Paillier Cryptosystem Optimisations for Homomorphic Computation

P. Ryšavá[1] and S. Ricci[1]

[1]Brno University of Technology, Technická 12, Brno, 616 00, Czech Republic

E-mail: xrysav29@vutbr.cz, ricci@vut.cz

Abstract—Homomorphic encryptions can ensure privacy in systems operating with sensitive data. It also allows outsourcing the data processing without the need to disclose the information within. To keep good performance over the growing mass of data, the execution of homomorphic schemes has to be efficient. In this article, we focus on the optimization of the Paillier scheme. This scheme allows the addition of a constant or another ciphertext without decryption of the encrypted values. Since the exponentiation used in the encryption process is time-consuming, we have implemented noise and message pre-computation to avoid time-demanding operations. These adjustments significantly fasten the encryption process, especially using the noise pre-computation.

Keywords— Homomorphic Encryption, Paillier Cryptosystem, Secret Sharing

1. INTRODUCTION

The Paillier cryptosystem [1] is a probabilistic asymmetric algorithm for public-key cryptography. This scheme has an additive homomorphic property, i.e., allows adding two ciphertexts without corrupting the result. The computation behaves just as if the corresponding plaintexts are added (without needing decryption). Because of this ability, the scheme became very popular, especially for e-voting [3], machine learning on encrypted data [4], and other algorithms that demand high confidentiality even during processing between mutually authenticated parties.

However, a drawback of the Paillier cryptosystem is its efficiency which is a common issue for any scheme based on homomorphic encryption. This article focuses on the Paillier scheme acceleration. We are especially interested in the encryption process. In fact, the encryption phase needs to be applied more times to do computation on encrypted data. Several optimisation proposition has already been made. For instance, Paillier proposed a variant, namely *Scheme 3*, of the original proposal, namely *Scheme 1*, which uses a bound α on exponentiation power size to speed up the computation. Other improving method can be the use of pre-computation of some values, particularly the exponentiation of either the message g^m or the noise r^n as suggested in the article [2]. See Section 2 for more details.

In this work, we implemented *Scheme 1* and *Scheme 3*, and we optimised them with message and noise exponentiation pre-computation. Moreover, we considered the pre-computation of (1) the modular inverse μ used during the decryption process, and (2) n^2 and $(g^n)^r$ used in *Scheme 3*. In particular, we develop an application in the *Visual Studio Code* editor that uses the C/C++ Extension Pack. The back-end layer runs in C programming language. At last, we compared the optimizations through experimental results.

2. IMPLEMENTATION

Our implementation uses the C programming language extended with the *OpenSSL* and *cJSON* package. The *OpenSSL* library implements the `BIGNUMBERs` library, which allows computation with larger numbers than the size of the `unsigned long long` variable type. The library also implements many cryptographic functions, such as the Digital Signature Algorithm (DSA) parameters generation, used in *Scheme 3* during the generation phase. *Scheme 1* and *Scheme 3* algorithms are sketched in Figure 1. Moreover, the *cJSON* library implements procedures simplifying the operations over the JavaScript Object Notation (JSON) files. In the implemented application, JSON files store the pre-computed values and corresponding keys since the JSON structure is simple and allows fast search.

The goal of our implementation is to speed up the cryptosystem procedure. We use the pre-computation of (1) message exponentiation (g^m), (2) noise exponentiation where r^n is computed as $(g^n)^r$, and (3) values n^2 and μ. Pre-computed message and noise values are stored in a JSON file whereas n^2 and μ values are stored directly in the keychain structure since the latter values remain the

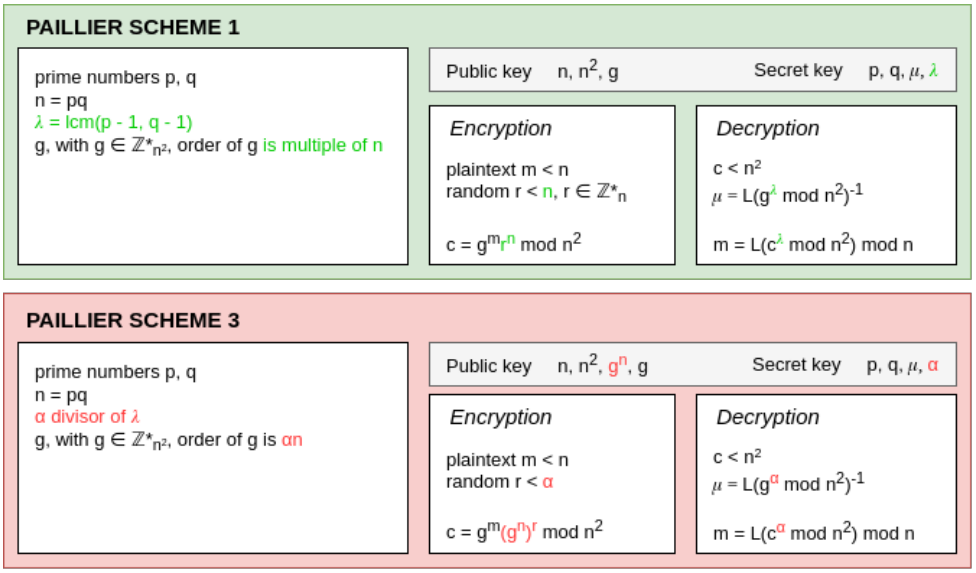

Figure 1: Paillier *Scheme 1* and *Scheme 3* description with highlighted differences in green and red colours.

same for the given keychain. For the purposes of *Scheme 3*, the g^n parameter is also precomputed and stored directly in the keychain structure.

2.1. Application

In the application development, the *Visual Studio Code* editor (v1.65.0) with the C/C++ Extension Pack (v1.1.0) was used. The directory tree of the implemented application is depicted on Figure 2. When started, the application opens with loading the saved keys from JSON files with prefix *saved_keys* located in the `keys` sub-directory. If any of the files are missing, a new corresponding keychain is generated and stored for future purposes. After the key-load phase, the algorithm checks if the file containing the pre-computed values exists. If not, the algorithm runs the pre-computation operation with parameters set in the `parameters` header file. Once the pre-computation is finished, the algorithm waits for the user to continue into a console menu.

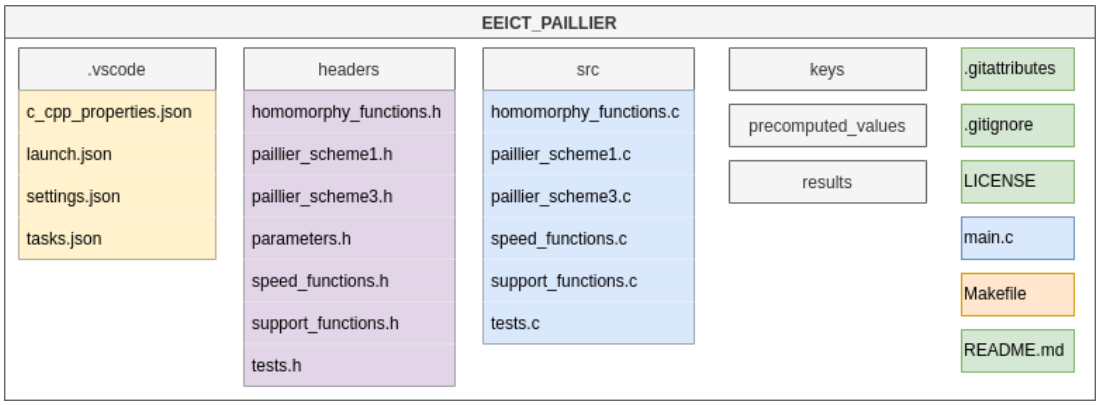

Figure 2: Directory tree of the application.

The application menu contains two areas named *RUN* and *TESTS*. Inside these sub-menus are listed all available functions. The ran operation is decided by inserting a number in the console. Each implemented scheme can be run either in the plain version, with noise, message pre-computation, or with both message and noise pre-computations. All of the above-stated ”modes” can also be launched together. Gained results from the computations are stored in Comma Separated Value (CSV) format in the results sub-directory.

It is important to notice that the amount of the precomputed values can be defined in the `parameters` header file by setting the parameter `RANGE`. This parameter states that the pre-computation will be applied on values from 1 to the agreed number. The parameter `RANGE` is set by default to 100. With this setting,

the size of files with the pre-computed values is 380.9 kB. When the parameter is equal to 1000, the file-size is about 259.1 kB. The size of the files with the saved keys is 8.4 kB big.

The implemented application is publicly available on GitHub repository[1].

Scheme 1			
	Encryption	Decryption	Total
Plain	2.172961	1.915783	4.088744
Random	0.036213	2.084682	2.120895
Message	2.137328	1.913080	4.050408
Both	0.008832	2.060453	2.069285

Scheme 3			
	Encryption	Decryption	Total
Plain	2.530260	2.387416	4.917676
Random	0.096640	2.514155	2.610795
Message	2.414621	2.359763	4.774384
Both	0.025454	2.521757	2.547211

Table I: Average time of encryption, decryption and total time in *ms* for 2048-bit length of n. ”Plain” states for in plain mode, ”random” for pre-computed noise values, ”message” for pre-computed message values, and ”both” for both pre-computations.

3. EXPERIMENTAL RESULTS

In this section, we compare the efficiency of plain schemes and their optimizations. For the experiments, we used an Intel(R) Core(TM) i7-4510U CPU at 2.00 GHz with 4 cores and Ubuntu 64-bit v20.04.4 LTS operating system. The compiler version is GCC v9.3.0. The experimental results were gained for a DSA modulus n of 2048-bit long. Encryption results of one message were averaged over 100 iterations and 10 different messages values were considered. In our case, the messages are integers since we are planning to use its homomorphic property as future work.

Table I depicts *Scheme 1* and *Scheme 2* performances averaged over 10 messages, respectively. Based on the gained results, the decryption speed is for all variants comparable. This is due to fact that the decryption operations are limitingly using the pre-computation optimisation. Regarding the encryption, *Scheme 1* is slightly quicker than *Scheme 3* despite expectations. This probably occurs because of the smaller size of the exponentiation base r in *Scheme 1* with respect to g^n in *Scheme 3*. The results were obtained with parameter `RANGE` sets to 100 where the pre-computed message file has size 64.9 kB for both schemes and the pre-computed noise files have size 64.3 and 125.9 kB for *Scheme 1* and *Scheme 3*, respectively.

Figure 3 depicts *Scheme 1* and *Scheme 3* computational cost of one message encryption averaged over 100 times iteration of the considered protocol. From the analyses, we can deduce that the most computationally intensive operation is the noise computation. This is the main reason of the drastic speedup when the noise pre-computation is applied to the process.

Optimisation technique	Scheme 1 saved time [*ms*]	Scheme 3 saved time [*ms*]
Noise pre-computation	3.286700	3.660420
μ pre-computation	2.544364	3.100727

Optimisation technique	Schemes 1 and 3 saved time [*ms*]
Message pre-computation	0.039340
n^2 pre-computation	0.039340
g^n pre-computation	13.568727

Table II: Average time in *ms* saved during each optimisation of the considered Paillier schemes.

Table II contains the saved times in average over 10 protocol executions of the considered pre-computations technique. Note that μ and n^2 acceleration allows saving time during the whole process since they are applied multiple times in the protocols.

[1] https://github.com/Norted/EEICT_Paillier

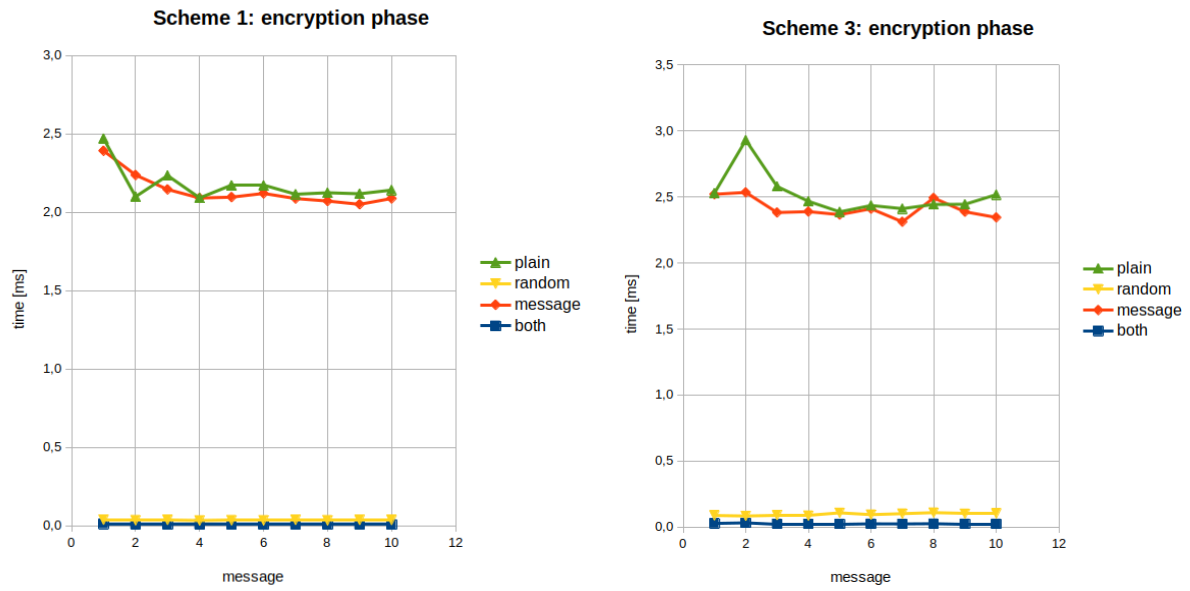

Figure 3: Encryption overtime of both *Schemes 1 and 3* with their optimisations. ”Plain” states for in plain mode, ”random” for pre-computed noise values, ”message” for pre-computed message values, and ”both” for both pre-computations.

4. CONCLUSION

Paillier is a homomorphic asymmetric cryptographic scheme that allows addition operations over encrypted values without corrupting the result. However, the scheme presents efficiency drawbacks and its performance needs to be perfected.

In this article, we develop an application in the *Visual Studio Code* editor with C programming language. Our application implements and compares several optimizations of the Paillier scheme. We mainly focused on the encryption process since it is normally applied more times in homomorphic computations, whereas the decryption phase is applied only once. In particular, we considered pre-computation of (1) noise exponentiation and (2) message exponentiation. Moreover, we implemented n^2 and μ pre-computations in all protocols.

The experimental results shows that the most time-saving operation is to use the noise pre-computation. Such operation speeds the encryption process significantly. As future work, we plan to apply the Paillier scheme with noise exponentiation pre-computation in our authentication key agreement protocol where it will be used during the sharing-phase in order to maintain the secrecy of the shares.

REFERENCES

[1] Paillier P. (1999). *Public-Key Cryptosystems Based on Composite Degree Residuosity Classes*. In: Stern J. (eds) Advances in Cryptology — EUROCRYPT ’99. EUROCRYPT 1999. Lecture Notes in Computer Science, vol 1592. Springer, Berlin, Heidelberg. DOI: https://doi.org/10.1007/3-540-48910-X_16. [Online; accessed 08-March-2022]

[2] Jost C., Lam H., Maximov A., and Smeets B. (2015). *Encryption Performance Improvements of the Paillier Cryptosystem*. Cryptology ePrint Archive, Report 2015/864. https://ia.cr/2015/864. [Online; accessed 08-March-2022]

[3] Anggriane S. M., Nasution S. M.and Azmi F. (2016). *Advanced e-voting system using Paillier homomorphic encryption algorithm*. International Conference on Informatics and Computing (ICIC), pp. 338-342. DOI: 10.1109/IAC.2016.7905741. [Online; accessed 08-March-2022]

[4] Bost R. and, Popa R. A., Tu S., and Goldwasser S. (2014). *Machine Learning Classification over Encrypted Data*. Cryptology ePrint Archive, Report 2014/331. DOI: https://dx.doi.org/10.14722/ndss.2015.23241.https://ia.cr/2014/331. [Online; accessed 08-March-2022]