Computation Operations on an Encrypted Relational Database Utilizing Homomorphic Properties of Paillier's Algorithm

For CRUD Operations and Addition

Tafia Alifianty Dinita Putri / 18218038
Program Studi Sistem dan Teknologi Informasi
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jalan Ganesha 10 Bandung
E-mail: tafiaalifianty@gmail.com

***Abstract*— Currently data security is a problem, especially for data stored on servers that do not belong to them. Encryption is a way to keep that data safe. However, one of the shortcomings of encrypted data is that it is difficult to compute the stored data because the data must be decrypted first. The paillier algorithm is one of the public-key cryptographic algorithms that have homomorphic properties that support the operation of encrypted data without decrypting it first. This paper will discuss how to apply paillier algorithms to relational database encryption that allow operations to be performed on these databases.**

Keywords—paillier; database; operation; encryption; decryption

I. Introduction

The digital era allows humans to store files not in physical form anymore but in digital form. These files can be stored on storage media such as hard disks, CDs, DVDs, Flash drives, or in the cloud. Besides being easy to store data in digital form it is also cheap compared to storage in physical form.

Even though it has many advantages, storage in digital form will certainly have security risks such as data theft and data integrity risks. The solution to overcome this problem is to use encryption on the data. What if we have important data that is encrypted and stored on a server, then we want to compute the data.

For example, a company has monthly sales data stored on a rented server and the data is very confidential so it is encrypted. Then at one point, the company wanted to know the total sales of its company for the last 12 months for its business analysis. The company must decrypt the data on the server by sending the key first so that it can be decrypted and then it can calculate the total sales. Of course, this has a big risk because there could be certain parties who intercepted the delivery of the company's keys.

One solution to this problem is to perform operations on encrypted data so that no key delivery is made to the server. The company's total sales will be calculated in encrypted form. The encrypted total sales will then be sent from the server to the client and then decrypted on the client-side. This method is possible by using the paillier algorithm because the paillier algorithm has additive homomorphic properties.

This paper will discuss how to apply homomorphic properties to paillier algorithms to perform computational operations on encrypted databases. These operations are read, insert, delete, update, and addition operations.

II. Basic Theory

A. *Public key Cryptographic Algorithms*

Public-key cryptographic algorithms are cryptographic algorithms that use different keys to encrypt and decrypt messages. The public key is used to encrypt messages and the private key is used to decrypt messages. The public key cryptography scheme can be seen in Figure 1.

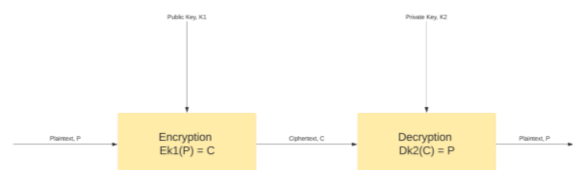


Figure 1. Schematic of public-key cryptographic algorithms

B. *Paillier Algorithm*

Paillier's Algorithm is a public key cryptographic algorithm that uses a probabilistic symmetric algorithm. This algorithm was invented by Pascal Paillier in 1999. Paillier's algorithm is based on the difficulty of calculating the nth class residue or the so-called composite residuosity problem. An integer z is said to be the nth residue of modulo n^2 if there is an integer y so $z = y^y \bmod n^2$ [1].

To generate the key pairs in the Pailier Algorithm, here are the steps that must be done:

1. Choose two prime numbers p and q that fulfill the *gdc* conditions $(pq, (p-1)(q-1)) = 1$

2. Calculate $n = pq$ and $\lambda = l\,cm\,(p - 1, q - 1)$
3. Select any integer g, with $g < n^2$
4. Calculate $\mu = (L\,(g^{\gamma} \bmod n^2))^{-1} \bmod n$

   With function L is $L(x) = .\frac{x-1}{n}$

The result of the above steps is a public key in the form of pair *g, n*, and a private key in the form of pair *λ, μ*.

The encryption process in the Pailier Algorithm is done with the way:

1. Divide plaintext into small blocks so the value of each plaintext block is less than n.
2. Choose r integer where r < n.
3. Encrypt every block with the formula

   $c = g^m r^{\eta} \bmod n^2,$

   with m is the plaintext block.

Then, the description process in the Paillier Algorithm is done with the way:

1. Describe every ciphertext block with formula

   $m = \frac{L\,(c^{\gamma} \bmod n^2)}{L\,(g^{\gamma} \bmod n^2)} \bmod n$, or

$m = L\,(c^{\gamma} \bmod n^2).\mu \bmod n$

2. Combine plaintext blocks into a complete message.

C. Paillier's Homomorphic Properties for Addition

Paillier Algorithm is a cryptographic algorithm that has homomorphic properties and also one of the homomorphic encryption algorithms [2].

Homomorphic encryption is a form of encryption that allows computation on the ciphertext without describing the ciphertext firstly. An operation performed on ciphertext that used homomorphic encryption will produce ciphertext which if described will make the same results with the similar operation on plaintext [4].

Mathematically, a homomorphic cryptosystem is a cryptosystem that used an encryption function that is homomorphic and allows the operation of the ciphertext to be carried out. There are two main types of operations namely addition and multiplication.

A cryptosystem is said to be additive if and only if:

$$\exists\triangle: \mathcal{E}(x1) \triangle \mathcal{E}(x2) = \mathcal{E}(x1 + x2)$$

With x1 and x2 is plaintext, Ɛ is encryption function, and △ is an operation that depends on properties of the encryption algorithm used. Then, a cryptosystem is said to be multiplicative if and only if:

$$\exists\triangle : \mathcal{E}(x1) \triangle \mathcal{E}(x2) = \mathcal{E}(x1.x2)$$

Paillier is an algorithm that had additive properties (addition). With multiplied two kinds of Paillier ciphertext so the description result will equivalent with added two values of the plaintext [2].

Additive properties of the Paillier Algorithm can be proven like this. Example: a and b are two kinds of plaintext, then c1 and c2 are ciphertexts from each of a and b with ElGamal encryption and g, y is the pair of the public key so:

$$c1 = g^a r_1^{\eta} \bmod n^2$$
$$c2 = g^b r_2^{\eta} \bmod n^2$$

With multiplied c1 and c2 will get the result:

$$c1.c2 = g^a r_1^{\eta}.g^b r_2^{\eta} \bmod n^2$$
$$= g^{a+b} (r_1 r_2)^{\eta} \bmod n^2$$

When the description is done, then:

$$d(g^{a+b} (r_1 r_2)^{\eta}) = a + b$$

So, d(c1.c2) = *a* + *b*.

D. Relational Database

The Relational Data Model is a database model that uses two-dimensional tables, which consist of rows and columns to describe a data file. This model shows how to physically manage/organize data in secondary memory, which will also have an impact on how we classify data and form all related data in the system we create.

In a relational database, data is stored in the form of a two-dimensional relationship or table, and between one table and another there is a relationship, so it can be concluded that the database is a collection of several tables that are interrelated. The collection of data organized as tables is stored in the form of electronic data on the computer hard disk and logically grouped according to the user schema [3].

E. JavaScript

JavaScript is a programming language used for creating and developing websites to make them more dynamic and interactive. JavaScript can increase functionality on web pages. Even JavaScript can also create applications, tools, or even games on the web.

Technically JavaScript or what is commonly called JS is an interpreter-type programming language, what is meant by an interpreter is a type of programming language that does not require a compiler to run it. JavaScript has features such as object-oriented, client-side, high-level programming, and loosely typed.

Javascript is a programming language that can also be used to execute complex algorithms and data logic. there are also some advantages of javascript compared to other programming languages. Among others:

- **General Purpose**

  Along with development, Javascript can now be used as a backend for application development purposes on any platform outside of the browser used. with NodeJs,

Javascript can now be used for desktop, console, mobile, IoT, gaming, application development.

- **Easy to Learn**
  Each programming language has a different level of difficulty. The level of difficulty can be seen from several factors. syntax or writing is one of the most important factors to learn. The program code required by JavaScript to execute a function or command is relatively shorter when compared to other programming languages. As an example :
  C++

```
#include <iostream>
int main()
{ std::cout << "Hello, world!\n";
return 0; }
```

Java

```
class AppHelloWorld {
public static void main(String[] args)
{
  System.out.println("Hello World!");
}
}
```

JavaScript

```
console.log('Hello World!')
```

The code above is the code to display the same screen “Hello World!”.

- **Community Support**
  This is one of the reasons JavaScript is loved by many developers. Support from the community is important in choosing a programming language. An example of the advantage of good community support is, if a bug is found, it will be easier to find a solution in one application or web because support from the community or group helps solve the problem.
- **Most Popular Languages**
  Being one of the most popular languages today, with the implementation of JavaScript in various applications and large websites such as Facebook, Linkedin, Trello, medium, and google.

F. MySQL

MySQL is a database management system (database management) using the basic command of SQL (Structured Query Language) which is quite well known. This multi-user and multi-flow MySQL database management system (DBMS) has been used by more than 6 million users worldwide. MySQL is an open-source DBMS with two license forms, namely Free Software (free software) and Shareware (proprietary software with limited use). So MySQL is a free database server with the GNU General Public License (GPL) so that it can be used for free without having to pay, whether used privately or commercially.

MySQL is included in the RDBMS (Relational Database Management System) type. Therefore, terms such as rows, columns, tables, are used in MySQL. For example, in a MySQL database, there are one or more tables. SQL itself is a language used in data retrieval in relational databases or structured databases. So MySQL is a database management system that uses the SQL language as the language of liaison between application software and the database server.

Despite being a fairly popular database, MySQL certainly has several advantages and disadvantages compared to other database servers. One of the drawbacks of MySQL is that its performance drops when some database management systems can perform well on large database management. As for the advantages of MySQL, such as:

- **Supports Integration With Other Programming Languages**
  Website or software is sometimes developed using a variety of programming languages, MySQL can help to develop software that is more effective and easier for integration between programming languages.
- **Doesn't Require Large RAM**
  MySQL can be installed on servers with small specifications. MySQL can still be used on servers with a capacity of 1 GB.
- **Multi-User Support**
  MySQL can be used by several users at the same time without making it crash or stop working. can be used when working on a team project so that the entire team can work at the same time without having to wait for other users to finish.
- **Open Source**
  MySQL is a free database management system. Even though it's free, it doesn't mean that this database has bad performance. Moreover, the free license used is the GPL under Oracle management, so the quality is good.
- **Flexible Table Structure**
  MySQL has a table structure that is easy to use and flexible. For example, when MySQL processes ALTER TABLE and so on. When compared to other databases such as Oracle and PostgreSQL, MySQL is classified as easier.
- **Various Data Types**
  Another advantage of MySQL is that it supports various kinds of data that you can use in MySQL. For example float, integer, date, char, text, timestamp, double, and so on.
- **Guaranteed Security**
  Open source doesn't mean MySQL provides bad security. On the contrary, MySQL has pretty slick security features. There are several layers of security implemented by MySQL, such as the hostname level, and the subnet mask. In addition, MySQL can also set user permissions with high-level password encryption.

III. Implementation Design

In implementing this encrypted database, only the values in the database records are encrypted, the schema and column names in the database are not encrypted. Encryption can be done on records in each column or only in certain columns. Keep in mind that the entire encryption and decryption process

is only done on the client-side, no encryption or decryption process is done on the server-side. This is done to ensure the security of data when sending and while on the server.

There are five operations on the database that will be implemented, namely read, insert, update, delete, and addition operations.

A. Read Operation

This operation is an operation to read records from the database. Because the records from the database are stored in encrypted form, to read the database, it is necessary to read the data using the select command in SQL first. After decrypting the encrypted data.

B. Insert Operation (Create)

Insert operation is used to insert a record into the database. Because the data stored in the database record is encrypted, the data must first be encrypted using a public key and then insertion is carried out with the insert command using SQL.

C. Update Operation

An update operation is an operation that changes the data in a record. This operation is performed almost the same as insert, namely by first encrypting the data and then updating it with ordinary SQL commands.

D. Delete Operation

The delete operation is an operation that deletes specific records in a table. This operation can be done without encrypting the data first if the conditions for doing the delete are not affected by the encrypted data. But if the conditions for doing a delete are influenced by encrypted data, the data must be encrypted first and then delete with ordinary SQL commands.

E. Addition Operation

This operation performs the sum of the values in the encrypted column. This operation is performed by utilizing the homomorphic nature of the paillier algorithm. The addition is done by the homomorphic nature of the Paillier algorithm, namely by multiplying the values of two encrypted data. This operation can be performed on the server-side because it does not involve any encryption or decryption processes.

IV. Implementation of Results Analysis

Simultaneously with the writing of this paper, a simple website program was implemented using the PHP and JavaScript programming languages to perform CRUD operations and addition on a simple database. The database has several tables, but in the experiments we conducted, the encryption was only done on one of my tables, namely the sales table. The sales table itself has several columns, namely the id column, date column, and total sales column.

Following will be displayed some of the pages contained in this simple program. This simple web-based program consists of several interface pages such as a sales list page, a sales data add a page and a sales data update page.

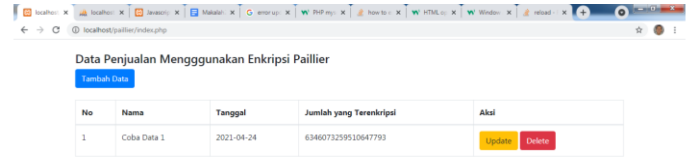


Figure 2. Start page

Figure 2 shows the start page of this simple program. There is a table that shows the data that has been stored in the database and there are also several buttons with their respective functions. The add data button is used to add data to the database. The update button functions to change data that has been stored in the database and the delete button functions to delete data that is already in the database. The sales table displays the data name, id, date of sale, as well as encrypted sales amount.

Figure 3. Add data page

Figure 3 shows the add data page. This page serves to fill in the data required for filling in the data in the sales table. Several fields must be filled first, such as name, the value of goods, and data requirements for encryption. The steps for filling in the data will be explained as follows:

1. Fill in the name
2. Fill in the Price Value
3. Select the Key Length to use for encryption
4. Press the "generate keypair" button
5. Press the "encrypt" button
6. Press the "Calculate [A + B]"

7. Enter the value that will be used for the multiplier as encryption needs
8. Press the "Calculate [(A + B) * C]" button
9. Press the button "Save to Database"

After performing the steps above, the data is stored in the database and will return to the first page. It can be seen that the data has been added to the table on the first page. In adding data, the total sales will be encrypted before entering the database and you will be able to see the details when you press and go to the update page.

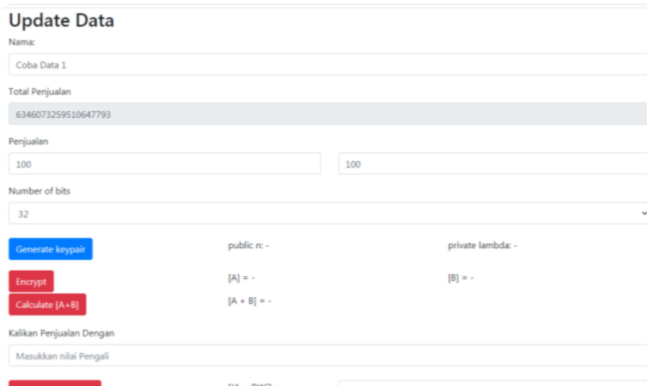


Figure 4. Data update page

Figure 4 shows the data update page. This page serves to change the data needed to replace the data in the sales table. On this page, you can change some data also re-encrypting it with a different key.

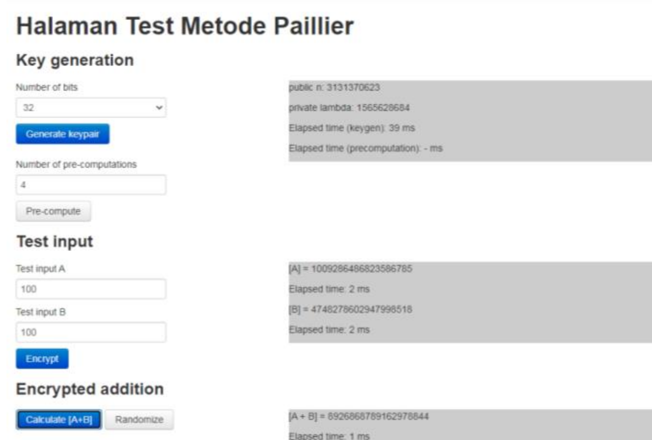


Figure 5. Paillier Method Test page

Figure 5 shows the Paillier Method Test page. This page serves to test the paillier method used to encrypt this program. This page also indicates the time required to perform encryption and decryption steps. Encryption is carried out by sequential steps starting from selecting the key bit length, entering plaintext, adding plaintext, to the last decryption process.

Experiments were carried out with different key lengths, namely 32 bits, 64 bits, 128 bits, 256 bits, 512 bits, and 1024 bits. The first experiment was conducted to see the speed of encryption and decryption of messages based on a certain key. The results of these experiments can be seen in Table 1. Based on the experimental results, the results show that the time for encryption and decryption of messages is proportional to the length of the key used. Then the length of the key is doubled causing the computation time to be quadrupled.

Table 1. The results of encryption and decryption experiments

No.	Key Size	Encryption Time (ms)	Decryption Time (ms)
1	32	1	1
2	64	1	1
3	128	2	2
4	256	2.5	2.5
5	512	4	4
6	1024	8	8

In the addition operation experiment, the ciphertext is added based on the homomorphic nature of the paillier algorithm (the ciphertext is multiplied). This operation is performed on 100 records in the sales column with a plaintext value range of 20 to 1000. The operation is repeated for different key lengths. The results of these experiments can be seen in Table 2.

Based on these experiments, it can be seen that the decryption of the homomorphic sum of the ciphertext produces the same result as the sum of unencrypted data. Meanwhile, the operating performance shows that the required operating time is proportional to the size of the key used. This is because the size of the ciphertext will also be doubled so that the operation time will also increase almost twice.

Table 2. The experimental results of the addition operation

No.	Key Size	Time (ms)	Amount (Results of the Program)	Actual Amount	Information
1	32	5	6229	6229	Succeed
2	64	5	6229	6229	Succeed
3	128	6	6229	6229	Succeed
4	256	10	6229	6229	Succeed
5	512	18	6229	6229	Succeed
6	1024	38	6229	6229	Succeed

In read, insert, and update operations, the operating speed is affected by the length of the key used. The results of the read,

insert, and update operation experiments respectively can be seen in Table 2, Table 4, and Table 5. From these tables, it can be seen that the longer the key used, the longer the time to operate will be. This is because the size of the ciphertext will also be longer or the size of the data will be bigger so that the time to query select, insert and update to the database will be even longer. Then the read, insert and update operations are also affected by the time to encrypt and decrypt data. In the delete operation, there is no encryption process or speed to delete the database. The results of the experiment can be seen in Table 6. In Table 6, it can be seen that the speed of the delete operation is not affected by the length of the key used. This is because, during the deletion process, the database only needs to know the location where the record to be deleted is stored and then deletes the pointer to the record.

Table 3. The experimental results of the read operation

No.	Key Size	Time (ms)	Time + decrypt (ms)
1	32	1	1.93
2	64	1	1.95
3	128	1	2.11
4	256	1	2.8
5	512	1	5.8
6	1024	1	17.8

Table 4. Experimental results of insert operations

No	Key Size	Time (ms)	Time + encrypt (ms)
1	32	30	30.13
2	64	33	33.15
3	128	34	34.31
4	256	55	56
5	512	53	57
6	1024	60	76

Table 5. The experimental results of the update operation

No	Key Size	Time (ms)	Time + encrypt (ms)
1	32	1	1.13
2	64	1	1.15
3	128	1	1.31
4	256	2	3
5	512	5	9
6	1024	9	25

In the delete operation, no encryption or decryption process is carried out so that the speed of the operation is only affected by the speed to delete to the database. The results of the experiment can be seen in Table 6. In Table 6, the speed of the delete operation is not affected by the length of the key used. This is because, during the deletion process, the database only needs to know the location where the record to be deleted is stored and then deletes the pointer to the record.

Table 6. The experimental results of the delete operation

No.	Key Size	Time (ms)
1	32	39
2	64	60
3	128	54
4	256	39
5	512	41
6	1024	31

The key generation code is written in the javascript code syntax below. In generating keys, the GenerateKey() function is used to generate keys according to the desired key length. Random function. SecureRandom() is used to generate any integer between 1 and $n2$ - 1.

```
paillier = {
publicKey: function(bits, n) {
            // bits
            this.bits = bits;
            // n
            this.n = n;
            // n2 (cached n^2)
            this.n2 = n.square();
            // np1 (cached n+1)
            this.np1 = n.add(BigInteger.ONE);
            this.rncache = new Array();
},
privateKey: function(lambda, pubkey) {
            // lambda
            this.lambda = lambda;
            this.pubkey = pubkey;
            // x (cached) for decryption
            this.x = pubkey.np1.modPow
       (this.lambda,pubkey.n2).
       subtract(BigInteger.ONE)
       .divide(pubkey.n).
       modInverse(pubkey.n);
},
generateKeys: function(modulusbits) {
  var p, q, n, keys = {}, rng = new
  SecureRandom();
  do {
    do {
     console.log(modulusbits>>1);
     console.log(rng);
     p = new BigInteger(modulusbits>>1,
```

```
    1,rng);
   } while (!p.isProbablePrime(10));
   do {
    console.log(modulusbits>>1);
    console.log(rng);
    q = new BigInteger(modulusbits>>1,
    1,rng);
  } while(!q.isProbablePrime(10));

n = p.multiply(q);
} while(!(n.testBit(modulusbits - 1)) || (p.compareTo(q) == 0));
keys.pub = new paillier.publicKey(modulusbits,n);
lambda = lcm(p.subtract(BigInteger.ONE)
,q.subtract(BigInteger.ONE));
keys.sec = new paillier.privateKey(lambda, keys.pub);
return keys;
}
}
```

```
paillier.publicKey.prototype = {
 encrypt: function(m) {
  return this.randomize (this.n.multiply(m)
  .add(BigInteger.ONE).mod(this.n2));
 },

 add: function(a,b) {
  return a.multiply(b).remainder(this.n2);
 },

 mult: function(a,b) {
  return a.modPow(b, this.n2);
 },

 randomize: function(a) {
  var rn;
  if (this.rncache.length > 0) {
  rn = this.rncache.pop();
  } else {
    rn = this.getRN();
  }
  return (a.multiply(rn)).mod(this.n2);
 },

 getRN: function() {
  var r, rng = new SecureRandom();
  do {
    r = new BigInteger(this.bits,rng);
  } while(r.compareTo(this.n) >= 0);
  return r.modPow(this.n, this.n2);
 },
\ precompute: function(n) {
  for (var i = 0; i < n; i++) {
    this.rncache.push(this.getRN());
  }
 }
}

paillier.privateKey.prototype = {
 decrypt: function(c) {
```

```
  return c.modPow(this.lambda,this.pubkey.n2)
  .subtract(BigInteger.ONE).
  divide(this.pubkey.n).
  multiply(this.x).mod(this.pubkey.n);
 }
}
```

In the encryption process, several steps must be done, the functions above show each step to complete the encryption process. These functions include encrypt, add, mult, randomize, getRN. The encrypt function is used to convert the initial plaintext into a chipper form with the generated key. The add function is used to add two ciphertexts into one ciphertext. The mult function is used for multiplication with new numbers. Another syntax will be attached in the attachment.

V. Security Analysis

Security in database operations that utilize paillier algorithms depends entirely on the security of the paillier algorithm. The paillier algorithm is based on the Composite Residuosity Class Problem or the problem of finding composite residues in a class. This problem is a problem with the complexity of the algorithm which is exponential depending on the length of the key used. Until now, there is no efficient algorithm to solve this problem.

Security in carrying out operations on an encrypted database is still guaranteed as long as the paillier algorithm is still secure. It's just that keep in mind that to ensure the paillier algorithm remains safe is to use a long key. The longer the key is used, the safer it will be. However, as a result, computational operations on an encrypted database will take longer.

VI. Conclusions and Suggestions

The use of encryption with a paillier algorithm on the database can increase the security of the database. The use of paillier encryption also makes it possible to perform several operations on the database such as read, insert, update, delete, and addition operations. The use of a long key will increase data security, but as a result, the time to perform operations on the data will also be longer due to the larger size of the ciphertext.

For further development, it is necessary to develop ways to perform operations on data so that computation time can be reduced.

VII. Attachment

BigInteger Functions page on generatekeys function:

```
var dbits;

// JavaScript engine analysis
var canary = 0xdeadbeefcafe;
var j_lm = ((canary&0xffffff)==0xefcafe);

// (public) Constructor
```

```
function BigInteger(a,b,c) {
  if(a != null)
    if("number" == typeof a) this.fromNumber(a,b,c);
    else if(b == null && "string" != typeof a)
this.fromString(a,256);
    else this.fromString(a,b);
}

// return new, unset BigInteger
function nbi() { return new BigInteger(null); }

// am: Compute w_j += (x*this_i), propagate carries,
// c is initial carry, returns final carry.
// c < 3*dvalue, x < 2*dvalue, this_i < dvalue

function am1(i,x,w,j,c,n) {
  while(--n >= 0) {
    var v = x*this[i++]+w[j]+c;
    c = Math.floor(v/0x4000000);
    w[j++] = v&0x3ffffff;
  }
  return c;
}
// am2 avoids a big mult-and-extract completely.
// Max digit bits should be <= 30 because we do bitwise ops
// on values up to 2*hdvalue^2-hdvalue-1 (< 2^31)
function am2(i,x,w,j,c,n) {
  var xl = x&0x7fff, xh = x>>15;
  while(--n >= 0) {
    var l = this[i]&0x7fff;
    var h = this[i++]>>15;
    var m = xh*l+h*xl;
    l = xl*l+((m&0x7fff)<<15)+w[j]+(c&0x3fffffff);
    c = (l>>>30)+(m>>>15)+xh*h+(c>>>30);
    w[j++] = l&0x3fffffff;
  }
  return c;
}


function am3(i,x,w,j,c,n) {
  var xl = x&0x3fff, xh = x>>14;
  while(--n >= 0) {
    var l = this[i]&0x3fff;
    var h = this[i++]>>14;
    var m = xh*l+h*xl;
    l = xl*l+((m&0x3fff)<<14)+w[j]+c;
    c = (l>>28)+(m>>14)+xh*h;
    w[j++] = l&0xfffffff;
  }
  return c;
}
if(j_lm && (navigator.appName == "Microsoft Internet
Explorer")) {
  BigInteger.prototype.am = am2;
  dbits = 30;
}
else if(j_lm && (navigator.appName != "Netscape")) {
  BigInteger.prototype.am = am1;
  dbits = 26;
}
else { // Mozilla/Netscape seems to prefer am3
  BigInteger.prototype.am = am3;
  dbits = 28;
}

BigInteger.prototype.DB = dbits;
BigInteger.prototype.DM = ((1<<dbits)-1);
BigInteger.prototype.DV = (1<<dbits);

var BI_FP = 52;
BigInteger.prototype.FV = Math.pow(2,BI_FP);
BigInteger.prototype.F1 = BI_FP-dbits;
BigInteger.prototype.F2 = 2*dbits-BI_FP;

// Digit conversions
var BI_RM = "0123456789abcdefghijklmnopqrstuvwxyz";
var BI_RC = new Array();
var rr,vv;
rr = "0".charCodeAt(0);
for(vv = 0; vv <= 9; ++vv) BI_RC[rr++] = vv;
rr = "a".charCodeAt(0);
for(vv = 10; vv < 36; ++vv) BI_RC[rr++] = vv;
rr = "A".charCodeAt(0);
for(vv = 10; vv < 36; ++vv) BI_RC[rr++] = vv;

function int2char(n) { return BI_RM.charAt(n); }
function intAt(s,i) {
  var c = BI_RC[s.charCodeAt(i)];
  return (c==null)?-1:c;
}


function bnpCopyTo(r) {
  for(var i = this.t-1; i >= 0; --i) r[i] = this[i];
  r.t = this.t;
  r.s = this.s;
}

// (protected) set from integer value x, -DV <= x < DV
function bnpFromInt(x) {
  this.t = 1;
  this.s = (x<0)?-1:0;
  if(x > 0) this[0] = x;
  else if(x < -1) this[0] = x+this.DV;
  else this.t = 0;
}

// return bigint initialized to value
function nbv(i) { var r = nbi(); r.fromInt(i); return r; }

// (protected) set from string and radix
function bnpFromString(s,b) {
  var k;
  if(b == 16) k = 4;
  else if(b == 8) k = 3;
  else if(b == 256) k = 8; // byte array
  else if(b == 2) k = 1;
  else if(b == 32) k = 5;
  else if(b == 4) k = 2;
```

```
  else { this.fromRadix(s,b); return; }
  this.t = 0;
  this.s = 0;
  var i = s.length, mi = false, sh = 0;
  while(--i >= 0) {
    var x = (k==8)?s[i]&0xff:intAt(s,i);
    if(x < 0) {
      if(s.charAt(i) == "-") mi = true;
      continue;
    }
    mi = false;
    if(sh == 0)
      this[this.t++] = x;
    else if(sh+k > this.DB) {
      this[this.t-1] |= (x&((1<<(this.DB-sh))-1))<<sh;
      this[this.t++] = (x>>(this.DB-sh));
    }
    else
      this[this.t-1] |= x<<sh;
    sh += k;
    if(sh >= this.DB) sh -= this.DB;
  }
  if(k == 8 && (s[0]&0x80) != 0) {
    this.s = -1;
    if(sh > 0) this[this.t-1] |= ((1<<(this.DB-sh))-1)<<sh;
  }
  this.clamp();
  if(mi) BigInteger.ZERO.subTo(this,this);
}

// (protected) clamp off excess high words
function bnpClamp() {
  var c = this.s&this.DM;
  while(this.t > 0 && this[this.t-1] == c) --this.t;
}

// (public) return string representation in given radix
function bnToString(b) {
  if(this.s < 0) return "-"+this.negate().toString(b);
  var k;
  if(b == 16) k = 4;
  else if(b == 8) k = 3;
  else if(b == 2) k = 1;
  else if(b == 32) k = 5;
  else if(b == 4) k = 2;
  else return this.toRadix(b);
  var km = (1<<k)-1, d, m = false, r = "", i = this.t;
  var p = this.DB-(i*this.DB)%k;
  if(i-- > 0) {
    if(p < this.DB && (d = this[i]>>p) > 0) { m = true; r =
int2char(d); }
    while(i >= 0) {
      if(p < k) {
        d = (this[i]&((1<<p)-1))<<(k-p);
        d |= this[--i]>>(p+=this.DB-k);
      }
      else {
        d = (this[i]>>(p-=k))&km;
        if(p <= 0) { p += this.DB; --i; }
      }
      if(d > 0) m = true;
      if(m) r += int2char(d);
    }
  }
  return m?r:"0";
}


function bnNegate() { var r = nbi();
BigInteger.ZERO.subTo(this,r); return r; }

function bnAbs() { return (this.s<0)?this.negate():this; }

// (public) return + if this > a, - if this < a, 0 if equal
function bnCompareTo(a) {
  var r = this.s-a.s;
  if(r != 0) return r;
  var i = this.t;
  r = i-a.t;
  if(r != 0) return (this.s<0)?-r:r;
  while(--i >= 0) if((r=this[i]-a[i]) != 0) return r;
  return 0;
}

// returns bit length of the integer x
function nbits(x) {
  var r = 1, t;
  if((t=x>>>16) != 0) { x = t; r += 16; }
  if((t=x>>8) != 0) { x = t; r += 8; }
  if((t=x>>4) != 0) { x = t; r += 4; }
  if((t=x>>2) != 0) { x = t; r += 2; }
  if((t=x>>1) != 0) { x = t; r += 1; }
  return r;
}

// (public) return the number of bits in "this"
function bnBitLength() {
  if(this.t <= 0) return 0;
  return this.DB*(this.t-1)+nbits(this[this.t-
1]^(this.s&this.DM));
}

// (protected) r = this << n*DB
function bnpDLShiftTo(n,r) {
  var i;
  for(i = this.t-1; i >= 0; --i) r[i+n] = this[i];
  for(i = n-1; i >= 0; --i) r[i] = 0;
  r.t = this.t+n;
  r.s = this.s;
}

// (protected) r = this >> n*DB
function bnpDRShiftTo(n,r) {
  for(var i = n; i < this.t; ++i) r[i-n] = this[i];
  r.t = Math.max(this.t-n,0);
  r.s = this.s;
}

// (protected) r = this << n
function bnpLShiftTo(n,r) {
```

```
  var bs = n%this.DB;
  var cbs = this.DB-bs;
  var bm = (1<<cbs)-1;
  var ds = Math.floor(n/this.DB), c = (this.s<<bs)&this.DM, i;
  for(i = this.t-1; i >= 0; --i) {
    r[i+ds+1] = (this[i]>>cbs)|c;
    c = (this[i]&bm)<<bs;
  }
  for(i = ds-1; i >= 0; --i) r[i] = 0;
  r[ds] = c;
  r.t = this.t+ds+1;
  r.s = this.s;
  r.clamp();
}

// (protected) r = this >> n
function bnpRShiftTo(n,r) {
  r.s = this.s;
  var ds = Math.floor(n/this.DB);
  if(ds >= this.t) { r.t = 0; return; }
  var bs = n%this.DB;
  var cbs = this.DB-bs;
  var bm = (1<<bs)-1;
  r[0] = this[ds]>>bs;
  for(var i = ds+1; i < this.t; ++i) {
    r[i-ds-1] |= (this[i]&bm)<<cbs;
    r[i-ds] = this[i]>>bs;
  }
  if(bs > 0) r[this.t-ds-1] |= (this.s&bm)<<cbs;
  r.t = this.t-ds;
  r.clamp();
}

// (protected) r = this - a
function bnpSubTo(a,r) {
  var i = 0, c = 0, m = Math.min(a.t,this.t);
  while(i < m) {
    c += this[i]-a[i];
    r[i++] = c&this.DM;
    c >>= this.DB;
  }
  if(a.t < this.t) {
    c -= a.s;
    while(i < this.t) {
      c += this[i];
      r[i++] = c&this.DM;
      c >>= this.DB;
    }
    c += this.s;
  }
  else {
    c += this.s;
    while(i < a.t) {
      c -= a[i];
      r[i++] = c&this.DM;
      c >>= this.DB;
    }
    c -= a.s;
  }
  r.s = (c<0)?-1:0;
  if(c < -1) r[i++] = this.DV+c;
  else if(c > 0) r[i++] = c;
  r.t = i;
  r.clamp();
}

// (protected) r = this * a, r != this,a (HAC 14.12)
// "this" should be the larger one if appropriate.
function bnpMultiplyTo(a,r) {
  var x = this.abs(), y = a.abs();
  var i = x.t;
  r.t = i+y.t;
  while(--i >= 0) r[i] = 0;
  for(i = 0; i < y.t; ++i) r[i+x.t] = x.am(0,y[i],r,i,0,x.t);
  r.s = 0;
  r.clamp();
  if(this.s != a.s) BigInteger.ZERO.subTo(r,r);
}

// (protected) r = this^2, r != this (HAC 14.16)
function bnpSquareTo(r) {
  var x = this.abs();
  var i = r.t = 2*x.t;
  while(--i >= 0) r[i] = 0;
  for(i = 0; i < x.t-1; ++i) {
    var c = x.am(i,x[i],r,2*i,0,1);
    if((r[i+x.t]+=x.am(i+1,2*x[i],r,2*i+1,c,x.t-i-1)) >= x.DV) {
      r[i+x.t] -= x.DV;
      r[i+x.t+1] = 1;
    }
  }
  if(r.t > 0) r[r.t-1] += x.am(i,x[i],r,2*i,0,1);
  r.s = 0;
  r.clamp();
}

// (protected) divide this by m, quotient and remainder to q, r
(HAC 14.20)
// r != q, this != m.  q or r may be null.
function bnpDivRemTo(m,q,r) {
  var pm = m.abs();
  if(pm.t <= 0) return;
  var pt = this.abs();
  if(pt.t < pm.t) {
    if(q != null) q.fromInt(0);
    if(r != null) this.copyTo(r);
    return;
  }
  if(r == null) r = nbi();
  var y = nbi(), ts = this.s, ms = m.s;
  var nsh = this.DB-nbits(pm[pm.t-1]);           // normalize
modulus
  if(nsh > 0) { pm.lShiftTo(nsh,y); pt.lShiftTo(nsh,r); }
  else { pm.copyTo(y); pt.copyTo(r); }
  var ys = y.t;
  var y0 = y[ys-1];
  if(y0 == 0) return;
  var yt = y0*(1<<this.F1)+((ys>1)?y[ys-2]>>this.F2:0);
  var d1 = this.FV/yt, d2 = (1<<this.F1)/yt, e = 1<<this.F2;
  var i = r.t, j = i-ys, t = (q==null)?nbi():q;
```

```
  y.dlShiftTo(j,t);
  if(r.compareTo(t) >= 0) {
    r[r.t++] = 1;
    r.subTo(t,r);
  }
  BigInteger.ONE.dlShiftTo(ys,t);
  t.subTo(y,y);       // "negative" y so we can replace sub with am
later
  while(y.t < ys) y[y.t++] = 0;
  while(--j >= 0) {
    // Estimate quotient digit
    var qd = (r[--i]==y0)?this.DM:Math.floor(r[i]*d1+(r[i-
1]+e)*d2);
    if((r[i]+=y.am(0,qd,r,j,0,ys)) < qd) {              // Try it out
      y.dlShiftTo(j,t);
      r.subTo(t,r);
      while(r[i] < --qd) r.subTo(t,r);
    }
  }
  if(q != null) {
    r.drShiftTo(ys,q);
    if(ts != ms) BigInteger.ZERO.subTo(q,q);
  }
  r.t = ys;
  r.clamp();
  if(nsh > 0) r.rShiftTo(nsh,r);           // Denormalize remainder
  if(ts < 0) BigInteger.ZERO.subTo(r,r);
}

// (public) this mod a
function bnMod(a) {
  var r = nbi();
  this.abs().divRemTo(a,null,r);
  if(this.s < 0 && r.compareTo(BigInteger.ZERO) > 0)
a.subTo(r,r);
  return r;
}

// Modular reduction using "classic" algorithm
function Classic(m) { this.m = m; }
function cConvert(x) {
  if(x.s < 0 || x.compareTo(this.m) >= 0) return x.mod(this.m);
  else return x;
}
function cRevert(x) { return x; }
function cReduce(x) { x.divRemTo(this.m,null,x); }
function cMulTo(x,y,r) { x.multiplyTo(y,r); this.reduce(r); }
function cSqrTo(x,r) { x.squareTo(r); this.reduce(r); }

Classic.prototype.convert = cConvert;
Classic.prototype.revert = cRevert;
Classic.prototype.reduce = cReduce;
Classic.prototype.mulTo = cMulTo;
Classic.prototype.sqrTo = cSqrTo;

// (protected) return "-1/this % 2^DB"; useful for Mont. reduction
// justification:
//         xy == 1 (mod m)
//         xy =  1+km
//   xy(2-xy) = (1+km)(1-km)
// x[y(2-xy)] = 1-k^2m^2
// x[y(2-xy)] == 1 (mod m^2)
// if y is 1/x mod m, then y(2-xy) is 1/x mod m^2

function bnpInvDigit() {
  if(this.t < 1) return 0;
  var x = this[0];
  if((x&1) == 0) return 0;
  var y = x&3;                // y == 1/x mod 2^2
  y = (y*(2-(x&0xf)*y))&0xf;            // y == 1/x mod 2^4
  y = (y*(2-(x&0xff)*y))&0xff;          // y == 1/x mod 2^8
  y = (y*(2-(((x&0xffff)*y)&0xffff)))&0xffff;   // y == 1/x mod
2^16
  // last step - calculate inverse mod DV directly;
  // assumes 16 < DB <= 32 and assumes ability to handle 48-bit
ints
  y = (y*(2-x*y%this.DV))%this.DV;                    // y ==
1/x mod 2^dbits
  // we really want the negative inverse, and -DV < y < DV
  return (y>0)?this.DV-y:-y;
}

// Montgomery reduction
function Montgomery(m) {
  this.m = m;
  this.mp = m.invDigit();
  this.mpl = this.mp&0x7fff;
  this.mph = this.mp>>15;
  this.um = (1<<(m.DB-15))-1;
  this.mt2 = 2*m.t;
}

// xR mod m
function montConvert(x) {
  var r = nbi();
  x.abs().dlShiftTo(this.m.t,r);
  r.divRemTo(this.m,null,r);
  if(x.s < 0 && r.compareTo(BigInteger.ZERO) > 0)
this.m.subTo(r,r);
  return r;
}

// x/R mod m
function montRevert(x) {
  var r = nbi();
  x.copyTo(r);
  this.reduce(r);
  return r;
}

// x = x/R mod m (HAC 14.32)
function montReduce(x) {
  while(x.t <= this.mt2)       // pad x so am has enough room later
    x[x.t++] = 0;
  for(var i = 0; i < this.m.t; ++i) {
    // faster way of calculating u0 = x[i]*mp mod DV
    var j = x[i]&0x7fff;
    var u0 =
(j*this.mpl+(((j*this.mph+(x[i]>>15)*this.mpl)&this.um)<<15))
&x.DM;
```

```
    // use am to combine the multiply-shift-add into one call
    j = i+this.m.t;
    x[j] += this.m.am(0,u0,x,i,0,this.m.t);
    // propagate carry
    while(x[j] >= x.DV) { x[j] -= x.DV; x[++j]++; }
  }
  x.clamp();
  x.drShiftTo(this.m.t,x);
  if(x.compareTo(this.m) >= 0) x.subTo(this.m,x);
}

// r = "x^2/R mod m"; x != r
function montSqrTo(x,r) { x.squareTo(r); this.reduce(r); }

// r = "xy/R mod m"; x,y != r
function montMulTo(x,y,r) { x.multiplyTo(y,r); this.reduce(r); }

Montgomery.prototype.convert = montConvert;
Montgomery.prototype.revert = montRevert;
Montgomery.prototype.reduce = montReduce;
Montgomery.prototype.mulTo = montMulTo;
Montgomery.prototype.sqrTo = montSqrTo;

// (protected) true iff this is even
function bnpIsEven() { return ((this.t>0)?(this[0]&1):this.s) ==
0; }

// (protected) this^e, e < 2^32, doing sqr and mul with "r" (HAC
14.79)
function bnpExp(e,z) {
  if(e > 0xffffffff || e < 1) return BigInteger.ONE;
  var r = nbi(), r2 = nbi(), g = z.convert(this), i = nbits(e)-1;
  g.copyTo(r);
  while(--i >= 0) {
    z.sqrTo(r,r2);
    if((e&(1<<i)) > 0) z.mulTo(r2,g,r);
    else { var t = r; r = r2; r2 = t; }
  }
  return z.revert(r);
}

// (public) this^e % m, 0 <= e < 2^32
function bnModPowInt(e,m) {
  var z;
  if(e < 256 || m.isEven()) z = new Classic(m); else z = new
Montgomery(m);
  return this.exp(e,z);
}
```

VIDEO LINK AT YOUTUBE

https://youtu.be/QmJbAuV7a-A

ACKNOWLEDGMENT

The program for conducting experiments can be seen at https://github.com/tafiaalifianty/Paillier

REFERENCES

[1] Paillier Pascal, "Public-Key Cryptosystems Based on Composite Degree Residuosity Classes", Gemplus Card International, 1999

[2] Morris Lieam, “Analysis of Partial and Fully Homomorphic Encryption”, Rochester Institute of Technology, 2013

[3] Silberschatz Abramam dkk, “Database System Concept 6th edition”, McGraw Hill, 2011.

[4] Wahana Komputer, “Panduan Belajar MySQL Database Server”, MediaKita, Indonesia, 2010.

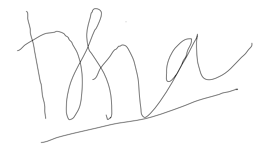
[5] Antony Pranata, ”Seri Pemrograman Internet : Panduan Pemrograman JavaScript (sampai dengan JavaScript 1.2)”, Penerbit Andi, Indonesia, 2011

[6] Potzelsberger, “KV Web Security: Application of Homomorphic Encryption”. 2013 http://www.fim.uni-linz.ac.at/lva/Web_Security/Abgaben/Poetzelsberger-Homomorphic.pdf . Diakses pada 7 Mei 2021

PERNYATAAN

Dengan ini saya menyatakan bahwa makalah yang saya tulis ini adalah tulisan saya sendiri, bukan saduran, atau terjemahan dari makalah orang lain, dan bukan plagiasi.

Bandung, 25 Mei 2021



Tafia Alifianty Dinita Putri
18218038