Private Information Retrieval

Simon Boré, Maël Bouquerot, Thomas Gergouil

May 15, 2021

Supervisor: Guilhem Castagnos

Contents

1 Introduction

In this paper, we intend to present one of the solutions to the following privacy problem, how could we allow a user to retrieve information from a database held by a server without disclosing what was claimed ? Even to the server itself.
The implementation will be based on the Private Information Retrieval (PIR) protocol. To do so, we will work with Paillier's cryptosystem, which offers security as it is based on a computationally hard assumption called the Decisionnal Composite Residuosity Assumption (DCRA) but it also has the properties that the protocol relies on.

2 Paillier's cryptosystem

2.1 Intractability hypothesis

Let $z \in (\mathbb{Z}/n^2\mathbb{Z})$, z is said to be a n-th residue modulo n^2 if $\exists y \in (\mathbb{Z}/n^2\mathbb{Z})$ such that z = $y^n \mod n^2$.
Deciding whether z is a n-th residue or not is not known to be achievable in polynomial time as long as the factorisation of n remains unknown. Meaning that if n = pq with p and q two large prime numbers of the same length, then the problem is assumed to be intractable. This is what we call "Decisionnal Composite Residuosity Assumption" (DCRA).

2.2 Security

For the Paillier's cryptosystem, the intractability hypothesis is essential in order to ensure the cryptosystem's security. Actually, the hypothesis being that the problem of composite residuosity is hard, if an attacker wanted to get the message from a cipher, he would have to find a solution to the composite residuosity problem, which is assumed not to be achievable in polynomial time when we consider that n is the product of two large random primes that remain private.

Thus, as the two primes used to create n are used for decryption, if an attacker were to intercept a cipher, the public keys g and n would be of no use to reverse the encryption.
Moreover, the two primes from which n is computed must remain secrete, otherwise the attacker just needs to apply the decryption scheme (see section 2.3.2 for details). In addition, finding these two primes from n is assumed to be impossible, this is the problem of prime factorization and it is known to be extremely hard to solve when primes are large enough, the current recommendation would be using at least 2048-bits long primes.

DCRA is said to be *random-self-reducible* which means that any case is as hard as the worst case is. Thus any case is an average case and the difficulty of solving an instance of the DCRA problem depends only on the length of n.

Finally, Paillier's cryptosystem is semantically secure as long as DCRA holds, meaning that if you encrypt one of the two messages m0 and m1 into a cipher c, none could determine which message has been encrypted. In other words, no information can be acquired from a cipher. This semantic security is related to the self-blinding property (see 2.4.1 for more information) it states that the same message can be encrypted by different ciphers, the decryption will not be impacted.

2.3 Cryptosystem

In order to implement the Paillier's cryptosystem we will use the first scheme presented in Paillier's paper [3], which is presented as *a probabilistic encryption scheme based on composite residuosity.*

2.3.1 Encryption

Let:

- n be the product of p and q two large primes of the same size (in bits),
- m be the message to encrypt,
- $g \in (\mathbb{Z}/n^2\mathbb{Z})$, its order is a non-zero multiple of n, we will use n+1,
- r be a random integer lower than n

Encryption is done as follow:

$$c = g^m.r^n \mod n^2$$

The trapdoorness of the function relies on DCRA which is, as said previously, intractable while the factorisation of n remains unknown.

2.3.2 Decryption

Let:

- $L(u) = \frac{u-1}{n}$
- $\lambda = lcm(p-1, q-1)$

The decryption is then:

$$m = \frac{L(c^\lambda \mod n^2)}{L(g^\lambda \mod n^2)} \mod n$$

2.3.3 Correctness

We work with a specific case: we have $g = n+1$, and $c = g^m.r^n \mod n^2$ the encryption of the message m.

We first need g^λ and c^λ, from Newton's Binomial Theorem:

$$g^\lambda = (n+1)^\lambda = \sum_{k=0}^{\lambda} \binom{\lambda}{k} n^k = \lambda n + 1 \mod n^2$$
$$\text{as for } k >= 2 \text{ then } n^k \text{ is divisible by } n^2$$

and similarly we have:

$$c^\lambda = (g^m.r^n)^\lambda = (n+1)^{m\lambda}.r^{n\lambda} = (n+1)^{m\lambda} = nm\lambda + 1 \mod n^2$$

For the function L defined above, we now have:

$$L(c^\lambda \mod n^2) = \frac{c^\lambda - 1}{n} = \frac{nm\lambda + 1 - 1}{n} = m\lambda \mod n$$

and

$$L(g^\lambda \mod n^2) = \frac{\lambda n + 1 - 1}{n} = \lambda \mod n$$

By definition, we have $gcd(n, \lambda) = 1$ thus λ is invertible modulo n, λ^{-1} exists.

With these, we have the following decryption of c:

$$\frac{L(c^\lambda \mod n^2)}{L(g^\lambda \mod n^2)} = \frac{m\lambda}{\lambda} = m \mod n$$

We can then conclude that the cryptosystem is correct.

2.4 Properties

2.4.1 Self-Blinding

Self-blinding is the property that any plaintext can produce different ciphertexts whithout affecting the decryption. This property is achieved thanks to the addition of randomness in the encryption process.

Indeed, with $r_0 \neq r_1$, $E_g(m, r_0) \neq E_g(m, r_1) \mod n^2$ but $D_g(E_g(m, r_0)) = D_g(E_g(m, r_1)) = m \mod n$

2.4.2 Additive Homomorphism

Additive homomorphic properties allows computations on ciphers. The decryption of the result will be the same as if corresponding computations were performed directly with the plaintext.

These two following properties will be useful for the PIR protocol:

1. $D_g(E_g(m_0, r_0)E_g(m_1, r_1) \mod n^2) = m_0 + m_1 \mod n$
2. $D_g(E_g(m_0, r_0)^c \mod n^2) = m_0 * c \mod n$

We show this with:

1. $E_g(m_0, r_0)E_g(m_1, r_1) = g^{m_0}{r_0}^n * g^{m_1}{r_1}^n = g^{m_0+m_1}(r_0 r_1)^n = E_g(m_0 + m_1, r_0 r_1) \mod n^2$
2. $E_g(m_0, r_0)^c = (g^{m_0}{r_0}^n)^c = g^{m_0 c}({r_0}^c)^n = E_g(m_0 c, {r_0}^c) \mod n^2$

And we should remember that the second argument in the encryption can be changed without affecting the decryption, only the ciphertext will be different. You can find the details of decryption in section 2.3.2.

In the rest of this paper, a particular case of this will be used several times so we show it here:

$$E_g(0, r_0)^{c_0} E_g(1, r_1)^{c_1} = E_g(0 * c_0, r_0^{c_0}) E_g(1 * c_1, r_1^{c_1}) = E_g(0, r_0^{c_0}) E_g(c_1, r_1^{c_1}) = E_g(c_1, r) \mod n^2$$

With $r = r_0^{c_0} r_1^{c_1}$. The decryption of this will return c_1.

3 Private Information Retrieval Protocol

Private Information Retrieval, or *PIR* for short, is a kind of cryptographic protocol that gives the ability to a user to retrieve some information from a database server, but the server must not know what that information is, this is called *oblivious transfer*.

The obvious answer would be to send the whole database, but the point here is to reduce as much as we can the amount of data to transfer as databases can be very large. We could as well argue that we don't want the user to retrieve more information than what he is supposed to.

We also used [2] to get more information about this protocol.

3.1 Paillier's Cryptosystem

As we need to reduce the amount of data that is transferred but not the security, the solution will be based on Paillier's Cryptosystem. It guarantees security under the DCRA assumption (see 2.2) and it also offers homomorphic properties that this solution relies on (see 2.4).

3.2 Protocol

Our solution is based on the description given in [1], as with it, we can achieve logarithmic amount of communications. Here, we talk about several propositions for the protocol according to the dimension of the database in order to reduce as much as possible this amount of communicated data. Details are in section 3.6.

3.3 Notation

For this section, we define the following:

- $l \in \mathbb{N}$, we have $[l] = \{1, 2, ..., l\}$
- x the database, we can see it as an array
- c, the dimension of the database
- $N = l^c$, the size of the database
- I(a,b), a function that returns 1 if a is equal to b and 0 otherwise

As well as an instance for Paillier's Cryptosystem (from section 2.3):

- $n = pq$, which is public, p and q are two large primes that are private
- $g \in \mathbb{Z}/n^2\mathbb{Z}$, we use $g = n + 1$
- E_g and D_g the corresponding encryption and decryption functions

Transfers and calculations are separated in several steps: Initialization (user side), Filtering/Invoking and Splitting-and-then-filtering (server side) and finally Reconstructing (user side).

3.4 Dimension 1

We now see the database as a normal array. We have:

- $N = l$
- $i \in [l]$
- $x(i) = x[i]$ the information at index i
- $x(i^*)$ the information the user wants to retrieve

3.4.1 Client Side Sending

Initialization:

For $t \in [l]$, the user takes one random in $\mathbb{Z}_n{}^*$ noted r_t.

Then computes,

- $\alpha_t = E_g(I(t, i^*), r_t) \mod n^2$

The user sends it all.

3.4.2 Server Side

Filtering:

The server computes $u = \prod_{t \in [l]} \alpha_t^{x(t)} \mod n^2$

Server sends u to the user.

3.4.3 Client Side Receiving

Reconstructing:

The user directly retrieves the wanted information by calculating: $D_g(u) \mod n$

3.4.4 Correctness

First, for $t \in [l]$, we have for α_t:

- if $t = i^*$ then $I(t, i^*) = 1$:
$$\alpha_t = E_g(1, r_t) \mod n^2$$
- else $I(t, i^*) = 0$:
$$\alpha_t = E_g(0, r_t) \mod n^2$$

So the construction of u gives:

$$u = (E_g(1, r_{i*}))^{x(i^*)}. \prod_{t' \in [l] \backslash i*} (E_g(0, r_{t'}))^{x(t')} \mod n^2$$

According to the homomorphic properties (you can find more details in 3.4.4), we now have:

$$u = E_g(x(i^*) + 0 + ... + 0, r) = E_g(x(i^*), r) \mod n^2$$

with r the random we get from the product of all randoms to their respective power.

From section 2.3.3 the decryption of u will return $x(i^*)$, the wanted information.

The user's security is also verified thanks to the semantic security, as when the different α_t are sent, the server nor any attacker can distinguish the α_t that has been computed with $t = i^*$ from the others.

3.5 Dimension 2

We now see the database as a double-entry array. We have:

- $N = l^2$
- $i, j \in [l]$
- $x(i, j) = x[(i-1)l + (j-1) + 1]$ the information at index (i,j)
- $x(i^*, j^*)$ the information the user wants to retrieve

3.5.1 Client Side Sending

Initialization:

For $t \in [l]$, the user takes two randoms in $\mathbb{Z}_n{}^*$ noted r_t and s_t.

Then computes,

- $\alpha_t = E_g(I(t, i^*), r_t) \mod n^2$
- $\beta_t = E_g(I(t, j^*), s_t) \mod n^2$

The user then sends it all to the server.

3.5.2 Server Side

Filtering:

For $i \in [l]$, the server computes $\sigma_i = \prod_{t \in [l]} \beta_t^{x(i,t)} \mod n^2$

Splitting-and-then-filtering:

First for each σ_i server computes u_i and $v_i \in Z_n$ such that $\sigma_i = u_i n + v_i \mod n^2$.

And then, it computes: $u = \prod_{t \in [l]} \alpha_t^{u_t} \mod n^2$ and $v = \prod_{t \in [l]} \alpha_t^{v_t} \mod n^2$.

Server sends u and v to the user.

3.5.3 Client Side Receiving

Reconstructing:

The user directly retrieves the wanted information by calculating: $D_g(D_g(u) * n + D_g(v)) \mod n$.

3.5.4 Correctness

First, for $t \in [l]$, we have for α_t and β_t:

- if I returns 1:

$$\alpha_t = E_g(1, r_t) \mod n^2$$
$$\beta_t = E_g(1, s_t) \mod n^2$$

- else I returns 0:

$$\alpha_t = E_g(0, r_t) \mod n^2$$
$$\beta_t = E_g(0, s_t) \mod n^2$$

Then for $i \in [l]$, the server computes σ_i:

$$\sigma_i = \prod_{t \in [l]} \beta_t^{x(i,t)} = \beta_{j*}^{x(i,j^*)}. \prod_{t' \in [l] \backslash j*} \beta_{t'}^{x(i,t')} = (E_g(1, s_{j*}))^{x(i,j^*)}. \prod_{t' \in [l] \backslash j*} (E_g(0, s_{t'}))^{x(i,t')} \mod n^2$$

And from homomorphic properties (you can find more details in 3.4.4), we have:

$$\sigma_i = E_g(x(i, j^*), s_i) \mod n^2$$

with s_i the random we get from the product of all randoms to their respective power.

Similarly, from $\sigma_i = u_i n + v_i \mod n^2$ the next step gives:

$$u = E_g(u_{i*}, r_u) \mod n^2$$
$$v = E_g(v_{i*}, r_v) \mod n^2$$

For the last step, we have:

$$D_g(u) = u_{i*} \mod n$$
$$D_g(v) = v_{i*} \mod n$$

Thus:

$$D_g(u)n + D_g(v) = u_{i*}n + v_{i*} = \sigma_{i*} = E_g(x(i*, j*), s_{i*}) \mod n^2$$

Finally, the user retrieves:

$$D_g(D_g(u)n + D_g(v)) = D_g(E_g(x(i*, j*), s_{i*})) \mod n$$

This decryption will return $x(i*, j*)$, the wanted information.

The user's security is also verified thanks to the semantic security, as when the different α_t, β_t are sent, the server nor any attacker can distinguish the α_t, β_t that have been computed with $t = i*$ and $t = j*$ from the others.

3.6 Further dimensions

3.6.1 Reduce the amount of data per transfer

As we will add dimensions to the database, the amount of information sent by the server will increase while the amount sent by the user will decreased. We will be able to reach communication balance between the server and the user.
Thus, to reduce as much as possible the data needed to be transferred we can build a dimension c database from a dimension (c-1) database using recursive calls.
To do that, both the server and the user treat the database x, which is a c dimensions database, as l databases of dimension (c-1).

3.6.2 Dimension c from c-1

We will present a sketch of the algorithm, it is recursive and stops calling lower dimensions databases at the execution of the algorithm for a 2 dimensions database.

We now see the database as a c-entry array. We have:

- $N = l^c$
- $i, j, \kappa ... \in [l]$, there are c of them
- $x(i, j, \kappa ...)$ the information at index $(i, j, \kappa ...)$
- $x(i*, j*, \kappa * ...)$ the information the user wants to retrieve

Client Side Sending

Initialization:

For $t \in [l]$, the user takes c randoms in $\mathbb{Z}_n{}^*$ noted $r_{1-t}, r_{2-t}, ..., r_{c-t}$

Then computes,

- $\alpha_t = E_g(I(t, i*), r_{1-t}) \mod n^2$
- $\beta_t = E_g(I(t, j*), r_{2-t}) \mod n^2$
- ...

There are c of them as the information needs c indexes to be retrieved.

The user then sends it all to the server.

Server Side

Invoking:

The server executes this algorithm for (c-1) dimensions on all l (c-1) dimensions databases without sending the result to the user, so the reconstructing part is not performed yet. As said previously, the dimension 2 algorithm does not call a dimension 1 algorithm, it is the last recursive call.

Splitting-and-then-filtering:

The server then splits the results and filters them using the encrypted indexes corresponding to the current dimension, raised to the splits.

If this is the original call (we are at the highest dimension), then the server sends the results to the user.

Client Side Receiving

Reconstructing:

The user retrieves the wanted information through recursive decryptions.

3.6.3 From 2 to 3 dimensions

Instead of working with a single database of dimension 3, we can work with l dimension 2 databases.

We see the database as a triple-entry array. We have:

- $N = l^3$
- $i, j, \kappa \in [l]$
- $x(i, j, \kappa) = x[(i-1)l^2 + (j-1)l + (\kappa - 1) + 1]$ the information at index (i, j, κ)
- $x(i^*, j^*, \kappa^*)$ the information the user wants to retrieve

Client Side Sending

Initialization:

For $t \in [l]$, the user takes three randoms in $\mathbb{Z}_n^*$ noted $r_{1-t}, r_{2-t}, r_{3-t}$

Then computes,

- $\alpha_t = E_g(I(t, i^*), r_{1-t}) \mod n^2$
- $\beta_t = E_g(I(t, j^*), r_{2-t}) \mod n^2$
- $\gamma_t = E_g(I(t, \kappa^*), r_{3-t}) \mod n^2$

The user then sends it all to the server.

Server Side

As the database is treated as l dimension 2 databases, we have $x(1) = x(i, j, 1)$, $x(2) = x(i, j, 2), ..., x(l) = x(i, j, l)$.

Invoking:

The algorithm for dimension 2 is used on every $x(d)_{d\in[l]}$ in parallel.

From each one of them, we get: $(u(d), v(d)) \mod n^2$.

Splitting-and-then-filtering:

The server calculates $uu_d, uv_d, vu_d, vv_d \in \mathbb{Z}_n$ with $u(d) = (uu_d)n + uv_d$ and $v(d) = (vu_d) + vv_d$.

Then it computes:

$$\begin{aligned}
uu &= \prod_{d\in[l]} (\gamma_d)^{uu_d} \mod n^2, \\
uv &= \prod_{d\in[l]} (\gamma_d)^{uv_d} \mod n^2, \\
vu &= \prod_{d\in[l]} (\gamma_d)^{vu_d} \mod n^2, \\
vv &= \prod_{d\in[l]} (\gamma_d)^{vv_d} \mod n^2.
\end{aligned}$$

And sends uu, uv, vu and vv to the user.

Client Side Receiving

Reconstructing:

Finally, the user retrieves the information by using recursive call of decryption:

$$x(i^*, j^*, \kappa^*) = D_g(D_g((D_g(uu)n + D_g(uv)))n + D_g((D_g(vu)n + D_g(vv)))) \mod n.$$

3.6.4 Correctness

To prove the correctness, we can reason by induction.

Correctness of dimension 3 from dimension 2

This proof uses the correctness of the dimension 2 that you can find in section 3.5.4. We then have:

$$uu = \prod_{d\in[l]} (\gamma_d)^{uu_d} = E_g(uu_{\kappa^*}, r) \mod n^2$$

Similarly with uv, vu, vv.

From this, we then have the two following results:

$$\begin{aligned}
D_g(uu)n + D_g(uv) = (uu_{\kappa^*})n + uv_{\kappa^*} = u_{\kappa^*} \mod n^2 \\
D_g(vu)n + D_g(vv) = (vu_{\kappa^*})n + vv_{\kappa^*} = v_{\kappa^*} \mod n^2
\end{aligned}$$

Thus, from final decryption:

$$D_g(D_g((D_g(uu)n + D_g(uv)))n + D_g((D_g(vu)n + D_g(vv)))) = D_g(D_g(u_{\kappa^*})n + D_g(v_{\kappa^*})) = x(i^*, j^*, \kappa^*) \mod n$$

Correctness of dimension c from dimension (c-1)

Now from a (c-1) dimensions database, we get $2^{(c-2)}$ server answers (that are supposed correct) as for each recursive call, you split the previous call answers into two. And it starts with 2 answers in dimension 2: u and v. In our example, in dimension 3, that was uu, uv, vu and vv. These answers have their (c-1) first indexes correct as the (c-1) dimensions call is supposed correct.

Then the algorithm for the c dimensions splits each answer into two and filters them using the c-th indexes to the power of the splits. This sets the last index correctly.

The user gets 2^c answers and through recursive decryption, retrieves the wanted information.

The user's security is also verified thanks to the semantic security, as when the different α_t, β_t... are sent, the server nor any attacker can distinguish the α_t, β_t... that have been computed with $t = i*$, $t = j*$... from the others.

3.6.5 Amount of transferred data

To measure the theoretical amount of data that will be transferred by the server and by the user, we use the security parameter $k = \lceil 2log(n) \rceil$ where n is the modulus used for Paillier's cryptosystem, this is the size of an element of $\mathbb{Z}_{n^2}$ in bits. Similarly to the rest of the paper, c is the dimension of the database and $l = N^{1/c}$

The server needs to return 2^{c-1} results as from each recursive call, each result will be split into two. At the end, the server returns all splits (that have been filtered) of the last call results. The dimension 2 is the last recursive call and it returns two results (u and v) thus it is indeed 2^{c-1}.

The user needs to send c indexes $(\alpha, \beta...)$ and each one of them is sent for $t \in [l]$ thus l times. So the number of elements sent is cl.

Thus, we have the following amount of data per communication in bits:

Dimension	Server side communication	User side communication
1	k	$l * k$
2	$2 * k$	$2l * k$
c	$2^{c-1} * k$	$cl * k$

We clearly see that when increasing the dimension of the database, we indeed reduce the amount of data sent by the user while the amount of data sent by the server increases. This is because $c * l = c * N^{1/c}$ decreases as we increase c the dimension but the server needs to return more information in order to retrieve the data.

4 Implementation

You can find the code for our implementation here: `https://github.com/tgergouil/TER_PIR`.

4.1 C and GMP

In order to implement an efficient protocol, the C is a go-to language thanks to its low memory and calculation time costs compared to other languages such as Python. Thus, the cryptosystem has been implemented in C language using the GMP library (`https://gmplib.org/`). This library provides functions for large integer arithmetic as well as functions for prime manipulation such as finding primes or deciding whether a given number is a prime or not.

4.2 Key generation

The intractability hypothesis is based on the assumption that the factorization of n is unknown and hard to compute. Therefore, the selection of the two prime factors p and q must be as random as possible, we will use 2048-bits long primes.

To pick a random prime number that is 2048-bits long, we use a set of two functions from the GMP library to pick a random number *gmp_randseed_ui* to add an initial seed value and *mpz_urandomb* to uniformly distributed generate a number. Then we use the *mpz_nextprime* function to get the next prime.

We could argue that this implementation would make some prime numbers more likely to be picked than others but the size of the numbers we are working on makes this attack impossible.

GMP's random number generator needs a random seed in order to be efficient, you will have to provide the program the seed as the argument when executing it. GMP recommends to use the system's random generator, for Linux it would be */dev/random*.

4.3 Efficiency

Efficiency tests were performed on our implementation for dimensions 1, 2 and 3 databases over databases of approximately 225 elements (n is 2048-bits long). We used the C function *clock()* and the constant *CLOCKS_PER_SECOND* from *time.h* library. The results are in seconds.

Dimension	N	User side Initialization	Server side	User side Reconstructing
1	220	65.32	0.15	0.59
2	225	6.27	0.09	0.96
3	216	5.35	0.18	4.16

For our function *get_primes()*, the execution takes: 2.304 seconds.

It was performed with a computer in 64 bits with a processor clock speed of 1.4GHz.

For the amount of data that was supposed to be transferred, we have $k = 4096$ bits or 512 bytes, we have in bytes:

Dimension	N	l	Server side communication	User side communication
1	220	220	512	$112,640$
2	225	15	1024	$15,360$
3	216	6	2048	$9,216$

References

[1] Yan-Cheng Chang. *Single Database Private Information Retrieval with Logarithmic Communication*. Cryptology ePrint Archive, Report 2004/036. `https://eprint.iacr.org/2004/036`. 2004.

[2] Rafail Ostrovsky and William E. Skeith III. *A Survey of Single Database PIR: Techniques and Applications*. Cryptology ePrint Archive, Report 2007/059. `https://eprint.iacr.org/2007/059`. 2007.

[3] Pascal Paillier. *Public-key cryptosystems based on composite degree residuosity classes*. IN ADVANCES IN CRYPTOLOGY — EUROCRYPT 1999 - Springer-Verlag - 223–238. `http://citeseer.ist.psu.edu/viewdoc/summary?doi=10.1.1.112.4035`. 1999.