Encryption Performance Improvements of the Paillier Cryptosystem

Christine Jost[1], Ha Lam[2], Alexander Maximov[3], and Ben Smeets[3]

[1] Ericsson Research, Stockholm, Sweden, christine.jost@ericsson.com
[2] work performed at Ericsson Research, San José, USA, hatlam@gmail.com
[3] Ericsson Research, Lund, Sweden, {alexander.maximov, ben.smeets}@ericsson.com

Abstract. Homomorphic encryption methods provide a way to outsource computations to the cloud while protecting the confidentiality of the data. In order to deal with the large and growing data sets that are being processed nowadays, good encryption performance is an important step for practicality of homomorphic encryption methods.
In this article, we study the encryption performance of the Paillier cryptosystem, a partially homomorphic cryptosystem that allows to perform sums on encrypted data without having to decrypt first. With a combination of both new and known methods, we increase the encryption performance by orders of magnitude compared to a naïve implementation. The new methods reduce the bottleneck of noise calculation by using pre-computed noise to generate new noise in a much faster way than by using standard methods.

1 Introduction

Homomorphic encryption schemes are currently attracting a lot of attention because they allow preserving confidentiality of sensitive data in cloud computing. However, fully homomorphic encryption schemes, which enable both multiplication and addition operations on encrypted data, are currently still inefficient in practical settings. For this reason, research efforts are also directed at different classes of homomorphic encryption, one of which is partially homomorphic encryption. These schemes allow performing one specific operation on encrypted data, usually either addition or multiplication. In this text, we consider two variants of a well-known partially homomorphic scheme, the Paillier scheme [8]. It allows performing sums on encrypted data, which is important in many use cases such as encrypted SQL databases [9, 5], machine learning on encrypted data [1], and electronic voting [8, 3].

In their most naïve implementation, both variants of the Paillier scheme have a rather bad encryption performance. There exist a couple of standard methods, some of which already mentioned in Paillier's original article [8], that considerably improve the encryption performance. In this article, we tune those existing methods and combine them with new methods. By doing this, we reach an encryption performance sufficient for high throughput use cases. On a commodity

machine, the throughput reached was between 30,000 and 190,000 encryptions per second, depending on the security level. See sections 4 and 5 for details on the results.

From a conceptual view point, encryption with the Paillier scheme consists of a fixed basis modular exponentiation with the message as exponent, and the generation of a noise factor used to mask the message. For fixed basis exponentiation, it is well-known that the performance can be increased by pre-computing powers of the fixed basis. By fine-tuning this and other known methods, we reduce the complexity of this step considerably. For the generation of the noise factor, we apply a new method that consists of using pre-computed noise to generate new noise factors. This reduces the bottleneck of noise computation to a few modular multiplications. Together, these methods achieve the considerable increase in encryption performance mentioned above. We elaborate on these methods in section 3. We will also describe methods that can be used for encrypting data sent in a data stream, especially a stream with irregular message frequency.

There have been other works built on the original Paillier scheme that improve its performance. Catalano et al. introduced an alternative decryption procedure, extending the scheme to allow an arbitrary exponent e instead of N; however, the drawback is that the system looses its additive homomorphic property [2]. Damgård et al. proposed a generalization of the scheme in which the expansion factor is reduced and implementations of both the generalized and original scheme are optimized without losing the homomorphic property [3]. Their system achieves the speed of 0.262 milliseconds/bit for the original Paillier scheme, equivalent to 3,816.79 bits/sec. This performance was reached by a clever choice of basis and using standard pre-computation techniques for fixed basis exponentiation. However, the encryption performance can be increased even further by using the techniques described in this paper. We reach a speed of 9,197,824 to 48,810,496 bits/sec depending on the security parameter.

Some of the methods described in this article become especially efficient if the message length is small compared to the key length, and hence also the ciphertext. This has the disadvantage of a larger ciphertext expansion. In order to reduce the ciphertext expansion, several smaller messages could be packed into a larger one, as described in [4]. However, using this method requires using a more complicated operation than simple multiplication in order to add the corresponding plaintexts.

In the following, we start by briefly describing the Paillier cryptosystem in Section 2, for completeness and in order to fix notation. We also comment on practical issues with key generation and parameter choice that are not completely addressed in the original article [8]. We continue by describing the methods we use for performance improvement in Section 3. Finally, we summarize the results in Section 4 and present our conclusion in Section 5.

2 The Paillier Cryptosystem, Key Generation and Parameter Choice

In [8], Paillier describes two partially-homomorphic cryptosystems, called Scheme 1 and Scheme 3. Scheme 1 is the basic version of the Paillier scheme, Scheme 3 is a variant with faster decryption. The security of the Paillier scheme is based on n-th residues in $\mathbb{Z}^*_{n^2}$ and the hardness of integer factorization. Here we only briefly recall the basic facts and comment on key generation and parameter choice. For details on the security of the scheme, we refer to the original article [8]. The setting for the Paillier scheme is the multiplicative group $\mathbb{Z}^*_{n^2}$, for $n = pq$ and two prime numbers p and q. Observe that $\mathbb{Z}^*_{n^2}$ has $|\mathbb{Z}^*_{n^2}| = \phi(n^2) = n\phi(n) = (p-1)(q-1)n$ elements. The Carmichael's function on n, $\lambda(n)$, is short-handed to λ.

2.1 Scheme 1

In its most basic form, the Paillier scheme is described in Table 1.

Table 1. Paillier's Scheme 1.

Parameters	
	prime numbers p, q
	$n = pq$
	$\lambda = \text{lcm}(p-1, q-1)$
	g, with $g \in \mathbb{Z}^*_{n^2}$ and the order of g is a multiple of n
Public key	
	n, g
Private key	
	p, q, λ
Encryption	
	plaintext $m < n$
	select a random $r < n$ such that $r \in \mathbb{Z}^*_n$
	ciphertext $c = g^m r^n \bmod n^2$
Decryption	
	ciphertext $c < n^2$
	plaintext $m = \frac{\text{L}(c^\lambda \bmod n^2)}{\text{L}(g^\lambda \bmod n^2)} \bmod n$

Following the notation in [8], $L(u) = \frac{u-1}{n}$, for $u = 1 \bmod n$. This function is only used on input values u that actually satisfy $u = 1 \bmod n$.

Key Generation and Secure Choice of Parameters. As the security of the Paillier cryptosystem is based on integer factoring, the same conditions should hold for n as for the modulus size in the RSA cryptosystem, which is recommended to be either 2048 or 3072 bits according to NIST recommendations [6].

When choosing the parameter g, it needs to be checked whether the order of the chosen g is a multiple of n. According to [8] equation (4), this can be effectively checked by testing whether

$$\gcd(\mathrm{L}(g^{\lambda} \bmod n^2), n) = 1.$$

Also, according to Paillier, g should be small for performance reasons, e.g., $g = 2$. In our implementation described in section 3, we choose to work with Scheme 3 instead that uses other values of g. This increases performance even further.

2.2 Scheme 3

This is a variant of the original Paillier scheme, with faster decryption. Instead of working in the whole group $\mathbb{Z}_{n^2}^*$, we work in the subgroup $\langle g \rangle$ generated by an element g of order αn. This allows decryption by performing an exponentiation with exponent α instead of λ, which speeds up decryption depending on the size of α. Scheme 3 is described in Table 2.

Table 2. Paillier's Scheme 3.

Parameters	
	prime numbers p, q
	$n = pq$
	$\lambda = \mathrm{lcm}(p-1, q-1)$
	α, a divisor of λ (see comment below)
	g, with $g \in \mathbb{Z}_{n^2}^*$ and the order of g is αn
Public key	
	n, g
Private key	
	p, q, α
Encryption	
	plaintext $m < n$
	select a random $r < \alpha$ (see comment below)
	ciphertext $c = g^m (g^n)^r \bmod n^2$
Decryption	
	ciphertext $c < n^2$
	plaintext $m = \frac{\mathrm{L}(c^{\alpha} \bmod n^2)}{\mathrm{L}(g^{\alpha} \bmod n^2)} \bmod n$

On α, Paillier only puts the restriction $1 \leq \alpha \leq \lambda$. However, as there is an element g of order αn, it follows from Carmichael's Theorem that α has to be a divisor of λ. Similarly, on r, Paillier actually puts the restriction $r < n$. However, it suffices to have $r < \alpha$ as the order of g^n is α.

Key Generation and Secure Choice of Parameters. The same conditions on p, q and n apply as for Scheme 1. Hence, in terms of parameters, we only need to consider the choice of α. Paillier comments in [8] that the parameter α

needs to be sufficiently large in order to avoid baby-step-giant-step attacks on α. We elaborate on this for the convenience of the reader.

The problem of computing α from the public keys g and n is the following: Solve $(g^n)^\alpha = 1 \bmod n^2$ for α. This problem is similar to the discrete logarithm problem, but not identical. To be more precise, instead of computing the discrete logarithm in a subgroup of known order, this task consists of finding the order of a given element. Hence, algorithms for solving the discrete logarithm problem that use the order of the subgroup cannot be used for this related problem. This includes the index calculus method and the Pohlig-Hellman algorithm. However, as already commented by Paillier, a baby-step-giant-step attack can be used for finding α. Applying this attack needs $\mathcal{O}(\sqrt{\alpha})$ multiplications.

Summing up, as the generic baby-step-giant-step method can be used to find α, but not the index calculus method, the same recommendations should hold for the size of α as for key size of ECC based systems. Also, it should be at least as hard to determine α as to factor n, hence the security level for α should be at least as high as the security level for n. According to NIST recommendations ([7] Table 2), 2048 bit security (or 3072 bit security, respectively) for factoring-based schemes is comparable to 224-255 bit security (256-383 bit security) for ECC based schemes. Hence, if n has size 2048 bits, α should have at least 224 bits. If n has size 3076 bits, α should have at least 256 bits.

The only new technical issue with parameter generation for Scheme 3 is to generate g with the correct order αn. Fortunately, this can be achieved simply by applying the DSA key generation twice, proper re-naming of variables and a few additional computations. The DSA key generation applied twice produces g_1 of order α_1 in $\mathbb{Z}_p^*$ and g_2 of order α_2 in $\mathbb{Z}_q^*$, respectively. As next step, we interpret g_1 and g_2 as elements in $\mathbb{Z}_{p^2}^*$ and $\mathbb{Z}_{q^2}^*$, respectively. It is straight-forward to check whether the order of g_1 in $\mathbb{Z}_{p^2}^*$ is $\alpha_1 p$ and the order of g_2 in $\mathbb{Z}_{q^2}^*$ is $\alpha_2 q$ indeed. (In our experiments with the OpenSSL DSA key generation, this was always the case.) One then finds g of order $\alpha_1\alpha_2 pq = \alpha n$ in $\mathbb{Z}_{p^2q^2}^* = \mathbb{Z}_{n^2}^*$ by using the Chinese Remainder Theorem. As can be easily checked, α is indeed a divisor of λ.

In the OpenSSL implementation of the DSA key generation, we can vary the size of the desired prime. Table 3 shows the bit sizes of different parameters for the Paillier key obtained by applying DSA key generation twice.

Table 3. Key sizes in bits for Scheme 3 of the Paillier cryptosystem, obtained by using OpenSSL DSA key generation twice.

p and q	α_1 and α_2	n	α
512	160	1024	320
1024	160	2048	320
1500	160	3072	320
2048	256	4096	512

2.3 Security of the Random n-th Powers r^n and g^{nr}

In [8], Paillier does not comment on how to generate the random n-th powers r^n (for Scheme 1) and g^{nr} (for Scheme 3). The only restriction on r^n, or g^{nr}, respectively, given by the schemes are the following: in Scheme 1, it suffices to choose r smaller than n, in order to produce all n-th powers in $\mathbb{Z}_{n^2}^*$. In Scheme 3, it suffices to choose r smaller than α, because the order of g is $n\alpha$.

However, it may be possible to choose r from a smaller set, which is interesting for performance reasons. In order to determine conditions on this smaller set, we consider two attacks on r and g^{nr} and measures to avoid them. We describe them as attacks on Scheme 3. However, they work for Scheme 1 in an analogous way.

The first attack on a specific ciphertext $c = g^m g^{nr}$ is by guessing r. To avoid this kind of attack, the set from which r is (pseudo-)randomly chosen may have less elements than α, as long as it is sufficiently large to make guessing r impossible in practice. Observe that, by guessing r, only a single ciphertext can be decrypted, so guessing r does not break the whole system. Assuming that the r's are chosen randomly and no other relations between them can be used, then the size of the random space hence only determines how hard it is to attack a single ciphertext. Its size may be chosen depending on the importance of single ciphertexts. For instance, for the performance tests in section 3.3, we choose r randomly from at least 2^{70} values. Hence we use at least 70-bit security for a single ciphertext.

The other kind of attack is to use relations between the random values used for different ciphertexts. Say we have the relation $r_3 = r_2 + r_1$. Then the attacker can compute

$$\frac{c_3}{c_1 \cdot c_2} = \frac{g^{m_3} g^{nr_3}}{g^{m_1} g^{nr_1} g^{m_2} g^{nr_2}} = g^{m_3 - m_1 - m_2}.$$

Obtaining $m_3 - m_1 - m_2$ is then a discrete logarithm problem. When the message length is short, the discrete logarithm problem should be solvable in reasonable time. In order to avoid this kind of attack, one should not introduce too much structure in the choice of r^n or g^{nr}.

The method for generating noise described in section 3.3 is designed to withstand the two attacks described above.

3 Implementation

In this section we discuss both the known and the new methods we applied to improve the encryption performance of the Paillier cryptosystem. We focus on Scheme 3, which has a better decryption performance than Scheme 1. However, the techniques can be used for Scheme 1 in an analogous way. Some of the techniques assume that the message length is short compared to the key size. As the key size is at least 2048 bits for modern applications and the messages are integer values, this seems to be a reasonable assumption for real world use cases. As mentioned in the introduction, an alternative way would be to use

the method described in [4], which reduces ciphertext expansion at the cost of introducing a more complicated method for additions on plain texts.

3.1 Reduced Moduli

kinda vague explanation, did not implement

To start with, if the private keys are known to the entity performing encryption, reduced moduli can be used. As $n^2 = p^2q^2$, one can split up numbers modulo n^2 in two parts modulo p^2 and q^2. The moduli p^2 and q^2 can be further reduced down to mod p and mod q. The computations are then be performed on the parts.

3.2 Computing $g^m \bmod n^2$

In order to compute the message part $g^m \bmod n^2$ of the ciphertext, standard methods for fixed basis modular exponentiation can be used. Pre-computations of powers of g modulo n^2 reduce the number of modular operations needed to compute $g^m \bmod n^2$ considerably. This holds especially if the messages m are rather short. There is of course always a trade-off between pre-computation storage and time on the one hand, and encryption speed on the other hand.

In our experiments, we assumed that the messages are 32-bit integers. In the pre-computation phase, we computed a large number of powers of g, namely $g^{(2^{16})^i j}$ for $i = 0, 1$ and $j = 0 \ldots 2^{16} - 1$. In order to compute g^m, we split m into two 16-bit numbers j_i, where $j_i = \lfloor m/(2^{16})^i \rfloor \bmod 2^{16}$ for $i = 0, 1$. Then g^m can be computed in only one modular multiplication, by $g^m \bmod n^2 = g^{2^{16} j_1} g^{j_0}$.

3.3 Computing $(g^n)^r \bmod n^2$

For computing the noise part of the ciphertext, it is of course possible to use similar pre-computation techniques as for the message part. However, in this case only *some* random power of g^n is needed to compute, not a specific power. We describe a new method that makes use of this fact and is much more efficient.

The idea is to pre-compute a large number of random powers of g^n and to use them to produce new random powers. This works because any product of random powers of g^n will produce a new random power. So instead of explicitly choosing r and computing $(g^n)^r$, we choose r implicitly while $(g^n)^r$ is computed as the product of previously computed random powers of g^n.

There is a trade-off again, between storage of pre-computed powers, number of multiplications and the "randomness" of the power of g^n. In order to evaluate the randomness of the power of g^n, we make use of the considerations in section 2.3 on the security of the choice of r.

In our implementation, according to the considerations in section 2.3, we want to make the probability of guessing r to be at most 2^{-70}. We consider pre-computing a table of 2^{16} random $(g^n)^r$'s and multiplying 5 of them together

during encryption. The $(g^n)^r$'s chosen might repeat themselves, so the number of possibilities is a 5-combination with repetitions, i.e.,

$$\left(\!\binom{2^{16}}{5}\!\right) = \binom{2^{16}+5-1}{5} \approx 2^{73}.$$

In Table 4, we also consider other values for the size of the table of pre-computed values. It details the pre-computation time (sec) and encryption speed (enc/sec) for different key lengths (bits).

Table 4. Pre-computation time (sec) and encryption speed (enc/sec), depending on the length of the key n, the number of pre-computed values, and the probability P of guessing r for a single ciphertext.

	2^{16} pre-computed values, noise = 5 $\Rightarrow P = 2^{-73}$		2^{16} pre-computed values, noise = 8 $\Rightarrow P = 2^{-113}$	
length of key n	Precomp	Enc	Precomp	Enc
1024	2.554725	190666	2.563800	116974
2048	5.661963	89483	5.848805	32480
3072	10.991574	35929	10.292777	42100
4092	25.156933	35236	25.510914	26553
	2^{16} pre-computed values, noise = 10 $\Rightarrow P = 2^{-138}$		2^{16} pre-computed values, noise = 16 $\Rightarrow P = 2^{-212}$	
length of key n	Precomp	Enc	Precomp	Enc
1024	2.676721	108682	2.674735	70687
2048	5.733593	56558	5.973831	34955
3072	10.416214	32978	10.189774	25287
4092	24.990774	23759	25.657435	17659
	2^{20} pre-computed values, noise = 4 $\Rightarrow P = 2^{-75}$		2^{20} pre-computed values, noise = 7 $\Rightarrow P = 2^{-128}$	
length of key n	Precomp	Enc	Precomp	Enc
1024	17.934370	205356	20.623479	121407
2048	42.923820	96326	43.360331	68066
3072	78.954561	57793	78.294688	42382
4092	188.125790	12140	188.942135	23693
	2^{20} pre-computed values, noise = 9 $\Rightarrow P = 2^{-162}$		2^{20} pre-computed values, noise = 15 $\Rightarrow P = 2^{-260}$	
length of key n	Precomp	Enc	Precomp	Enc
1024	20.578311	98633	20.312363	62165
2048	43.011205	55688	43.059426	36991
3072	78.187848	35953	78.725464	24269
4092	188.286545	17684	186.999658	17268

So far, we have described the method for the case that the random noise is of the form $(g^n)^r$. However, the same method can be employed for any random noise where pre-computed noise can be used to produce new noise. Especially, this method can be used for random noise of the form h^r, where h is fixed and

r is random. It also can be used for noise of the form r^n for random r and fixed n, such as for Scheme 1 of the Paillier cryptosystem. The reason is that even in this case new noise can be generated by multiplying pre-computed noise factors of the same form.

3.4 Parallelization

As the encryption of different ciphertexts is independent of each other, it is natural to use parallelization to encrypt large amounts of messages, or messages arriving with high throughput. In our implementation, we use parallelization using OpenMP, around the for loop that goes through the data set.

imagine these performance improvements over multiple CPUs working in parallel

3.5 Inhomogeneous Throughput

In the performance tests presented here, we have worked under the assumption that the messages either arrive as a large block of data, or arrive at constant speed. However, in many practical use cases, data will arrive at a fluctuating rate. In this case, it makes sense to split up the noise generation part from the message encryption part. The noise $(g^n)^r$ can be produced and stored either in a separate machine or in parallel in the same machine that encrypts the messages. When a message arrives in order to be encrypted, a new noise $(g^n)^r$ will be requested from the noise generation part. This noise will either taken from storage or be produced on the fly. During this time, the message encryption part can start computing the g^m part of the ciphertext. When both the g^m part and the $(g^n)^r$ noise part are present, they are multiplied in order to generate the ciphertext.

Although we have not tested this method, we believe that it should be very suited for the case that messages arrive in a stream of fluctuating throughput, such as produced by analytics flows. In one extreme form, all the noise parts of the expected ciphertexts are computed beforehand. If this is possible, it provides a very good encryption performance. This special case has been described before in [8] and [9].

4 Results

Summing up the results of the performance tests in the last section, the suggested improvements increase the encryption performance of the Paillier scheme dramatically. Naïve implementations of Paillier's Scheme 3 (and Scheme 1 as well) are rather slow. In our tests, Scheme 3 has speed of approximately 500 encryptions per second for 2048 bit length of n, see Table 5.

In contrast, the implementation with all the proposed improvements featured pre-computed tables of exponents, split of 32-bit number into smaller limbs, reduced moduli, and parallelization, etc., see Section 3 for more details. The speed increase that we gained is quite large. For 2048 key length, we achieved an encryption speed of 89,483 encryptions/second or 2,863,456 bits/second. Table 5 summarizes the speed of encryption (enc/sec) for different key lengths.

Table 5. Encryptions per second with different variations of Paillier, depending on the length of the key n. For our implementation, we use 73-bit security for guessing r.

length of key n	Naïve scheme 3	Our implementation
1024	1898	190666
2048	522	89483
3072	269	35929

The computer used for all the experiments has an Intel i7-4600U CPU at 2.10GHz with 4 cores, it runs Ubuntu 64-bit v14.04. The compiler version was g++ v4.8.2 with GMP v6.0.0 64-bit compiled against this setup.

Note that in our test cases, we were more interested in the number of encryptions per seconds rather than the number of bits encrypted per second, hence the improvements were optimized for messages of length 32 bits. For optimal encrypted bits/sec, messages of keylength size should be chosen.

5 Conclusion

Although naïve implementations of the Paillier cryptosystem have rather poor encryption performance, the performance can be improved considerable by using a combination of well-known and new methods. The well-known methods are pre-computations of fixed-basis powers, computing with reduced moduli, and parallelization. The new methods described in this article are a method to increase the speed of producing noise, and a method to deal with varying data rata of the message input stream. With these improvements, the Paillier cryptosystem becomes practical even for use cases that need high encryption throughput or have fluctuating data rate.

References

1. R. Bost, R. A. Popa, S. Tu, and S. Goldwasser, *Machine Learning Classification over Encrypted Data*. NDSS Symposium, 2015.
2. D. Catalano, R. Gennaro, N. Howgrave-Graham, and P. Q. Nguyen, *Paillier's cryptosystem revisited*. Proceedings of the 8th ACM conference on Computer and Communications Security, pp. 206-214, 2001.
3. I. Damgård, M. Jurik, and J. B. Nielsen, *A generalization of Paillier's public-key system with applications to electronic voting*. International Journal of Information Security 9, no. 6, pp. 371-385, 2010
4. T. Ge, and S. Zdonik, *Answering Aggregation Queries in a Secure System Model*. Proceedings of the 33rd International Conference on Very Large Data Bases, pp. 519-530, 2007.
5. P. Grofig, M. Härterich, I. Hang, F. Kerschbaum, M. Kohler, A. Schaad, A. Schröpfer, and W. Tighzert, *Experiences and observations on the industrial implementation of a system to search over outsourced encrypted data*. Sicherheit 2014: Sicherheit, Schutz und Zuverlässigkeit, Beiträge der 7. Jahrestagung des Fachbereichs Sicherheit der Gesellschaft für Informatik e.V. (GI), pp. 115-125, 2014.

6. NIST. SP 800-56B: *Recommendation for Pair-Wise Key Establishment Schemes Using Integer Factorization Cryptography*, 2009.
7. NIST. SP 800-57: *Recommendation for Key Management - Part 1: General (Revision 3)*, 2012.
8. Pascal Paillier, *Public-Key Cryptosystems Based on Composite Degree Residuosity Classes*. Advances in Cryptology - EUROCRYPT'99, vol. 1592 of Lecture Notes in Computer Science, pp. 223-238, 1999.
9. R. A. Popa, C. M. S. Redfield, N. Zeldovich, and H. Balakrishnan, *CryptDB: Protecting Confidentiality with Encrypted Query Processing*. Proceedings of the 23rd ACM Symposium on Operating Systems Principles, pp. 85-100, 2011.