A Java implementation of a multisignature scheme

V. Gayoso Martínez, L. Hernández Encinas, and A. Martín Muñoz
Information Processing and Cryptography (TIC), Institute of Physical and Information Technologies (ITEFI)
Spanish National Research Council (CSIC), Madrid, Spain

Abstract—*Multisignature protocols are digital signature schemes that allow a group of users to sign a message so that the signature thus produced is valid only if all the members of the group participate in the signature process. In general, these schemes need the collaboration of a Trusted Third Party, which computes and securely stores some of the parameters associated to the scheme.*

In this work, we present our results and conclusions after implementing as a Java application a multisignature scheme based on the Integer Factorization Problem and the Subgroup Discrete Logarithm Problem.

Keywords: Authentication, Digital signatures, Java, Multisignatures, Trusted Third Parties

1. Introduction

In multisignature schemes, a group of users, typically denoted as G, signs a document such that the signature is valid only if all the members of the group take part in the process and the signature verifies a specific condition. These schemes have a direct application in corporate scenarios for signing contracts, validating agreements, etc.

From a naive point of view, the easiest way to carry out a multisignature consists in computing the individual signatures of all the signers and concatenating them, so the multisignature is composed of the sequence of individual signatures. However, this method is not practical for large groups of users, since the length of the multisignature is proportional to the number of signers.

The first practical multisignature scheme was proposed in [1], where a modification of the RSA cryptosystem was used in such a way that the RSA module consisted in the product of three primes instead of just two. In [2], another scheme was proposed where the signature length is similar to the length of a simple signature and shorter than the signature obtained from the scheme presented in [1]. However, this scheme can be used only if the cryptosystem is bijective, making it difficult to implement. Other proposals based on the RSA cryptosystem are, for example, those described in [3], [4], [5], [6], [7].

Regarding multisignature schemes based on the Discrete Logarithm Problem, in the scheme described in [8] the group of signers must cooperate in order to sign the message and send the signature to a given group of verifiers, but only through the union of all the verifiers it is possible to validate the multisignature. In addition to that, when producing the multisignature the signers not only must use their own private keys, but also the public key of each verifier, which is an important limitation [9], [10]. In the scheme proposed in [11], a multisignature can be performed only if the verifiers of the signature belong to a previously specified group, but apart from that limitation the scheme has some weaknesses [12], [13].

From a more general point of view, a multisignature scheme for a generic model of public key is presented in [14]. In that model, each one of the signers must have a certified public key with its corresponding private key, which must be generated by the signer himself. The signers interact completing a number of rounds, where in each round each signer receives a message, performs several calculations and sends the resulting message to the next signer. In this generic model, it should be computationally infeasible to forge a multisignature if there exists at least one honest signer in the group.

In comparison with the previous proposals, the multisignature scheme presented in [15] by one of the authors of this contribution has the advantage that each signer has his own private key, but all of them share the same public key. Besides, the procedure is secure, efficient, and independent of the number of signers. In addition to that, the signature is determined by all the signers in a certain pre-established order and the scheme allows to add new signers at the end of the signing chain, making it easier to update old signatures. Regarding the validation procedure, the scheme requires the verification of a certain property involving the signature itself, the original message, the number of signers, and some of the scheme's public parameters.

This work presents the results obtained when implementing a modified version of the multisignature scheme described in [15] using the Java language, where the modifications introduced have the goal of adapting the scheme to devices with limited resources and making the signing procedure more flexible by allowing the users to operate the scheme in any given order.

The rest of this paper is organized as follows: In Section 2, a detailed description of the mulsignature scheme is included. Section 3 describes the Java application developed in order to test the feasibility of the scheme. Section 4 provides a numerical example of the parameter and signature generation procedures. In Section 5, we offer to the readers the experimental results obtained with our Java application. Finally, our conclusions are presented in Section 6.

2. Description of the scheme

The security of the scheme described in [15] is based on the Integer Factorization Problem (IFP) and the Subgroup Discrete Logarithm Problem (SDLP).

As it is well known, the IFP can be described as follows [16]: 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 p_i are pairwise distinct primes and each $e_i \geq 1$.

Besides, the SDLP is defined as follows [16]: Let p be a prime and q a prime divisor of $p-1$. Let us consider g a generator of the unique subgroup H of $\mathbb{Z}_p^*$ of order q, and y an element in H. The problem is that of computing the integer x, $0 \leq x \leq q-1$, such that $y = g^x \pmod{p}$.

Let $G = \{U_1, U_2, \ldots, U_t\}$ be the group of t users allowed to perform signatures, and $\mathcal{T}$ the Trusted Third Party (TTP) managing the scheme's parameter generation process. The following subsections include all the details of the multisignature scheme.

2.1 Setup phase

In this phase, $\mathcal{T}$ generates the system parameters, its own private key, and the public key shared by the group. The steps that $\mathcal{T}$ must complete are the following:

1) $\mathcal{T}$ 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, and 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 [15], 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 must be selected so that the SDLP of order r in $\mathbb{Z}_n^*$ is computationally infeasible.
2) $\mathcal{T}$ computes the values n, the Euler function $\phi(n)$, and the Carmichael function $\lambda(n)$, where $n = p \cdot q$, $\phi(n) = (p-1)(q-1) = u_1 u_2 r^2 p_1 q_1$, and $\lambda(n) = \operatorname{lcm}(p-1, q-1) = 2 v_1 v_2 r p_1 q_1$.
3) $\mathcal{T}$ selects an element $\alpha \in \mathbb{Z}_n^*$ with multiplicative order r modulo n, and that fulfils the condition $\gcd(\alpha, \phi(n)) = 1$. The element α can be efficiently computed, as at this point $\mathcal{T}$ 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 [17]. 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 $\lambda(n) = 2 v_1 v_2 r p_1 q_1$ and all the factors are prime numbers. Once the value g is found, the generator is obtained through the following computation [17]:
$$\alpha = g^{\lambda(n)/r} \pmod{n}.$$
4) $\mathcal{T}$ generates a secret random number $s \in \mathbb{Z}_r^*$ and determines
$$\beta = \alpha^s \pmod{n}.$$
5) $\mathcal{T}$ publishes the values n, r, α, and β, while the elements p, q, and s are kept secret.
6) $\mathcal{T}$ sets up its private key by generating four random numbers $a_0, b_0, c_0, d_0 \in \mathbb{Z}_r^*$.
7) $\mathcal{T}$ determines the shared public key for G by computing the elements
$$\begin{aligned} P &= \alpha^{a_0} \cdot \beta^{b_0} \pmod{n} \equiv \alpha^h \pmod{n}, \\ Q &= \alpha^{c_0} \cdot \beta^{d_0} \pmod{n} \equiv \alpha^m \pmod{n}, \end{aligned}$$
   where $h \equiv (a_0 + s b_0) \pmod{r}$ and $m \equiv (c_0 + s d_0) \pmod{r}$.

2.2 User's private key generation

In order to prevent $\mathcal{T}$ from impersonating any member of G, the secret key of each user U_i is composed of four values, two of which are only known to U_i. With that goal in mind, the following steps must be completed:

1) U_i generates two secret integers $b_i, d_i \in \mathbb{Z}_r$ at random and sends the values $\alpha^{b_i} \pmod{n}$ and $\alpha^{d_i} \pmod{n}$ to $\mathcal{T}$ using a secure channel.
2) $\mathcal{T}$ computes
$$\begin{aligned} A_i &= \alpha^h (\alpha^{b_i})^{-s} \pmod{n} \equiv \alpha^{a_i} \pmod{n}, \\ C_i &= \alpha^m (\alpha^{d_i})^{-s} \pmod{n} \equiv \alpha^{c_i} \pmod{n}, \end{aligned}$$
   and sends the values A_i and C_i to the user U_i using a secure channel.
3) The private key of U_i is the set (A_i, b_i, C_i, d_i). Note that $\mathcal{T}$ can determine a_i and c_i since it knows h, k, α^{b_i}, and α^{d_i}, but it can compute neither b_i nor d_i because it cannot solve the SDLP. Similarly, U_i cannot compute the values a_i and c_i. As a consequence, both $\mathcal{T}$ and U_i have access to only two out of the four user's secret key parameters.

2.3 Parameter and key verification

Each member of the signer group, U_i, $1 \leq i \leq t$, may check the validity of the system parameters by verifying that $\alpha \not\equiv 1 \pmod{n}$ and $\alpha^r \equiv 1 \pmod{n}$.

Then, each signer, U_i, $1 \leq i \leq t$ can verify that their private key is related to the shared public key, by checking
$$P \equiv A_i \cdot \beta^{b_i} \pmod{n}, \quad Q \equiv C_i \cdot \beta^{d_i} \cdot \beta^{d_i} \pmod{n}. \quad (1)$$

This verification works because of the following chain of equivalences:
$$\begin{aligned} A_i \cdot \beta^{b_i} &\equiv \alpha^{a_i} \cdot \beta^{b_i} \equiv \alpha^{a_i + s \cdot b_i} \equiv \alpha^h \equiv P \pmod{n}, \\ C_i \cdot \beta^{d_i} &\equiv \alpha^{c_i} \cdot \beta^{d_i} \equiv \alpha^{c_i + s \cdot d_i} \equiv \alpha^k \equiv Q \pmod{n}. \end{aligned}$$

2.4 Multisignature 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 [18], either the signing user or $\mathcal{T}$ compute $\mathfrak{h}(M) = m$, where m represents the hash output.

In this contribution we have modified the scheme originally proposed in [15] so, given the set of signing users $G = \{U_1, U_2, \ldots, U_t\}$, they can complete the signature in any order, which reflects better the reality of organizations and the potential temporary (un)availability of the members of G. In order to generate the multisignature, the following steps must be completed:

1) The first signer, U_j, $1 \leq j \leq t$, must obtain the values F_j and g_j that compose his partial signature in the following way:

$$\begin{aligned} F_j &\equiv A_j \cdot C_j^m \pmod{n}, \\ g_j &\equiv b_j + m \cdot d_j \pmod{r}. \end{aligned} \tag{2}$$

Then, U_j sends the partial signature (F_j, g_j) to the next signer, U_k, $1 \leq k \leq t$, $k \neq j$.

2) The second signer, U_k, verifies U_j's signature by checking if the following equivalence holds:

$$P \cdot Q^m \equiv F_j \cdot \beta^{g_j} \pmod{n}.$$

If that is the case, U_k computes his partial signature for the message in the following way:

$$\begin{aligned} F_k &\equiv F_j \cdot A_k \cdot C_k^m \pmod{n} \\ &\equiv \alpha^{a_j + a_k + m(c_j + c_k)} \pmod{n}, \\ g_k &\equiv g_j + b_k + m \cdot d_k \pmod{r} \\ &\equiv b_j + b_k + m(d_j + d_k) \pmod{r}. \end{aligned} \tag{3}$$

3) Then, U_k sends the partial signature (F_k, g_k) to the next signer, U_l, $1 \leq l \leq t$, $l \neq j, k$, and the procedure is repeated until all the group members have signed the message. The signature computed by the last user represents the multisignature for M, denoted as (F, g).

2.5 Multisignature verification

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

$$P^t \cdot Q^{tm} \equiv F \cdot \beta^g \pmod{n}. \tag{4}$$

Equation (4) can be justified from expressions (1)–(3):

$$\begin{aligned} &F \cdot \beta^g \pmod{n} \equiv \\ &\equiv \alpha^{a_1 + \cdots + a_t + m(c_1 + \cdots + c_t)} \beta^{b_1 + \cdots + b_t + m(d_1 + \cdots + d_t)} \\ &\equiv \prod_{j=1}^{t} \alpha^{a_j} \cdot \beta^{b_j} \left(\alpha^{c_j} \cdot \beta^{d_j}\right)^m \pmod{n} \\ &\equiv \prod_{j=1}^{t} P \cdot Q^m = P^t \cdot Q^{t \cdot m} \pmod{n}. \end{aligned}$$

3. Java implementation of the scheme

The multisignature 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 application user has the option of converting the data from decimal (or text, in the case of the message to be sign) to hexadecimal and vice versa.

In all the cases where a random number is needed, the application uses the standard Java classes `BigInteger` [19] and `Random` [20], 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 [21].

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

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

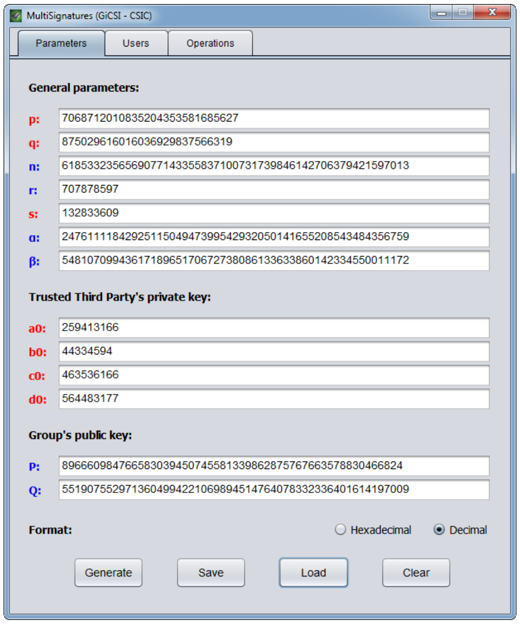
By calling the method `nextProbablePrime()` over the element `number`, the application obtains the first integer greater than `number` that is probably prime, where the probability that the number returned is composite does not exceed 2^{-100} [19].

Rearding the process of checking if a given value is a prime number, we have used the method `isProbablePrime(int certainty)` implemented by the `BigInteger` class, where `certainty` represents the measure of uncertainty tolerated by the method: if the call returns `true` the probability that the `BigInteger` element is prime exceeds $(1 - (1/2)^{\texttt{certainty}})$ [19], [22]. If the bit length of the number to be analysed is less than 100, the function makes 50 passes of the Miller-Rabin test [23]. On the other hand, if the bit length is higher, it makes a variable number of passes of the Miller-Rabin test (the precise number depends on the actual bit length: 27 for numbers with less than 256 bits, 15 for numbers with less than 512 bits, 8 for numbers with less than 768 bits, 4 for numbers with less than 1024 bits, and 2 for numbers having at least 1024 bits), but in addition to that it runs the Lucas-Lehmer test [23]. An example code would be the following:

```
boolean isprime =
    number.isProbablePrime(10);
```

3.1 *Parameters* panel

This panel includes the general parameters, $\mathcal{T}$'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 for the public elements n, r, α, β, P, and Q.



General parameters:

p: 7068712010835204353581685627
q: 87502961601603692983756 6319
n: 6185332356569077143355837100731739846142706379421597013
r: 707878597
s: 132833609
α: 2476111184292511504947399542932050141655208543484356759
β: 5481070994361718965170672738086133633860142334550011172

Trusted Third Party's private key:

a0: 259413166
b0: 44334594
c0: 463536166
d0: 564483177

Group's public key:

P: 896660984766583039450745581339862875767663578830466824
Q: 5519075529713604994221069894514764078332336401614197009

Format: ○ Hexadecimal ● Decimal

Fig. 1: *Parameters* pannel

There are four buttons available in this panel:

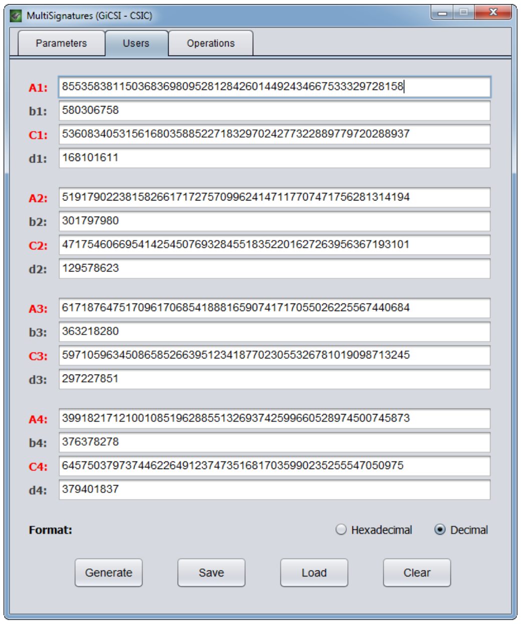
- *Generate*: It computes all the parameters according to the steps 1-7 of the procedure described in §2.1.
- *Save*: It allows the user to save either the public data or all the data included in this panel. The information is stored in a file using an XML structure.
- *Load*: It allows the user to overwrite 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.

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 2. We remind the reader that the values b_i and d_i are known only to U_i, while only $\mathcal{T}$ knows the value of the elements a_i and c_i.

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



A1: 855358381150368369809528128426014492434667533329728158
b1: 580306758
C1: 5360834053156168035885227183297024277322889779720288937
d1: 168101611

A2: 5191790223815826617172757099624147117707471756281314194
b2: 301797980
C2: 4717546066954142545076932845518352201627263956367193101
d2: 129578623

A3: 6171876475170961706854188816590741717055026225567440684
b3: 363218280
C3: 5971059634508658526639512341877023055326781019098713245
d3: 297227851

A4: 3991821712100108519628855132693742599660528974500745873
b4: 376378278
C4: 645750379737446226491237473516817035990235255547050975
d4: 379401837

Format: ○ Hexadecimal ● Decimal

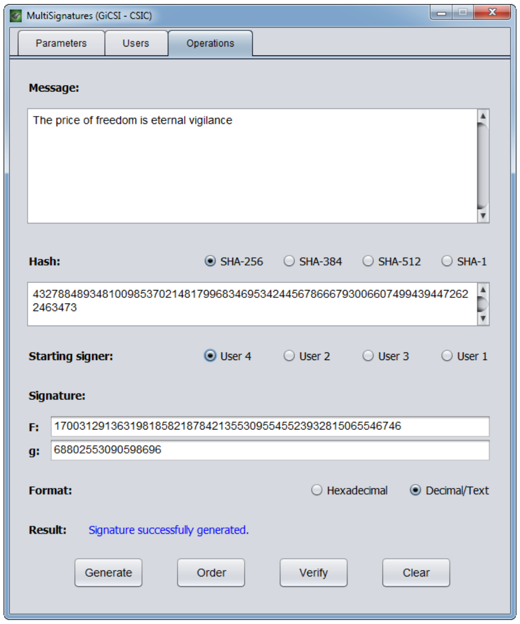
Fig. 2: *Users* panel.

- *Generate*: It generates all the private elements associated to the private keys of the users according to the steps 1 and 2 of the procedure described in §2.2.
- *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 3:

- *Generate*: It generates the multisignature of the text message provided manually by the user according to the steps 1-3 described in §2.4. In order to obtain the elements F and g associated to the signature, it is mandatory to select in the panel the hash function and the starting signing user.
- *Order*: By selecting this button, the application changes the order of the users randomly, with the condition that the new order must be different from the previous one. Once a specific order is displayed, the user can select the starting signer by checking the proper element.
- *Verify*: It allows to verify if the multisignature provided by the user corresponds to the text message entered in its text box, as described in Section 2.5.
- *Clear*: It deletes the content of all the text boxes displayed in this panel.



Message:

The price of freedom is eternal vigilance

Hash: ● SHA-256 ○ SHA-384 ○ SHA-512 ○ SHA-1

4327884893481009853702148179968346953424456786667930066074994394472622463473

Starting signer: ● User 4 ○ User 2 ○ User 3 ○ User 1

Signature:

F: 1700312913631981858218784213553095545523932815065546746

g: 68802553090598696

Format: ○ Hexadecimal ● Decimal/Text

Result: Signature successfully generated.

Fig. 3: *Operations* panel.

4. Numerical example

This section provides the details of the signature process depicted in Figures 1, 2, and 3, where the selected bit length for the base elements r, p_1, q_1, v_1, and v_2 is 32. This bit length is clearly inadequate from a security perspective, but making this choice allows us to manage smaller numbers in order to facilitate the comprehension of the example.

After selecting the option *Generate*, the application randomly produces the prime values $r = 707878597$, $p_1 = 3641604649$, and $q_1 = 303316411$. Then, the application enters a loop where it randomly generates the prime values v_1 and v_2 and computes p and q, exiting the loop once it checks that the values p and q are both prime numbers. In the example, the first values that satisfy that condition are $v_1 = 1371067121$, and $v_2 = 2037689777$, producing $p = 7068712010835204353581685627$ and $q = 875029616016036929837566319$, which can be represented using 93 and 90 bits, respectively.

Then, after computing $\phi(n) = 6185332356569077143355837092787998219291465096002345068$ (a 183-bit number), $\lambda(n) = 436892172102858277501773167241839791017969222 2$, and the 62 non-trivial divisors of $\lambda(n)$, the application enters a loop for computing a generator α such that it is coprime with $\phi(n)$. In the example, the generator thus calculated is $\alpha = 2476111184292511504947399542932050141655208543484356759$.

Next, the application randomly generates the value $s = 132833609$, which must be coprime with r. Using α and s, the application computes $\beta = 5481070994361718965170672738086133633860142334550011172$.

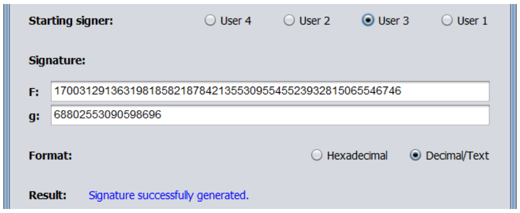
After that, it randomly generates the elements of the $\mathcal{T}$'s private key ($a_0 = 259413166$, $b_0 = 44334594$, $c_0 = 463536166$, and $d_0 = 564483177$) and computes the elements of the public key ($P = 896660984766583039450745581339862875767663578830466824$ and $Q = 5519075529713604994221069894514764078332336401614197009$).

As for the private keys of the signing users, in the example the application generates the following values: $a_1 = 855358381150368369809528128426014492434667533329728158$, $b_1 = 580306758$, $c_1 = 5360834053156168035885227183297024277322889779720288937$, $d_1 = 168101611$, $a_2 = 5191790223815826617172757099624147117707471756281314194$, $b_2 = 301797980$, $c_2 = 4717546066954142545076932845518352201627263956367193101$, $d_2 = 129578623$, $a_3 = 6171876475170961706854188816590741717055026225567440684$, $b_3 = 363218280$, $c_3 = 5971059634508658526639512341877023055326781019098713245$, $d_3 = 297227851$, $a_4 = 3991821712100108519628855132693742599660528974500745873$, $b_4 = 376378278$, $c_4 = 645750379737446226491237473516817035990235255547050975$, and $d_4 = 379401837$.

Given the example message (the quote "The price of freedom is eternal vigilance") and the selected hash function (SHA-256), before computing the multisignature the application calculates the message's digest, whose representation in hexadecimal is `09917EFCA9E63C6BE3F5710D4E146146A152B64CE2E1DCDBBAAC3F6EBD6E19F1`. When considered as an integer modulo r, the value associated to the message is 70616700.

In the example, after using the *Order* button the distribution of signing users obtained is 4-2-3-1. If we select User 4 as the starting signer (see Figure 3), the elements forming the subsequent signatures calculated by the application are the following: $F_4 = 5035768167055077718477864679949082104696821277077281113$, $g_4 = 26792106079256178$, $F_2 = 2746734222845697561460978062345247914787770187195849232$, $g_2 = 35942521127858258$, $F_3 = 5492470811037444492215183076659234465569196156819515847$, $g_3 = 56931771476788238$, $F_1 = 1700312913631981858218784213553095545523932815065546746$, and $g_1 = 68802553090598696$. The multisignature resulting from this process is the signature computed by the last user, so $(F, g) = (F_1, g_1)$.

If, given the initial distribution 4-2-3-1, we had selected User 3 has the starting signer (see detail in Figure 4), the signatures calculated by the application would contain the elements $F_3 = 1525558802257083259276411227735843971228217026108102837$, $g_3 = 20989250348929980$, $F_1 = 4266748247603434305823011807817369532998324145115483942$, $g_1 = 32860031962740438$, $F_4 = 5269682463891211812109596999207768861301755362219881607$, $g_4 = 59652138041996616$, $F_2 = 1700312913631981858218784213553095545523932815065546746$, and $g_2 = 68802553090598696$, where the resulting multisignature is $(F, g) = (F_2, g_2)$. As it can be observed, the multisignature obtained is the same, as the order does not affect its final result, even though the partial signatures are different in each case. The same multisignature (F, g) would be produced if, for example, the initial distribution of users had been 1-2-3-4 and we had selected any of the four users as the starting signer.


Starting signer: ○ User 4 ○ User 2 ● User 3 ○ User 1

Signature:

F: 1700312913631981858218784213553095545523932815065546746

g: 68802553090598696

Format: ○ Hexadecimal ● Decimal/Text

Result: Signature successfully generated.

Fig. 4: Signature generation example.

5. 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.

As expected, the running time has an exponential shape, as it can be seen in Figure 5.

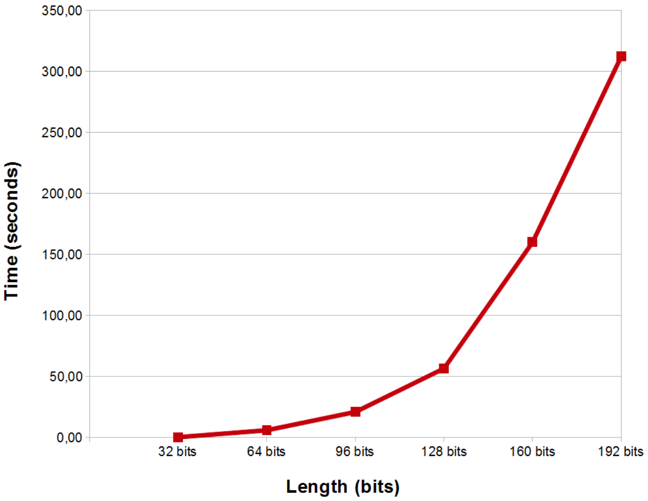


Fig. 5: General parameters' generation running time

Most of the parameter generation running time is due to the operations with prime numbers and `BigInteger` elements: obtaining the first prime number bigger than a certain value (method `nextProbablePrime()`) and checking if a candidate value is a prime number (method `isProbablePrime()`). Table 2 shows the average number of executions of the pieces of code calling those methods.

Figures 6 and 7 show graphically the information contained in Table 2. The main reason for the increase in the execution time in now clear: not only the application spends more time in each call to the methods `nextProbablePrime()` and `isProbablePrime()`, as a result of dealing with bigger numbers, but it also needs to call those methods more times, as the probability of $p = u_1 r p_1 + 1$ and $q = u_2 r q_1 + 1$ being prime numbers is lower as the bit length of those numbers increase.

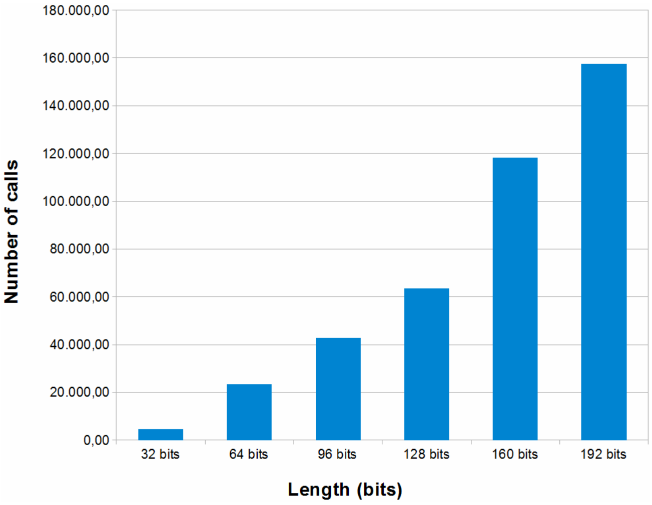


Fig. 6: Number of calls to the method `nextProbablePrime()`

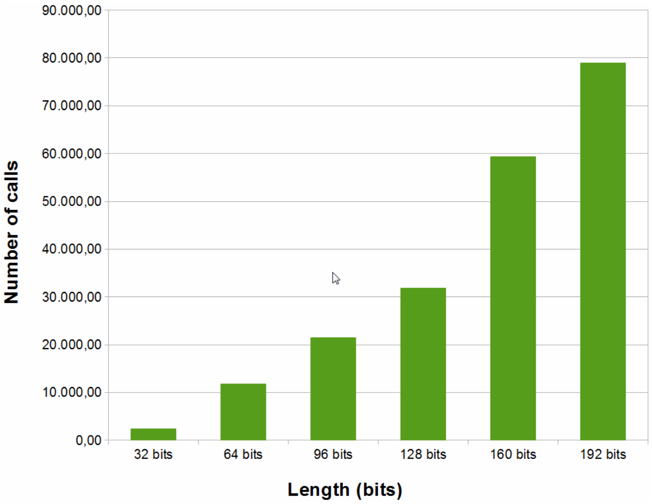


Fig. 7: Number of calls to the method `isProbablePrime()`

Table 1: General parameters generation running time

Length (bits)	32	64	96	128	160	192
Time (seconds)	0.37	6.06	21.03	56.70	160.11	312.36

Table 2: Number of calls to some methods implementd by the `BigInteger` class

Length (bits)	32	64	96	128	160	192
`nextProbablePrime()`	4,583.88	23,370.20	42,538.68	63,448.86	118,089.20	157,435.14
`isProbablePrime()`	2,337.15	11,802.72	21,416.38	31,883.81	59,279.56	78,977.87

6. Conclusions

In this contribution we have presented a modification of the multisignature scheme described in [15]. 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 $\lambda(n)$. In addition to that, we have modified the scheme so the members of the group can sign a certain message in any given order.

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 Trusted Third Party, it is not a limitation for implementing this multisignature scheme in other devices that most of the times will only perform the signature generation and verification procedures.

Acknowledgment

This work has been partially supported by Comunidad de Madrid (Spain) under the project S2013/ICE-3095-CM (CIBERDINE).

References

[1] N. K. Itakura, K., “A public-key cryptosystem suitable for digital multisignatures,” *NEC Research & Development*, vol. 71, pp. 1–8, 1983.

[2] T. Okamoto, “A digital multisignature scheme using bijective public-key cryptosystems,” *ACM Transactions on Computer Systems*, vol. 6, no. 4, pp. 432–441, 1988.

[3] A.-F. M. Aboud, S.J., “A new multisignature scheme using re-encryption technique,” *Journal of Applied Sciences*, vol. 7, pp. 1813–1817, 2007.

[4] K.-T. Harn, L., “New scheme for digital multisignature,” *Electronics Letters*, vol. 25, pp. 1002–1003, 1989.

[5] H.-L. Kiesler, T., “Rsa blocking and multisignature schemes with no bit expansion,” *Electronics Letters*, vol. 26, pp. 1490–1491, 1990.

[6] P.-S. K. K. W. D. Park, S., “Two efficient RSA multisignature schemes,” *Lecture Notes in Computer Science*, vol. 1334, pp. 217–222, 1997.

[7] L.-E. L. J. Pon, S.F., “Dynamic reblocking RSA-based multisignatures scheme for computer and communication networks,” *IEEE Communications Letters*, vol. 6, no. 1, pp. 43–44, 2002.

[8] Y.-S. Laih, C.S., “Multisignature for specified group of verifiers,” *Journal of Information Science and Engineering*, vol. 12, no. 1, pp. 143–152, 1996.

[9] W. He, “Weakness in some multisignature schemes for specified group of verifiers,” *Information Processing Letters*, vol. 83, no. 2, pp. 95–99, 2002.

[10] S. Yen, “Cryptanalysis and repair of the multi-verifier signature with verifier specification,” *Computers & Security*, vol. 15, no. 6, pp. 537–544, 1996.

[11] X.-G. Zhang, Z., “New multisignature scheme for specified group of verifiers,” *Applied Mathematics and Computation*, vol. 157, pp. 425–431, 2004.

[12] W.-X. K. K. Lv, J., “Security of a multisignature scheme for specified group of verifiers,” *Applied Mathematics and Computation*, vol. 166, pp. 58–63, 2005.

[13] Y.-K. Yoon, E.J., “Cryptanalysis of zhang-xiao’s multisignature scheme for specified group of verifiers,” *Applied Mathematics and Computation*, vol. 170, pp. 226–229, 2005.

[14] N.-G. Bellare, M., “Multi-signatures in the plain public-key model and a general forking lemma,” in *13th ACM conference on Computer and Communications Security (CCS’06)*, 2006, pp. 390–399.

[15] R. Durán Díaz, L. Hernández Encinas, and J. Muñoz Masqué, “A multisignature scheme based on the sdlp and on the ifp,” *Lecture Notes in Computer Science*, vol. 6694, pp. 135–142, 2011.

[16] A. J. Menezes, P. C. van Oorschot, and S. A. Vanstone, *Handbook of Applied Cryptography*. Boca Raton, FL, USA: CRC Press, Inc., 1996.

[17] W. Susilo, “Short fail-stop signature scheme based on factorization and discrete logarithm assumptions,” *Theoretical Computer Science*, vol. 410.

[18] NIST, *Secure Hash Standard*, National Institute of Standard and Technology, Federal Information Processing Standard Publication, FIPS 180-4, 2012.

[19] Oracle Corporation, *BigInteger (Java Platform SE 8)*, http://docs.oracle.com/javase/8/docs/api/java/math/BigInteger.html, 2014.

[20] ——, *Random (Java Platform SE 8)*, http://docs.oracle.com/javase/8/docs/api/java/util/Random.html, 2014.

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

[22] Oracle Corporation, *OpenJDK - jdk8 - BigInteger.java)*, http://hg.openjdk.java.net/jdk8/jdk8/jdk/file/00cd9dc3c2b5/src/share/classes/java/math/BigInteger.java, 2015.

[23] R. E. Crandall and C. Pomerance, *Prime Numbers: A computational perspective*. Springer, New York, USA: Springer, 2005.