailed security analysis can be referred to [6].

Homomorphic Properties.

1. **Addition.** $E(m_1, r_1) \cdot E(m_2, r_2) = E(m_1 + m_2, r_1 + r_2)$

$$E(m_1, r_1) \cdot E(m_2, r_2) = g^{m_1} h^{r_1} \cdot g^{m_2} h^{r_2} = g^{m_1+m_2} \cdot h^{r_1+r_2} = E(m_1 + m_2, r_1 + r_2)$$

2. **Multiplication.** $E(m_1, r_1)^{m_2} = E(m_1 \cdot m_2, r_1 \cdot m_2)$

$$E(m_1, r_1)^{m_2} = (g^{m_1} \cdot h^{r_1})^{m_2} = g^{m_1 m_2} \cdot h^{r_1 m_2} = E(m_1 m_2, r_1 m_2)$$

3. **Self-Blinding.** $E(m_1, r_1) \cdot h^{r_2} = E(m_1, r_1 + r_2)$

$$E(m_1, r_1) \cdot h^{r_2} = g^{m_1} \cdot h^{r_1+r_2} = E(m_1, r_1 + r_2)$$

4. **Multiplication-II.** $e(E(m_1, r_1), E(m_2, r_2)) = E'(m_1 \cdot m_2, m_1 r_2 + r_1 m_2 + q r_1 r_2)$

$$C = e(E(m_1, r_1), E(m_2, r_2)) = e(g^{m_1} h^{r_1}, g^{m_2} h^{r_2}) = e(g, g)^{m_1 m_2} \cdot e(g, h)^{m_1 r_2 + r_1 m_2 + q r_1 r_2}$$
$$= E'(m_1 \cdot m_2, m_1 r_2 + r_1 m_2 + q r_1 r_2)$$

Observe that $C^p = (e(g, g)^{m_1 m_2} \cdot e(g, h)^{m_1 r_2 + r_1 m_2 + q r_1 r_2})^p = (e(g, g)^p)^{m_1 m_2}$. Let $\overline{g} = e(g, g)^p$. To recover $m_1 m_2$, it suffices to compute the discrete log of C^p base $\overline{g}$ by using the Pollard's lambda method [14](p. 128). Note that the homomorphic multiplication can be taken only *once* upon two ciphertexts in $\mathbb{G}$, and then the result will be in $\mathbb{G}_T$, but it still supports additive homomorphism. Also note that, if we do not expect the **Multiplication-II** homomorphic property in some scenarios, we do not need to use bilinear groups of composite order, and can simply build the BGN PKE over the general group $(\mathbb{G}, \times)$ with composite order $n = pq$.

Source Code. The sample java source code of BGN PKE is available in Appendix 2 in this chapter.

2.3 Summary

In this chapter, we have discussed two popular homomorphic encryption techniques Paillier PKE [2] and BGN PKE [6], which will be used in the design of most privacy-preserving aggregation schemes in this monograph. Note that, the fully homomorphic encryption techniques [15–21] can also be applied in privacy-preserving data aggregation in smart grid communications [22]. However, the efficiency needs to be extensively exploited in practical scenarios. Therefore, in this monograph, the fully homomorphic encryption techniques are not our focuses, interested readers can refer to [15–21] for more details.

Appendix 1: A Sample Java Source Code of Paillier PKE

```
import java.math.BigInteger;
import java.security.SecureRandom;

/**
 * @ClassName: Paillier
 * @Description: This is a sample java source code of Paillier
```

(continued)

```
 * PKE.
 */
public class Paillier {

    /**
     * @ClassName: PublicKey
     * @Description: This is a class for storing the public
     *               key (n, g) of Paillier PKE.
     */
    public class PublicKey {
        private BigInteger n, g;

        public PublicKey(BigInteger n, BigInteger g) {
            this.n = n;
            this.g = g;
        }

        public BigInteger getN() {
            return n;
        }

        public BigInteger getG() {
            return g;
        }
    }

    /**
     * @ClassName: PrivateKey
     * @Description: This is a class for storing the private
     *               key (lambda, mu) of Paillier PKE.
     */
    public class PrivateKey {
        private BigInteger lambda, mu;

        public PrivateKey(BigInteger lambda, BigInteger mu) {
            this.lambda = lambda;
            this.mu = mu;
        }

        public BigInteger getLambda() {
            return lambda;
        }

        public BigInteger getMu() {
            return mu;
        }
    }

    private final int CERTAINTY = 64;
```

(continued)

```
private PublicKey pubkey; // The public key of Paillier
                             PKE, (n, g)
private PrivateKey prikey; // The private key of Paillier
                              PKE, (lambda, mu)
/**
 * @Title: getPubkey
 * @Description: This function returns the generated
 *               public key.
 * @return PublicKey The public key used to encrypt
 * the data.
 */
public PublicKey getPubkey() {
    return pubkey;
}

/**
 * @Title: getPrikey
 * @Description: This function returns the generated
 *               private key.
 * @return PrivateKey The private key used to decrypt
 * the data.
 */
public PrivateKey getPrikey() {
    return prikey;
}

/**
 * @Title: keyGeneration
 * @Description: This function is to help generate the
 *               public key and
 *               private key for encryption and decryption.
 * @param k
 *            k is the security parameter, which decides
 *            the length of two large primes (p and q).
 * @return void
 */
public void keyGeneration(int k) {

    BigInteger p_prime, q_prime, p, q;

    do {
        p_prime = new BigInteger(k, CERTAINTY,
                  new SecureRandom());
        p = (p_prime.multiply(BigInteger.valueOf(2)))
            .add(BigInteger.ONE);
    } while (!p.isProbablePrime(CERTAINTY));

    do {
        do {
            q_prime = new BigInteger(k, CERTAINTY,
```

(continued)

```
                        new SecureRandom());
            } while (p_prime.compareTo(q_prime) == 0);
            q = (q_prime.multiply(BigInteger.valueOf(2)))
                .add(BigInteger.ONE);
        } while (!q.isProbablePrime(CERTAINTY));

        // The following steps are to generate the keys
        // n=p*q
        BigInteger n = p.multiply(q);
        // nsquare=n^2
        BigInteger nsquare = n.pow(2);
        // a generator g=(1+n) in Z*_(n^2)
        BigInteger g = BigInteger.ONE.add(n);
        // lambda = lcm(p-1, q-1) = p_prime*q_prime
        BigInteger lambda = BigInteger.valueOf(2)
                            .multiply(p_prime)
                .multiply(q_prime);
        // mu = (L(g^lambda mod n^2))^{-1} mod n
        BigInteger mu = Lfunction(g.modPow(lambda, nsquare), n)
                        .modInverse(n);

        pubkey = new PublicKey(n, g);
        prikey = new PrivateKey(lambda, mu);
    }

    /**
     * @Title: encrypt
     * @Description: This function is to encrypt the message
     *               with Paillier's public key.
     * @param m
     *            The message.
     * @param pubkey
     *            The public key of Paillier PKE.
     * @return BigInteger The ciphertext.
     * @throws Exception
     *             If the message is not in Z*_n, there is
     *             an exception.
     */
    public static BigInteger encrypt(BigInteger m,
            PublicKey pubkey) throws Exception {
        BigInteger n = pubkey.getN();
        BigInteger nsquare = n.pow(2);
        BigInteger g = pubkey.getG();
        if (!belongToZStarN(m, n)) {
            throw new Exception(
                    "Paillier.encrypt(BigInteger m, PublicKey
                              pubkey): plaintext m is not
                                       in Z*_n");
        }
        BigInteger r = randomZStarN(n);
```

(continued)

```
        return (g.modPow(m, nsquare).multiply(r.modPow(n,
                nsquare))).mod(nsquare);
    }

    /**
     * @Title: decrypt
     * @Description: This function is to decrypt the ciphertext
     *               with the public key and the private key.
     * @param c
     *            The ciphertext.
     * @param pubkey
     *            The public key of Paillier PKE.
     * @param prikey
     *            The private key of Paillier PKE.
     * @return BigInteger The plaintext.
     * @throws Exception
     *             If the cipher is not in Z*_(n^2), there is
     *             an exception.
     */
    public static BigInteger decrypt(BigInteger c, PublicKey
            pubkey, PrivateKey prikey) throws Exception {
        BigInteger n = pubkey.getN();
        BigInteger nsquare = n.pow(2);
        BigInteger lambda = prikey.getLambda();
        BigInteger mu = prikey.getMu();
        if (!belongToZStarNSquare(c, nsquare)) {
            throw new Exception(
                "Paillier.decrypt(BigInteger c, PrivateKey
                 prikey): ciphertext c is not in Z*_(n^2)");
        }

        return Lfunction(c.modPow(lambda, nsquare), n).
                            multiply(mu).mod(n);
    }

    /**
     * @Title: add
     * @Description: The function supports the homomorphic
     * addition with two ciphertext.
     * @param c1
     *            The ciphertext.
     * @param c2
     *            The ciphertext.
     * @param pubkey
     *            The public key of Paillier PKE.
     * @return BigInteger The return value is c1*c2 mod n^2.
     */
    public static BigInteger add(BigInteger c1, BigInteger c2,
        PublicKey pubkey) {BigInteger nsquare = pubkey.getN()
        .pow(2);
```

(continued)

```
        return c1.multiply(c2).mod(nsquare);
    }

    /**
     * @Title: mul
     * @Description: The function supports the homomorphic
     *  multiplication with one ciphertext and one plaintext.
     * @param c
     *            The ciphertext.
     * @param m
     *            The plaintext.
     * @param pubkey
     *            The public key of Paillier PKE.
     * @return BigInteger The return value is c^m mod n^2.
     */
    public static BigInteger mul(BigInteger c, BigInteger m,
    PublicKey pubkey) {BigInteger nsquare =
                        pubkey.getN().pow(2);
        return c.modPow(m, nsquare);
    }

    /**
     * @Title: selfBlind
     * @Description: The function supports the homomorphic
     * self-blinding with one ciphertext and one random number.
     * @param c
     *            The ciphertext.
     * @param r
     *            A random number in Z*_n.
     * @param pubkey
     *            The public key of Paillier PKE.
     * @return BigInteger The return value is c*r^n mod n^2.
     */
    public static BigInteger selfBlind(BigInteger c,
            BigInteger r, PublicKey pubkey) {
        BigInteger n = pubkey.getN();
        BigInteger nsquare = n.pow(2);
        return c.multiply(r.modPow(n, nsquare)).mod(nsquare);
    }

    /**
     * @Title: Lfunction
     * @Description: This function is the L function which is
     * defined by Paillier PKE, L(mu)=(mu-1)/n.
     * @param mu
     *            The input parameter.
     * @param n
     *            n=p*q.
     * @return BigInteger The return value is (mu-1)/n.
     */
```

(continued)

```
private static BigInteger Lfunction(BigInteger mu,
 BigInteger n) {return
               mu.subtract(BigInteger.ONE).divide(n);
}

/**
 * @Title: randomZStarN
 * @Description: This function returns a ramdom number in
 *               Z*_n.
 * @param n
 *            n=p*q.
 * @return BigInteger A random number in Z*_n.
 */
public static BigInteger randomZStarN(BigInteger n) {
    BigInteger r;
    do {
        r = new BigInteger(n.bitLength(), new
            SecureRandom());
    } while (r.compareTo(n) >= 0 || r.gcd(n).intValue()
     != 1);
     return r;
}

/**
 *
 * @Title: belongToZStarN
 * @Description: This function is to test whether the
 *               plaintext is in Z*_n.
 * @param m
 *            The plaintext.
 * @param n
 *            n=p*q.
 * @return boolean If it is true, the plaintext is Z*_n,
 *         otherwise, not.
 */
private static boolean belongToZStarN(BigInteger m,
    BigInteger n) {
    if (m.compareTo(BigInteger.ZERO) < 0 ||
    m.compareTo(n) >= 0
            || m.gcd(n).intValue() != 1) {
        return false;
    }
    return true;
}

/**
 *
 * @Title: belongToZStarNSquare
 * @Description: This function is to test whether the
 *               ciphertext is in
```

(continued)

```
     *               Z*_(n^2).
     * @param c
     *            The ciphertext.
     * @param nsquare
     *            nsquare=n^2.
     * @return boolean If it is true, the ciphertext is
     *         Z*_(n^2), otherwise, not.
     */
    private static boolean belongToZStarNSquare(BigInteger c,
        BigInteger nsquare){
        if (c.compareTo(BigInteger.ZERO) < 0 ||
        c.compareTo(nsquare) >= 0
                || c.gcd(nsquare).intValue() != 1) {
            return false;
        }
        return true;
    }

    public static void main(String[] args) {
        Paillier paillier = new Paillier();

        // KeyGeneration
        paillier.keyGeneration(512);
        Paillier.PublicKey pubkey = paillier.getPubkey();
        Paillier.PrivateKey prikey = paillier.getPrikey();

        // Encryption and Decryption
        BigInteger m = new BigInteger(new
                       String("Hello").getBytes());
        BigInteger c = null;
        BigInteger decrypted_m = null;
        try {
            c = Paillier.encrypt(m, pubkey);
            decrypted_m = Paillier.decrypt(c, pubkey, prikey);
        } catch (Exception e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        }

        if (decrypted_m.compareTo(m) == 0) {
            System.out.println("Encryption and Decryption
            test successfully.");
        }

        // Homomorphic Properties

        // Addition
        BigInteger m1 = new BigInteger("12345");
        BigInteger m2 = new BigInteger("56789");
        BigInteger m1plusm2 = m1.add(m2);
```

(continued)

```
        try {
            BigInteger c1 = Paillier.encrypt(m1, pubkey);
            BigInteger c2 = Paillier.encrypt(m2, pubkey);
            BigInteger c1mulc2 = Paillier.add(c1, c2, pubkey);
            BigInteger decrypted_c1mulc2 =
                     Paillier.decrypt(c1mulc2, pubkey, prikey);
            if (decrypted_c1mulc2.compareTo(m1plusm2) == 0) {
                System.out.println("Homomorphic addition
                tests successfully.");
            }
        } catch (Exception e) {
            e.printStackTrace();
        }

        // Multiplication
        m1 = new BigInteger("12345");
        m2 = new BigInteger("56789");
        BigInteger m1mulm2 = m1.multiply(m2);
        try {
            BigInteger c1 = Paillier.encrypt(m1, pubkey);
            BigInteger c1expm2 = Paillier.mul(c1, m2, pubkey);
            BigInteger decrypted_c1expm2 =
                    Paillier.decrypt(c1expm2, pubkey, prikey);
            if (decrypted_c1expm2.compareTo(m1mulm2) == 0) {
                System.out
                        .println("Homomorphic multiplication
                         tests successfully.");
            }
        } catch (Exception e) {
            e.printStackTrace();
        }

        // Self-Blinding
        m1 = new BigInteger("12345");
        BigInteger r2 = Paillier.randomZStarN(pubkey.getN());
        try {
            BigInteger c1 = Paillier.encrypt(m1, pubkey);
            BigInteger c1mulrn = Paillier.selfBlind(c1, r2,
                        pubkey);
            BigInteger decrypted_c1mulrn =
                    Paillier.decrypt(c1mulrn, pubkey, prikey);
            if (decrypted_c1mulrn.compareTo(m1) == 0) {
                System.out
                        .println("Homomorphic self-blinding
                         tests successfully.");
            }
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

Appendix 2: A Sample Java Source Code of BGN PKE

```
import java.math.BigInteger;

/*
 * This source code uses the JPBC (Java Pairing-Based
 *      Cryptography) library,
 * which can be downloaded from
 *     http://gas.dia.unisa.it/projects/jpbc/
 */

import it.unisa.dia.gas.jpbc.*;
import it.unisa.dia.gas.plaf.jpbc.pairing.PairingFactory;
import it.unisa.dia.gas.plaf.jpbc.pairing.a1
       .TypeA1CurveGenerator;

/**
 * @ClassName: BGN
 * @Description: This is a sample java source code of BGN PKE.
 */
public class BGN {
    /**
     * @ClassName: PublicKey
     * @Description: This is a class for storing the
     * public key (n,G,GT,e,g,h) of BGN PKE.
     */
    public class PublicKey {
        private BigInteger n;
        private Field<Element> Field_G, Field_GT;
        private Pairing pairing;
        private Element g, h;

        public PublicKey(BigInteger n, Field<Element> G,
               Field<Element> GT, Pairing pairing, Element g,
               Element h) {
            this.n = n;
            this.Field_G = G;
            this.Field_GT = GT;
            this.pairing = pairing;
            this.g = g;
            this.h = h;
        }

        public Element getG() {
            return g;
        }
```

(continued)

```
        public Element getH() {
            return h;
        }

        public BigInteger getN() {
            return n;
        }

        public Pairing getPairing() {
            return pairing;
        }

        public Field<Element> getField_G() {
            return Field_G;
        }

        public Field<Element> getField_GT() {
            return Field_GT;
        }
    }

    /**
     * @ClassName: PrivateKey
     * @Description: This is a class for storing the
     *               private key (p) of BGN PKE.
     */
    public class PrivateKey {
        private BigInteger p;

        public PrivateKey(BigInteger p) {
            this.p = p;
        }

        public BigInteger getP() {
            return p;
        }
    }

    private static final int T = 100; // The max range of
       message m
    private PublicKey pubkey;
    private PrivateKey prikey;

    /**
     * @Title: keyGeneration
     * @Description: This function is responsible for
     * generating the public keys and the private keys.
     * @param k
     *            the security parameter, which decides the
     *            length of two large prime (p and q).
```

(continued)

```
     * @return void
     */
    public void keyGeneration(int k) {
        TypeA1CurveGenerator pg = new
          TypeA1CurveGenerator(2, k);
        PairingParameters pp = pg.generate();
        Pairing pairing = PairingFactory.getPairing(pp);
        BigInteger n = pp.getBigInteger("n");
        BigInteger q = pp.getBigInteger("n0");
        BigInteger p = pp.getBigInteger("n1");
        Field<Element> Field_G = pairing.getG1();
        Field<Element> Field_GT = pairing.getGT();
        Element g = Field_G.newRandomElement().getImmutable();
        Element h = g.pow(q).getImmutable();

        pubkey = new PublicKey(n, Field_G, Field_GT,
                 pairing, g, h);
        prikey = new PrivateKey(p);
    }

    /**
     * @Title: getPubkey
     * @Description: This function returns the public key of
     *               BGN PKE.
     * @return PublicKey The public key used to encrypt
     *               the data.
     */
    public PublicKey getPubkey() {
        return pubkey;
    }

    /**
     * @Title: getPrikey
     * @Description: This function returns the private key of
     *               BGN PKE.
     * @return PrivateKey The private key used to decrypt
     *               the data.
     */
    public PrivateKey getPrikey() {
        return prikey;
    }

    /**
     * @Title: encrypt
     * @Description: This function is to encrypt the message
     *               m, m in [0,1,2,...,T],
     *               T=100 with public key.
     * @param m
     *            The message
     * @param pubkey
```

(continued)

```
     *            The public key of BGN PKE.
     * @return Element The ciphertext.
     * @throws Exception
     *             If the plaintext is not in [0,1,2,...,n],
     *             there is an exception.
     */
    public static Element encrypt(int m, PublicKey pubkey)
              throws Exception {
        if (m > T) {
            throw new Exception(
                    "BGN.encrypt(int m, PublicKey pubkey): "
                    + "plaintext m is not in [0,1,2,...,"
                            + T + "]");
        }
        Pairing pairing = pubkey.getPairing();
        Element g = pubkey.getG();
        Element h = pubkey.getH();
        BigInteger r = pairing.getZr().newRandomElement()
                        .toBigInteger();
        return g.pow(BigInteger.valueOf(m)).mul(h.pow(r))
                        .getImmutable();
    }

    /**
     *
     * @Title: decrypt
     * @Description: This function is to decrypt the ciphertext
     *               with the public key and the private key.
     * @param c
     *            The ciphertext.
     * @param pubkey
     *            The public key of BGN PKE.
     * @param prikey
     *            The private key of BGN PKE.
     * @return int The plaintext.
     * @throws Exception
     *             If the plaintext is not in [0,1,2,...,n],
     *             there is an exception.
     */
    public static int decrypt(Element c, PublicKey pubkey,
            PrivateKey prikey) throws Exception {
        BigInteger p = prikey.getP();
        Element g = pubkey.getG();
        Element cp = c.pow(p).getImmutable();
        Element gp = g.pow(p).getImmutable();
        for (int i = 0; i <= T; i++) {
            if (gp.pow(BigInteger.valueOf(i)).isEqual(cp)) {
                return i;
            }
        }
```

(continued)

```
        throw new Exception(
                "BGN.decrypt(Element c, PublicKey pubkey,
                 PrivateKey prikey): "
                + "plaintext m is not in [0,1,2,...,"
                        + T + "]");
    }

    public static int decrypt_mul2(Element c, PublicKey pubkey,
            PrivateKey prikey) throws Exception {
        BigInteger p = prikey.getP();
        Element g = pubkey.getG();
        Element cp = c.pow(p).getImmutable();
        Element egg = pubkey.getPairing().pairing(g, g).pow(p)
        .getImmutable();
        for (int i = 0; i <= T; i++) {
            if (egg.pow(BigInteger.valueOf(i)).isEqual(cp)) {
                return i;
            }
        }
        throw new Exception(
                "BGN.decrypt(Element c, PublicKey pubkey,
                 PrivateKey prikey): "
                + "plaintext m is not in [0,1,2,...,"
                        + T + "]");
    }

    /**
     * @Title: add
     * @Description: The function supports the homomorphic
     * addition with two ciphertext.
     * @param c1
     *            The ciphertext.
     * @param c2
     *            The ciphertext.
     * @param pubkey
     *            The public key of BGN PKE.
     * @return Element The return value is c1*c2.
     */
    public static Element add(Element c1, Element c2) {
        return c1.mul(c2).getImmutable();
    }

    /**
     * @Title: mul1
     * @Description: The function supports the homomorphic
     *               multiplication with one ciphertext
     *               and one plaintext.
     * @param c
     *            The ciphertext.
     * @param m
```

(continued)

```
     *            The plaintext.
     * @param pubkey
     *            The public key of BNG PKE.
     * @return Element The return value is c^m.
     */
    public static Element mul1(Element c1, int m2) {
        return c1.pow(BigInteger.valueOf(m2)).getImmutable();
    }

    /**
     * @Title: mul2
     * @Description: TODO
     * @param c1
     *            The ciphertext.
     * @param c2
     *            The ciphertext.
     * @param pubkey
     *            The public key of BNG PKE.
     * @return Element The return value is e(c1,c2).
     */
    public static Element mul2(Element c1, Element c2,
        PublicKey pubkey) {
        Pairing pairing = pubkey.getPairing();
        return pairing.pairing(c1, c2).getImmutable();
    }

    /**
     * @Title: selfBlind
     * @Description: The function supports the homomorphic
     *               self-blinding with one ciphertext
     *               and one random number.
     * @param c
     *            The ciphertext.
     * @param r
     *            A random number in Z_n.
     * @param pubkey
     *            The public key of BNG PKE.
     * @return Element The return value is c1*h^r2.
     */
    public static Element selfBlind(Element c1, BigInteger r2,
        PublicKey pubkey) {
        Element h = pubkey.getH();
        return c1.mul(h.pow(r2)).getImmutable();
    }

    public static void main(String[] args) {
        BGN bgn = new BGN();
        // Key Generation
        bgn.keyGeneration(512);
        BGN.PublicKey pubkey = bgn.getPubkey();
```

(continued)

```
BGN.PrivateKey prikey = bgn.getPrikey();

// Encryption and Decryption
int m = 5;
Element c = null;
int decrypted_m = 0;
try {
    c = BGN.encrypt(m, pubkey);
    decrypted_m = BGN.decrypt(c, pubkey, prikey);
} catch (Exception e) {
    e.printStackTrace();
}
if (decrypted_m == m) {
    System.out.println("Encryption and Decryption "
            + "test successfully.");
}

// Homomorphic Properties

// Addition
int m1 = 5;
int m2 = 6;
try {
    Element c1 = BGN.encrypt(m1, pubkey);
    Element c2 = BGN.encrypt(m2, pubkey);
    Element c1mulc2 = BGN.add(c1, c2);
    int decrypted_c1mulc2 = BGN.decrypt(c1mulc2,
    pubkey, prikey);
    if (decrypted_c1mulc2 == (m1 + m2)) {
        System.out.println("Homomorphic addition "
                + "tests successfully.");
    }
} catch (Exception e) {
    e.printStackTrace();
}

// multiplication-1
m1 = 5;
m2 = 6;
try {
    Element c1 = BGN.encrypt(m1, pubkey);
    Element c1expm2 = BGN.mul1(c1, m2);
    int decrypted_c1expm2 = BGN.decrypt(c1expm2,
    pubkey, prikey);
    if (decrypted_c1expm2 == (m1 * m2)) {
        System.out.println("Homomorphic
                multiplication-1 "
                + "tests successfully.");
    }
} catch (Exception e) {
```

(continued)

```
            e.printStackTrace();
        }

        // multiplication-2
        m1 = 5;
        m2 = 6;
        try {
            Element c1 = BGN.encrypt(m1, pubkey);
            Element c2 = BGN.encrypt(m2, pubkey);
            Element c1pairingc2 = pubkey.getPairing()
                    .pairing(c1, c2).getImmutable();
            int decrypted_c1pairingc2 =
            BGN.decrypt_mul2(c1pairingc2, pubkey, prikey);
            if (decrypted_c1pairingc2 == (m1 * m2)) {
                System.out.println("Homomorphic
                        multiplication-2 "
                        + "tests successfully.");
            }
        } catch (Exception e) {
            e.printStackTrace();
        }

        // self-Blinding
        m1 = 5;
        try {
            BigInteger r2 = pubkey.getPairing().getZr()
                    .newRandomElement().toBigInteger();
            Element c1 = BGN.encrypt(m1, pubkey);
            Element c1_selfblind = BGN.selfBlind(c1,
            r2, pubkey);
            int decrypted_c1_selfblind =
                  BGN.decrypt(c1_selfblind, pubkey, prikey);
            if (decrypted_c1_selfblind == m1) {
                System.out.println("Homomorphic self-blinding "
                        + "tests successfully.");
            }
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

References

1. S. Goldwasser and S. Micali, "Probabilistic encryption," *J. Comput. Syst. Sci.*, vol. 28, no. 2, pp. 270–299, 1984. [Online]. Available: http://dx.doi.org/10.1016/0022-0000(84)90070-9
2. P. Paillier, "Public-key cryptosystems based on composite degree residuosity classes," in *Advances in Cryptology - EUROCRYPT '99, International Conference on the Theory and Application of Cryptographic Techniques, Prague, Czech Republic, May 2–6, 1999, Proceeding*, 1999, pp. 223–238. [Online]. Available: http://dx.doi.org/10.1007/3-540-48910-X_16
3. T. Okamoto and S. Uchiyama, "A new public-key cryptosystem as secure as factoring," in *Advances in Cryptology - EUROCRYPT '98, International Conference on the Theory and Application of Cryptographic Techniques, Espoo, Finland, May 31 - June 4, 1998, Proceeding*, 1998, pp. 308–318. [Online]. Available: http://dx.doi.org/10.1007/BFb0054135
4. D. Naccache and J. Stern, "A new public key cryptosystem based on higher residues," in *CCS '98, Proceedings of the 5th ACM Conference on Computer and Communications Security, San Francisco, CA, USA, November 3–5, 1998.*, 1998, pp. 59–66. [Online]. Available: http://doi.acm.org/10.1145/288090.288106
5. I. Damgård and M. Jurik, "A generalisation, a simplification and some applications of paillier's probabilistic public-key system," in *Public Key Cryptography, 4th International Workshop on Practice and Theory in Public Key Cryptography, PKC 2001, Cheju Island, Korea, February 13–15, 2001, Proceedings*, 2001, pp. 119–136. [Online]. Available: http://dx.doi.org/10.1007/3-540-44586-2_9
6. D. Boneh, E. Goh, and K. Nissim, "Evaluating 2-dnf formulas on ciphertexts," in *Theory of Cryptography, Second Theory of Cryptography Conference, TCC 2005, Cambridge, MA, USA, February 10–12, 2005, Proceedings*, 2005, pp. 325–341. [Online]. Available: http://dx.doi.org/10.1007/978-3-540-30576-7_18
7. Y. Ishai and A. Paskin, "Evaluating branching programs on encrypted data," in *Theory of Cryptography, 4th Theory of Cryptography Conference, TCC 2007, Amsterdam, The Netherlands, February 21–24, 2007, Proceedings*, 2007, pp. 575–594. [Online]. Available: http://dx.doi.org/10.1007/978-3-540-70936-7_31
8. R. L. Rivest, A. Shamir, and L. M. Adleman, "A method for obtaining digital signatures and public-key cryptosystems," *Commun. ACM*, vol. 21, no. 2, pp. 120–126, 1978. [Online]. Available: http://doi.acm.org/10.1145/359340.359342
9. T. E. Gamal, "A public key cryptosystem and a signature scheme based on discrete logarithms," *IEEE Transactions on Information Theory*, vol. 31, no. 4, pp. 469–472, 1985. [Online]. Available: http://dx.doi.org/10.1109/TIT.1985.1057074
10. N. Koblitz, *A course in number theory and cryptography*. Springer Science & Business Media, 1994, vol. 114.
11. D. Boneh and M. K. Franklin, "Identity-based encryption from the weil pairing," in *Advances in Cryptology - CRYPTO 2001, 21st Annual International Cryptology Conference, Santa Barbara, California, USA, August 19-23, 2001, Proceedings*, 2001, pp. 213–229. [Online]. Available: http://dx.doi.org/10.1007/3-540-44647-8_13
12. D. Boneh, B. Lynn, and H. Shacham, "Short signatures from the weil pairing," in *Advances in Cryptology - ASIACRYPT 2001, 7th International Conference on the Theory and Application of Cryptology and Information Security, Gold Coast, Australia, December 9–13, 2001, Proceedings*, 2001, pp. 514–532. [Online]. Available: http://dx.doi.org/10.1007/3-540-45682-1_30
13. D. Boneh and B. Waters, "Conjunctive, subset, and range queries on encrypted data," in *Theory of Cryptography, 4th Theory of Cryptography Conference, TCC 2007, Amsterdam, The Netherlands, February 21–24, 2007, Proceedings*, 2007, pp. 535–554. [Online]. Available: http://dx.doi.org/10.1007/978-3-540-70936-7_29
14. A. J. Menezes, P. C. Van Oorschot, and S. A. Vanstone, *Handbook of applied cryptography*. CRC press, 1997.

15. C. Gentry, "Fully homomorphic encryption using ideal lattices," in *Proceedings of the 41st Annual ACM Symposium on Theory of Computing, STOC 2009, Bethesda, MD, USA, May 31 - June 2, 2009*, 2009, pp. 169–178. [Online]. Available: http://doi.acm.org/10.1145/1536414.1536440
16. M. van Dijk, C. Gentry, S. Halevi, and V. Vaikuntanathan, "Fully homomorphic encryption over the integers," in *Advances in Cryptology - EUROCRYPT 2010, 29th Annual International Conference on the Theory and Applications of Cryptographic Techniques, French Riviera, May 30 - June 3, 2010. Proceedings*, 2010, pp. 24–43. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-13190-5_2
17. Z. Brakerski, C. Gentry, and V. Vaikuntanathan, "(leveled) fully homomorphic encryption without bootstrapping," in *Innovations in Theoretical Computer Science 2012, Cambridge, MA, USA, January 8–10, 2012*, 2012, pp. 309–325. [Online]. Available: http://doi.acm.org/10.1145/2090236.2090262
18. Z. Brakerski and V. Vaikuntanathan, "Efficient fully homomorphic encryption from (standard) LWE," in *IEEE 52nd Annual Symposium on Foundations of Computer Science, FOCS 2011, Palm Springs, CA, USA, October 22–25, 2011*, 2011, pp. 97–106. [Online]. Available: http://dx.doi.org/10.1109/FOCS.2011.12
19. Z. Brakerski, "Fully homomorphic encryption without modulus switching from classical gapsvp," in *Advances in Cryptology - CRYPTO 2012 - 32nd Annual Cryptology Conference, Santa Barbara, CA, USA, August 19–23, 2012. Proceedings*, 2012, pp. 868–886. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-32009-5_50
20. A. López-Alt, E. Tromer, and V. Vaikuntanathan, "On-the-fly multiparty computation on the cloud via multikey fully homomorphic encryption," in *Proceedings of the 44th Symposium on Theory of Computing Conference, STOC 2012, New York, NY, USA, May 19–22, 2012*, 2012, pp. 1219–1234. [Online]. Available: http://doi.acm.org/10.1145/2213977.2214086
21. C. Gentry, A. Sahai, and B. Waters, "Homomorphic encryption from learning with errors: Conceptually-simpler, asymptotically-faster, attribute-based," in *Advances in Cryptology - CRYPTO 2013 - 33rd Annual Cryptology Conference, Santa Barbara, CA, USA, August 18–22, 2013. Proceedings, Part I*, 2013, pp. 75–92. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-40041-4_5
22. C. Li, R. Lu, H. Li, L. Chen, and J. Chen, "PDA: a privacy-preserving dual-functional aggregation scheme for smart grid communications," *Security and Communication Networks*, vol. 8, no. 15, pp. 2494–2506, 2015. [Online]. Available: http://dx.doi.org/10.1002/sec.1191

http://www.springer.com/978-3-319-32897-3

Privacy-Enhancing Aggregation Techniques for Smart Grid Communications

Lu, R.

2016, XVI, 177 p. 28 illus., Hardcover
ISBN: 978-3-319-32897-3