Group signatures in practice

V. Gayoso Martínez[1], L. Hernández Encinas[1], and Seok-Zun Song[2]

[1] Institute of Physical and Information Technologies (ITEFI)
Spanish National Research Council (CSIC), Madrid, Spain
{victor.gayoso,luis}@iec.csic.es
[2] Department of Mathematics, Jeju National University, Jeju, Korea
szsong@jejunu.ac.kr

Abstract. Group signature schemes allow a user to sign a message in an anonymous way on behalf of a group. In general, these schemes need the collaboration of a Key Generation Center or a Trusted Third Party, which can disclose the identity of the actual signer if necessary (for example, in order to settle a dispute). This paper presents the results obtained after implementing a group signature scheme using the Integer Factorization Problem and the Subgroup Discrete Logarithm Problem, which has allowed us to check the feasibility of the scheme when using big numbers.

Keywords: Cryptography, Digital signature, Group signature, Java

1 Introduction

The concept of group signatures was first proposed by Chaum and van Heyst in 1991 [1]. Group signatures allow a certain group to sign a message such that only one member of the group computes the signature on behalf of the whole group.

The main properties that must satisfy a group signature scheme are the following:

 (i) Only one member signs the message on behalf of the group.
 (ii) The receiver of the message can verify that its associated signature was generated by a member of the signer group, but he or she cannot determine which member of the group was the actual signer.
(iii) In case it is necessary, it must be possible to determine which group member was the actual signer of the message.

There exist several proposals involving group signatures and their applicability to different scenarios (e.g, [2–7]). Some of these proposals need a Key Generation Center (KGC), denoted by $\mathcal{C}$, or a Trusted Third Party (TTP), denoted by $\mathcal{T}$, at least for the initialization process. Other schemes, however, allow any user to create the group they choose to belong to. The actions performed by $\mathcal{C}$ and $\mathcal{T}$ are similar and, for this reason, the roles of both entities are usually considered equivalent.

As a general rule, the cryptographic primitives used by those proposals base their security on computationally-intractable mathematical problems such as the Integer Factorization Problem (IFP) and the Subgroup Discrete Logarithm Problem (SDLP). As it is well known, the IFP can be described as follows [8]: Given a positive integer n, find its prime factorization; that is, write $n = p_1^{e_1} p_2^{e_2} \cdots p_k^{e_k}$ where the p_i are pairwise distinct primes and each $e_i \geq 1$. Besides, the SDLP is defined as follows [8]: Let p be a prime and q a prime divisor of $p - 1$. Let us consider g a generator of the unique subgroup G of $\mathbb{Z}_p^*$ of order q, and y an element in G. The problem is that of computing the integer x, $0 \leq x \leq q - 1$, such that $y = g^x \pmod{p}$.

For example, the schemes described in [9–14] base their security in the random oracle model, the strong RSA problem and the Decisional (bilinear) Diffie-Hellman problem. Other schemes, such as [15–17], are identity-based group signature protocols. They mainly use bilinear maps, such as bilinear pairings, which tend to be heavy in terms of computational load. In our case, we have employed a construction based on the SDLP which is different to those proposed in the aforementioned references.

This work presents the results obtained when implementing a modified version of a group signature scheme previously designed by one of the authors [18]. The goal of the modification, as it will become clear when describing the scheme, is to facilitate the implementation in devices with limited resources [19, 20]. In our scheme, $G = \{U_1, U_2, \ldots, U_t\}$ is the group of users allowed to perform signatures. Those users share a public key, and at the same time they have different private keys. When a signature is needed, the element playing the role of the KGC, $\mathcal{C}$, randomly chooses a member of the group so that member can perform the signature on behalf of G. After that, the verifier of the signature can check if the signature was performed by one of the members of G by using the public key that all members share. Without further information, the verifier will not be able to identify who was the original signer, unless the verifier is the PKC.

The rest of this paper is organized as follows: In section 2, a detailed description of the group signature scheme is included. Section 3 describes the Java application developed in order to test the feasibility of the scheme. In Section 4, we offer to the readers the experimental results obtained with that application. Finally, our conclusions are presented in section 5.

2   Description of the scheme

Let $G = \{U_1, U_2, \ldots, U_t\}$ be the group of users allowed to perform signatures, and $\mathcal{C}$ the element acting as the Key Generation Center. The following subsections describe all the details of the group signature scheme.

2.1   Setup phase

In this phase, $\mathcal{C}$ generates the system parameters, its own private key, the public key shared by the group, and the private keys of all members of G [21]. The steps that $\mathcal{C}$ must complete are the following ones:

1. $\mathcal{C}$ chooses two large primes p and q, such that $p = u_1 r p_1 + 1$ and $q = u_2 r q_1 + 1$, where r, p_1, q_1 are prime numbers and $u_1, u_2 \in \mathbb{Z}$ with $\gcd(u_1, u_2) = 2$; that is, $u_1 = 2v_1$, $u_2 = 2v_2$, where v_1 and v_2 are prime numbers. In the original version [18], v_1 and v_2 could be composite numbers; we have introduced this modification so that the number of factors of $\lambda(n)$ (see next step) does not depend on v_1 and v_2, which improves the iteration through the divisors of $\lambda(n)$ in the third step.
   In order to guarantee the security of the scheme, the bit length of r is selected so that the SDLP of order r in $\mathbb{Z}_n^*$ is computationally infeasible.

2. $\mathcal{C}$ computes the values $n = pq$, $\phi(n) = (p-1)(q-1) = u_1 u_2 r^2 p_1 q_1$, and $\lambda(n) = \mathrm{lcm}(p-1, q-1) = 2v_1 v_2 r p_1 q_1$, where $\phi(n)$ is the Euler function and $\lambda(n)$ is the Carmichael function.

3. $\mathcal{C}$ selects an element $\alpha \in \mathbb{Z}_n^*$ with multiplicative order r modulo n, such that $\gcd(\alpha, \phi(n)) = 1$. The element α can be efficiently computed as $\mathcal{C}$ knows the factorization of n and consequently it knows $\phi(n)$ and $\lambda(n)$.
   In practice, it is enough to find a random value $g \in \mathbb{Z}_n^*$ such that $g^{\lambda(n)} \equiv 1 \pmod{n}$ and check that none of the 62 non-trivial divisors of $\lambda(n)$ are the actual order of g [21]. By non-trivial divisor we mean a divisor of $\lambda(n)$ different from 1 or $\lambda(n)$. The number of non-trivial divisors of $\lambda(n)$ is derived from the fact that $\lambda(n) = 2v_1 v_2 r p_1 q_1$ and all the factors are prime numbers. Once the value g is found, the generator is obtained through the following computation [21]:
$$\alpha = g^{\lambda(n)/r} \pmod{n}.$$

4. $\mathcal{C}$ generates a secret random number $s \in \mathbb{Z}_r^*$ and determines
$$\beta = \alpha^s \pmod{n}.$$

5. $\mathcal{C}$ publishes the values n, r, α, and β, while the elements p, q, and s are kept secret.

6. $\mathcal{C}$ sets its private key by generating four random numbers $a_0, b_0, c_0, d_0 \in \mathbb{Z}_r^*$.

7. $\mathcal{C}$ determines the shared public key for G by computing
$$P = \alpha^{a_0} \beta^{b_0} \pmod{n} = \alpha^{a_0 + s b_0} \pmod{n},$$
$$Q = \alpha^{c_0} \beta^{d_0} \pmod{n} = \alpha^{c_0 + s d_0} \pmod{n}.$$

8. $\mathcal{C}$ computes the integers $h, k \in \mathbb{Z}_r$ such that $h = a_0 + s b_0 \pmod{r}$ and $k = c_0 + s d_0 \pmod{r}$.

9. $\mathcal{C}$ determines the private key for each signer $U_i \in G$, $1 \le i \le t$, where each private key is the tuple (a_i, b_i, c_i, d_i) and $a_i, b_i, c_i, d_i \in \mathbb{Z}_r$.
   In order to do that, $\mathcal{C}$ first generates t pairs of random numbers, $b_i, d_i \in \mathbb{Z}_r$. Then, it obtains the remaining elements by using the following equations:
$$a_i = h - s b_i \pmod{r}, \quad c_i = k - s d_i \pmod{r}.$$

   Once $\mathcal{C}$ has obtained the private keys of all the users, it distributes them to the signers via some secure channel.

2.2   Parameter and key verification

Each member of the signer group, U_i, $1 \leq i \leq t$, may check the parameters of the system by verifying that $\alpha \neq 1 \pmod{n}$ and $\alpha^r = 1 \pmod{n}$.

Moreover, each signer, U_i, $1 \leq i \leq t$, may verify that their private key is related to the shared public key, by checking:

$$P = \alpha^{a_i} \beta^{b_i} \pmod{n}, \quad Q = \alpha^{c_i} \beta^{d_i} \pmod{n}. \tag{1}$$

2.3   Group signature generation

Let M be the message to be signed by a member of G. By using, for example, a public hash function of the SHA-2 family [22], either the signing user or $\mathcal{C}$ compute $\mathfrak{h}(M) = m$, where m represents the hash output.

In order to calculate the signature, U_i must obtain the values f_i and g_i that compose the signature in this way:

$$f_i = a_i + c_i m \pmod{r}, \quad g_i = b_i + d_i m \pmod{r}. \tag{2}$$

After that, the group signature for the message M, which is $(f, g) = (f_i, g_i)$, can be published.

2.4   Group signature verification

Any verifier knowing the message, M, the hash function, $\mathfrak{h}$, the public key of the group G, (P, Q), and the group signature, (f, g), can check if the signature is valid through the following computation:

$$PQ^m = \alpha^f \beta^g \pmod{n}. \tag{3}$$

Equation (3) can be justified from expressions (1)–(2):

$$\alpha^f \beta^g \pmod{n} = \alpha^{a_i + mc_i} \beta^{b_i + mc_i} = \alpha^{a_i} \beta^{b_i} \left(\alpha^{c_i} \beta^{d_i} \right)^m = PQ^m.$$

2.5   Disclosure of the signing user

In case it is necessary, $\mathcal{C}$ is able to verify the signature and determine the actual signer, as $\mathcal{C}$ knows all the private keys of the users belonging to the group G.

If the original message is M, its associated hash code is $m = \mathfrak{h}(M)$, and the corresponding group signature is the pair (f, g), $\mathcal{C}$ needs to iterate the following loop:

$$\left. \begin{array}{l} \bar{f}_i = a_i + c_i m \pmod{r} \\ \bar{g}_i = b_i + d_i m \pmod{r} \end{array} \right\} \quad 1 \leq i \leq t, \tag{4}$$

stopping whenever it finds an index j, such that $(\bar{f}_j, \bar{g}_j) = (f, g)$. Using this procedure, $\mathcal{C}$ determines that the actual signer was U_j.

3 Java implementation of the scheme

The group signature scheme presented in this contribution has been implemented as a Java application using Java SE 8. The application is composed of three panels which are described in detail in the next subsections. In each panel, the user has the option of converting the data from decimal (or text, in the case of the message to be signed) to hexadecimal and vice versa.

In all the cases where a random number is needed, the application uses the standard Java classes `BigInteger` [23] and `Random` [24], so the requested number is obtained through the following code:

```
Random random = new Random();
BigInteger number = new BigInteger(numBits,random);
```

In the previous code, the element `numBits` indicates that the desired number must be uniformly distributed over the range 0 to $2^{\texttt{numBits}} - 1$. Regarding the `Random` class, it uses a 48-bit seed which is modified using a linear congruential formula according to the method described in Section 3.2.1 of [25].

Whenever a random prime number is needed, the following code is used after obtaining a random number:

```
BigInteger prime = number.nextProbablePrime();
```

By calling the method `nextProbablePrime()` over the element `number`, the application obtains the first integer greater than `number` that is probably prime, where the probability that the number returned is composite does not exceed 2^{-100} [23].

3.1 *Parameters* panel

This panel includes the general parameters, the KGC's private key and the group's public key, as it can be seen in Figure 1. More specifically, it includes text boxes for the private elements p, q, s, a_0, b_0, c_0, and d_0 and the public elements n, r, α, β, P, and Q.

There are four buttons available in this panel:

- *Generate*: It computes all the parameters according to the steps 1-8 of the procedure described in Section 2.1.
- *Save*: It allows the user to save either the public data or all the data included in this panel. The information is stored in a file using an XML structure.
- *Load*: It allows the user to overwrite the data existing in the text boxes with the information stored in the XML file selected by the user.
- *Clear*: It deletes the content of all the text boxes pertaining to this panel.

Fig. 1. View of the three panels of the application

3.2   *Users* panel

This panel includes the private keys of the four users managed by this application. It is important to point out that the number of users implemented in this version of the application is not a limitation of the scheme, but a figure selected in order to simplify the usage of the application. For each user from $i = 1$ to 4, a set consisting of the associated values a_i, b_i, c_i, and d_i is displayed, as it can be seen in Figure 1.

The four buttons available in this panel implement the following functionality:

- *Generate*: It generates all the private elements associated to the private keys of the users according to the step 9 of the procedure described in Section 2.1.
- *Save*: It allows the user to save the private elements of the four users in a file using an XML structure.
- *Load*: It allows to overwrite the data existing in the text boxes with the information stored in the XML file selected by the user.
- *Clear*: It deletes the content of all the text boxes displayed in this panel.

3.3   *Operations* panel

This panel includes the operational functionality that can be accessed through the following buttons, as displayed in Figure 1:

- *Generate*: It generates the signature of the text message provided manually by the user according to Equation (2) included in Section 2.3. In order to obtain the elements f and g associated to the signature, it is necessary to select in the panel the hash function and the signing user.
- *Verify*: It allows to verify if the signature provided by the application user corresponds to the text message entered in its text box. For this functionality, the application implements Equation (3) from Section 2.4. In order to perform this verification, the application only uses the public data available in the *Parameters* panel.
- *Disclose*: By selecting this button, the application first verifies if the signature provided is correct, and then it completes the calculations that allow the server to identify which of the four users signed the message. In order to do this, the application implements Equation (4) included in 2.5.
- *Clear*: It deletes the content of all the text boxes displayed in this panel.

4   Experimental results

The tests whose results are presented in this section were completed using a PC with Windows 7 Professional OS and an Intel Core i7 processor at 3.40 GHz.

Table 1 includes the running time obtained when executing the general parameters generation procedure in the testing computer with the bit lengths indicated in each case, where the bit length represents the maximum length in bits

of the parameters r, p_1, q_1, v_1, and v_2. The time displayed for each bit length represents the average time of the generation of 100 sets of parameters.

Table 1. General parameters generation running time

Length (bits)	32	64	96	128	160	192
Time (seconds)	0.30	4.24	25.21	57.19	128.83	300.57

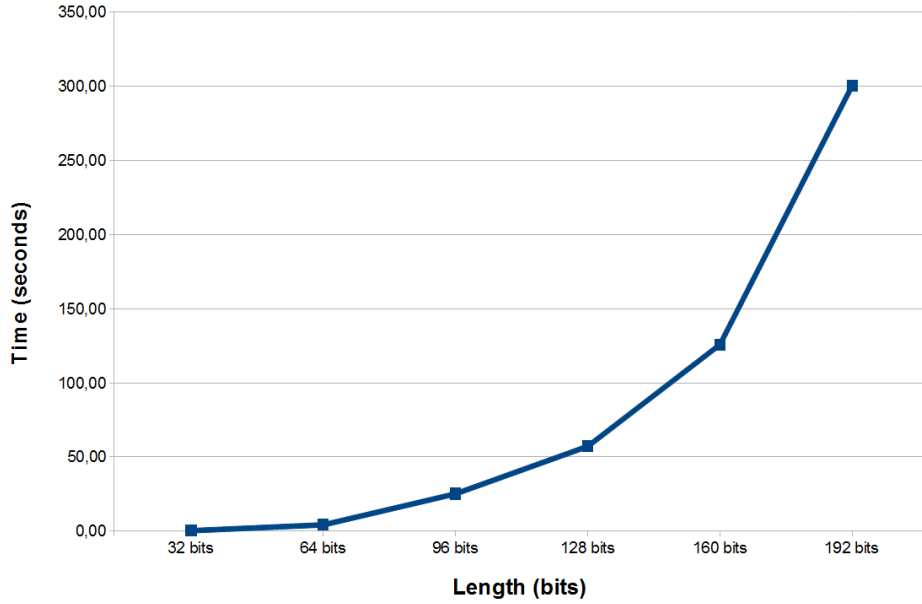As expected, the running time has an exponential shape, as it can be seen in Figure 2.

Fig. 2. General parameters' generation running time

Even though the running time obtained for large bit lengths is considerable, it is important to point out that the process of generating the general parameters is executed only once during the life cycle of that set of parameters. In the rest of operations (signature generation and verification, and signature disclosure) the running time obtained is less than 5 milliseconds even for parameters of 192 bits.

5   Conclusions

In this contribution we present a modification of the first group signature scheme described in [18]. In order to implement the scheme as a Java application, we have modified the scheme by adding a new requirement which mandates v_1 and v_2 to be both prime numbers, as explained in §2.1. With this modification, we force the number of non-trivial divisors of $\lambda(n)$ to be exactly 62, which facilitates the implementation in devices with limited resources as the application does not need to factor v_1 and v_2 in order to determine the actual number of non-trivial divisors of $\lambda(n)$.

The tests performed with the application allow us to confirm the expected difficulty in generating the system parameters for bit lengths greater than 64 bits. Nevertheless, as the system parameters generation procedure is only executed once by the PKC, it is not a limitation for deploying the group signature service in devices with limited resources, as they must only perform the signature generation and verification procedures.

Acknowledgment

References

1. Chaum, D., van Heyst, E.: Group signatures. Lecture Notes in Computer Science **547** (1991) 257–265
2. Camenisch, J., Stadler, M.: Efficient group signature schemes for large groups. Lecture Notes in Computer Science **1296** (1997) 410–424
3. Camenisch, J., Michels, M.: Separability and efficiency for generic group signature schemes. Lecture Notes in Computer Science **1666** (1999) 413–430
4. Bresson, E., Stern, J.: Efficient revocation in group signature. Lecture Notes in Computer Science **1992** (2001) 190–206
5. Chung, Y.F., Chen, T.L., Chen, T.S., Chen, C.S.: A study on efficient group-oriented signature schemes for realistic application environment. International Journal of Innovative Computing, Information and Control **8**(4) (2012) 2713–2727
6. Ogawa, K., Ohtake, G., Fujii, A., Hanaoka, G.: Weakened anonymity of group signature and its application to subscription services. IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences **E97-A**(6) (2014) 1240–1258
7. Emura, K., Miyaji, A., Omote, K.: An r-hiding revocable group signature scheme: Group signatures with the property of hiding the number of revoked users. Journal of Applied Mathematics **2014** (2014) 1–14
8. Menezes, A.J., van Oorschot, P.C., Vanstone, S.A.: Handbook of Applied Cryptography. CRC Press, Inc., Boca Raton, FL, USA (1996)

9. Ateniese, G., Camenisch, J., Joye, M., Tsudik, G.: A practical and provably secure coalition-resistant group signature scheme. Lecture Notes in Computer Science **1880** (2000) 255–270

10. Ateniese, G., de Medeiros, B.: Efficient group signatures without trapdoors. Lecture Notes in Computer Science **2894** (2003) 246–268

11. Nguyen, L., Safavi-Naini, R.: Efficient and provably secure trapdoor-free group signature schemes from bilinear pairings. Lecture Notes in Computer Science **3329** (2004) 89–102

12. Shoup, V., Gennaro, R.: Securing threshold cryptosystems against chosen ciphertext attack. Journal of Cryptology **15**(2) (2002) 75–96

13. Camenisch, J., Lysyanskaya, A.: Signature schemes and anonymous credentials from bilinear maps. Lecture Notes in Computer Science **3152** (2004) 56–72

14. Boneh, D., Boiyen, X., Shacham, H.: Short group signatures. Lecture Notes in Computer Science **3152** (2004) 41–55

15. Han, S., Wang, J., Liu, W.: An efficient identity-based group signature scheme over elliptic curves. Lecture Notes in Computer Science **3262** (2004) 417–429

16. Tan, Z.: An improved identity-based group signature scheme. Lecture Notes in Computer Science **3262** (2004) 417–429

17. Li, L., De-gong, D., Ying-liang, D.: An improved identity–based group signature scheme. In: International Conference on Information Technology, Computer Engineering and Management Sciences 2011 (ICM 2011). Volume 2. (2011) 269–271

18. Durán Díaz, R., Hernández Encinas, L., Muñoz Masqué, J.: Two proposals for group signature schemes based on number theory problems. Logic Journal of the IGPL **21**(4) (2013) 630–647

19. Potzmader, K., Winter, J., Hein, D., Hanser, C., Teufl, P., Chen, L.: Group signatures on mobile devices: Practical experiences. Lecture Notes in Computer Science **7904** (2013) 47–64

20. Spreitzer, R., Schmidt, J.M.: Group-signature schemes on constrained devices: the gap between theory and practice. In: First Workshop on Cryptography and Security in Computing Systems (CS2'14). (2014) 31–36

21. Susilo, W.: Short fail-stop signature scheme based on factorization and discrete logarithm assumptions. Theoretical Computer Science **410**(8) (2009) 736–744

22. NIST: Secure Hash Standard. National Institute of Standard and Technology, Federal Information Processing Standard Publication, FIPS 180-4. (2012)

23. Oracle Corporation: BigInteger (Java Platform SE 8). `http://docs.oracle.com/javase/8/docs/api/java/math/BigInteger.html`. (2014)

24. Oracle Corporation: Random (Java Platform SE 8). `http://docs.oracle.com/javase/8/docs/api/java/util/Random.html`. (2014)

25. Knuth, D.E.: The Art of Computer Programming, Volume 2 (3rd Ed.): Seminumerical Algorithms. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA (1997)