Off-line/On-line Generation of RSA Keys with Smart Cards

[Published in S.-P. Shieh, Ed., *2nd International Workshop for Asian Public Key Infrastructures*, pp. 153–158, Taipei, Taiwan, Oct. 30 – Nov. 1, 2002.]

Nathalie Feyt, Marc Joye, David Naccache, and Pascal Paillier

Gemplus Card International, France
{nathalie.feyt, marc.joye, david.naccache, pascal.paillier}@gemplus.com

Abstract. Standard bodies and organizations are pushing for increasingly larger RSA keys. Today, RSA keys range from 512 bits to 2048 bits and some bodies envision 4096-bit RSA keys in the near future.
This paper devises a new methodology for generating RSA keys. Contrary to what is usually done, the key generation is divided into two phases. The first phase is performed off-line, before the input parameters are even known. The second phase is performed on-line by the smart card once the input parameters are known, and is meant to be very fast. Compared to the fastest reported method ([4]), our solution —or more precisely the on-line phase thereof, is conceptually more advanced and achieves extreme execution speeds as generating 1024-bit or 2048-bit RSA keys amounts to practical running times lowered by *several orders of magnitude*. Moreover, our technique achieves on-line generation of RSA keys of arbitrary length from a small set of seeds computed during the off-line phase. Subsequently, in addition to be fast and flexible, our solution also features attractively low memory requirements.

Keywords. Cryptography, RSA keys, smart cards, PKIs.

1 Introduction

Public-key cryptography faces the problem of the authentication of the public keys: How can we be sure that a pair of public key/user's identity are matching. A related problem is how to distribute public keys trustfully. These issues are proved to be the bottleneck for a wide deployment of public-key systems, such as the RSA cryptosystem [8]. It is here the Public Key Infrastructures (PKIs) come into play [6]. The idea behind PKI is fairly simple. It basically consists in producing an analogue of a phone directory. In the 'PKI directory', one should be able to find a user (or more generally an application) and the corresponding public key. Of course, this directory must in some sense be certified. To this purpose, in addition to the name and the public key, the directory also contains a certificate issued by a Certification Authority (CA). Furthermore, in order to make the system inter-operable, each user belongs to a domain and each domain has its own associated certification authority. Then, when the user has to be identified and authenticated, he just produces the certificate issued by the CA of his domain. This certificate is a digital signature by the CA on at least the user's public key and his identity (along with some other credentials, if needed).

At present, when smart cards hold public keys, it is common that a companion certificate is issued by a CA, for each embedded public key. A certificate has a cost, even if the corresponding public key is never used by the card holder. Moreover, the memory which is used to store the keys, generated off-board, has to be paid even if the end-user never uses the public-key functions.

A cheaper solution may be to have an on-board key generation. So, sets of keys will be generated only if they will be used. A second advantage is that there is more memory available. Furthermore, we should note that on-board key generation is more *secure* as the private keys are only known by the card holder, i.e., the end-user. Although attractive, this second solution may be too slow for certain applications. The on-board generation of a complete 2048-bit RSA key takes 30 seconds with the very efficient algorithm of [4], on average.

This paper is aimed at presenting a mixed off-board/on-board solution. The variable and time-consuming part is performed off-line: it produces small seeds that are used in the second, fast, on-line part of the generation of the keys themselves. Moreover, it can virtually accommodate any RSA bit-length and any public exponent. As a result, we obtain a *fast*, *flexible*, *on-board* RSA key generation algorithm.

The rest of this paper is organized as follows. In the next section, we introduce the notations and briefly review the RSA cryptosystem. In Section 3, building on the algorithm of [3], we present our efficient off-line/on-line key generation algorithm. Finally, we conclude in Section 4.

2 The RSA Cryptosystem

The RSA cryptosystem [8] is a pair of algorithms: a public algorithm (encryption or signature verification) and a private algorithm (decryption or signature generation). Its security relies on the difficulty of integer factorization.

Each user chooses two large primes p and q, and publishes the product $N = pq$. Next, he chooses a public exponent e that is relatively prime to $(p-1)$ and $(q-1)$. Finally, he computes the secret exponent d according to

$$ed \equiv 1 \pmod{\operatorname{lcm}(p-1, q-1)} \ . \tag{1}$$

The public parameters are (N, e) and the secret parameters are (p, q, d).

To send a message m to Bob, Alice looks to Bob's public key (e, N) and forms the ciphertext $c = \mu(m)^e \bmod N$, where μ is an appropriate padding function (e.g., OAEP [1]). Next, to recover the plaintext m, Bob uses his secret decryption key d to obtain $\mu(m) = c^d \bmod N$ and so m.

This encryption scheme can be converted into a signature scheme. If Bob wants to sign a message m, he uses his secret key d to compute the signature $s = \mu(m)^d \bmod N$ (a valid choice for μ is PSS [2]). Next, he sends m and s to Alice. Then Alice can verify that s corresponds to Bob's signature on message m by checking whether $s^e \equiv \mu(m) \pmod N$ where e is the public exponent of Bob.

2.1 General moduli

RSA moduli are not restricted to products of two large primes. It is for example possible to work with moduli consisting of 3 or more factors. If $N = \prod_{i \geq 2} p_i$ (with p_i large primes) denotes an RSA modulus then public exponent e must be co-prime to $\lambda(N)$ where λ is the Carmichæl function and secret exponent d is defined according to $ed \equiv 1 \pmod{\lambda(N)}$.

2.2 Chinese remaindering

It is possible to speed up the private operation (i.e., decryption or signature generation) through Chinese remaindering [7]: the private operation is carried out modulo each prime factor of modulus N and these partial results are then recombined. For example, if $N = pq$, we set $d_p = d \bmod (p-1)$, $d_q = d \bmod (q-1)$ and compute $R_p = c^{d_p} \bmod p$ and $R_q = c^{d_q} \bmod q$. Next, letting $i_q = 1/q \bmod p$, we obtain $c^d \bmod N$ as

$$\mathrm{CRT}(R_p, R_q) = R_q + q[i_q(R_p - R_q) \bmod p] \ . \qquad (2)$$

This mode of operation is referred to as *CRT mode* and the secret parameters are (p, q, d_p, d_q, i_q).

3 Generation of RSA Keys

As briefly mentioned in the previous section, the RSA setup requires the values of public exponent e and of the key length (i.e., the length of modulus N). We let ℓ denote the bit-length of N. Then, on input e and ℓ (determined by the application), the card must possess two primes p and q so that

(i) $(p-1)$ and $(q-1)$ are co-prime to e, and
(ii) $N = pq$ is exactly an ℓ-bit integer.

The obvious solution is to let the card randomly compute on-board values for p and q from e and ℓ. The drawback in this approach is the running time; typically, given the state-of-the-art, a 2048-bit RSA key requires 30 seconds. Another solution consists in pre-computing values for p and q for various pairs (e, ℓ) and to store those values in EEPROM-like, non volatile memory. The drawback here is either the cost —EEPROM-like memory is expensive— (when there are lots of chosen pairs (e, ℓ)) or the lack of interoperability (when there are few chosen pairs (e, ℓ)).

In the sequel, we are looking for *quick* and *cheap* processes for producing two primes p and q satisfying Conditions (i) and (ii). To ensure that $N = pq$ is exactly an ℓ-bit integer, it suffices to choose $p \in \left[\lceil 2^{(\ell-\ell_0)-1/2} \rceil, 2^{\ell-\ell_0} - 1\right]$ and $q \in \left[\lceil 2^{\ell_0-1/2} \rceil, 2^{\ell_0} - 1\right]$ for some $1 < \ell_0 < \ell$. Indeed, we then have $N \geq \min(p)\min(q) \geq 2^{\ell-1}$ and $N \leq \max(p)\max(q) < 2^{\ell}$, as required. Consequently, Condition (ii) above reduces to finding primes in a range of the form $\left[\lceil 2^{\ell_0-1/2} \rceil, 2^{\ell_0} - 1\right]$.

3.1 First solution

A very natural yet cumbersome solution consists in precomputing and writing in the card's non-volatile memory a set of integer values $\{\sigma_i\}$ such that for each i, $\mathtt{PRNG}(\sigma_i)$ yields a prime number. Here, PRNG denotes a *pseudo-random number generator*, that is, a deterministic function that expanses fixed-length integers to bit-streams of desirable length (for instance 1024 bits). Once the card is personalized with its own set of seeds, it simply computes $p = \mathtt{PRNG}(\sigma_i)$ whenever the generation of a prime number is required. Each time, a counter for i is decremented so that the routine will jump one seed ahead in non-volatile memory at the next execution.

Clearly, the seeds σ_i should be as short as possible in order to minimize the memory space needed to store them all in the card. On the other hand, these have to be large enough to prevent anyone from being able to guess their value and anticipate prime numbers the card is meant to generate during its lifetime (as this would lead to a complete breaking). The on-line phase, i.e., the sequence of computations carried out by the card when some prime is generated, is trivially simple (one single invocation of PRNG) and can be extremely fast. There exist, indeed, numerous ways of basing a pseudo-random generator on a cryptographically secure hash function or block-cipher that achieve highest execution throughputs. Unfortunately, the off-line phase necessary to precompute the seeds may be quite long. Indeed, the process of randomly picking some σ such that $\mathtt{PRNG}(\sigma)$ is prime cannot be much smarter than applying primality tests on $\mathtt{PRNG}(\sigma)$ for random values of σ. This yields roughly

$$\Pr_{\sigma}\left[\mathtt{PRNG}(\sigma) \text{ prime}\right] \approx \frac{1}{|\mathtt{PRNG}| \cdot \ln 2} \quad .$$

Therefore, in the common setting where $|\mathtt{PRNG}| = 1024$, about 709.78 primality tests are necessary for selecting a single seed, on average. This complexity may be halved down by forcing $\mathtt{PRNG}(\sigma)$ to be odd for any σ (and we would also have to take into account the constraint that $\gcd(p-1, e) = 1$ with respect to the RSA cryptosystem).

3.2 Second solution

The off-line phase of the previous solution is somewhat time-consuming. This section investigates how to speed up this phase (at the expense, however, of a slightly slower on-line phase).

3.2.1 Granularity

An efficient prime generation algorithm has been devised in [4]. It exploits the elementary property that a prime number has no trivial factors. Let $\Pi = \prod_{p_i \text{ prime}} p_i$ be the product of the first small primes. The algorithm of [4] proceeds in two steps. The first step consists in generating a number relatively prime to Π, say k, and the second step is, given k, the construction of a prime candidate q satisfying $\gcd(q, \Pi) = \gcd(k, \Pi) = 1$. If candidate q is not prime, then k is updated and a new prime candidate is constructed, and so on. Because candidate q is such that it is already prime to the first primes (namely, to Π), the probability that it is prime is high and so few iterations have to be performed until a prime q is found.

Building on [3], we now present an algorithm that works for *any* given bit-length ℓ_0 for prime q being generated (the generation of p is similar). We assume that we are given a lower bound B_0 for ℓ_0: $\ell_0 \geq B_0$. For example, one can choose $B_0 = 256$ since a factor smaller than 256 bits is nowadays considered insecure. We define $\Pi = \prod_{i=1}^{43} p_i = 2 \cdot 3 \cdots 191 < 2^{256}$. (More generally, Π is defined as the largest product of the consecutive first primes so that $\prod_i p_i < 2^{B_0}$.) We also define the unique integers v and w satisfying

$$\begin{cases} \lfloor 2^{\ell_0 - 1/2} \rfloor \leq v\Pi < \lfloor 2^{\ell_0 - 1/2} \rfloor + \Pi \\ 2^{\ell_0} - \Pi < w\Pi \leq 2^{\ell_0} \end{cases} \tag{3}$$

namely, $v = \left\lceil \frac{\lfloor 2^{\ell_0 - 1/2} \rfloor}{\Pi} \right\rceil$ and $w = \left\lfloor \frac{2^{\ell_0}}{\Pi} \right\rfloor$.

Next, given an element $k \in \mathbb{Z}_\Pi{}^*$ (that is, $k \in \{0, \ldots, \Pi - 1\}$ and $\gcd(k, \Pi) = 1$), we construct prime candidate q as

$$q = k + j\Pi \quad \text{for some } j \in [v, w-1] \ . \tag{4}$$

[An efficient way for generating invertible elements in $\mathbb{Z}_\Pi{}^*$ is presented in § 3.2.2; see Lemma 1.]

As $k \in \mathbb{Z}_\Pi{}^*$, it follows that $\gcd(q, \Pi) = \gcd(k, \Pi) = 1$. Moreover, we have $\min(q) = 1 + v\Pi \geq \lceil 2^{\ell_0 - 1/2} \rceil$ and $\max(q) = (\Pi - 1) + (w - 1)\Pi \leq 2^{\ell_0} - 1$, or equivalently, $q \in \left[\lceil 2^{\ell_0 - 1/2} \rceil, 2^{\ell_0} - 1 \right]$. If the so-obtained q is not prime, we update k as $k \leftarrow ak \pmod{\Pi}$ with $a \in \mathbb{Z}_\Pi{}^*$. This implies that the updated k also belongs to $\mathbb{Z}_\Pi{}^*$ since $\mathbb{Z}_\Pi{}^*$ is a group.

The usual way to test the primality (or more exactly, the pseudo-primality) of a number is the Rabin-Miller test. We refer the reader to [5, Chapter 4] for details on Rabin-Miller test and variants thereof.

A description of our modified algorithm is depicted in Fig. 1. This algorithm outputs an ℓ_0-bit prime q, for any value for ℓ_0.

3.2.2 Storage efficiency

A direct application of the previous algorithm (Fig. 1) requires for *each* RSA key bit-length the storage of k and j in order to re-construct q. A first improvement consists in constructing j from a short random seed, say 64-bit long, used as the input of a mask generating function (MGF), rather than randomly choosing j as in Step 2 of Fig. 1 (a concrete construction of MGF can be found in [2, Appendix A]). Let σ be a 64-bit random

```
Input:  parameters $\ell_0$, $e$, and
        $a$ (of large order) in $\mathbb{Z}_{\varPi}{}^*$
Output: a prime $q \in \left[\lceil 2^{\ell_0 - 1/2} \rceil, 2^{\ell_0} - 1\right]$

1. Compute $v = \left\lceil \frac{\lfloor 2^{\ell_0 - 1/2} \rfloor}{\varPi} \right\rceil$ and $w = \left\lfloor \frac{2^{\ell_0}}{\varPi} \right\rfloor$
2. Randomly choose $j \in_R \{v, \ldots, w-1\}$ and set $l \leftarrow j\varPi$
3. Randomly choose $k \in_R \mathbb{Z}_{\varPi}{}^*$
4. Set $q \leftarrow k + l$
5. If ($q$ is not prime) or ($\gcd(e, q-1) \neq 1$) then
   (a) Set $k \leftarrow ak \pmod{\varPi}$
   (b) Go to Step 4
6. Output $q$
```

Fig. 1. RSA Prime Generation Algorithm.

value. Given ℓ_0, the values of v and w are computed according to Eq. (3) and j is defined as $\mathrm{MGF}_1(\sigma) \pmod{(w-v)} + v$. This simple improvement drastically reduces the amount of EEPROM-like memory ne needed as only the values of σ and k have to be stored (the value of $\varPi$ is in code memory).

Further memory can be saved by observing that if $k_{(0)}$ denotes the initial value of $k \in \mathbb{Z}_{\varPi}{}^*$ then the primes generated by our algorithm have the form

$$q = a^{f-1} k_{(0)} \bmod \varPi + j\varPi \tag{5}$$

where f is the number of failures of the test in Step 4 (Fig. 1). The second observation is that a value $k_{(0)} \in \mathbb{Z}_{\varPi}{}^*$ can be easily computed from a short random seed using an MGF. We use the following lemma.

Lemma 1 ([3, Proposition 2]). *For all* $b, c \in \mathbb{Z}_{\varPi}$ *s.t.* $\gcd(b, c, \varPi) = 1$*, we have*

$$\left[c + b(1 - c^{\lambda(\varPi)})\right] \in \mathbb{Z}_{\varPi}{}^*$$

where $\lambda(\varPi)$ *denotes the Carmichæl function of* $\varPi$*.*

As an immediate corollary, if $b \in \mathbb{Z}_{\varPi}{}^*$ so do $(\varPi - b)$ and consequently $\left[c + b(c^{\lambda(\varPi)} - 1)\right] \pmod{\varPi}$. Therefore, given the random seed σ, we can form $k_{(0)}$ as

$$k_{(0)} = \left[\mathrm{MGF}_2(\sigma) + b^{\mathrm{MGF}_3(\sigma)}(\mathrm{MGF}_2(\sigma)^{\lambda(\varPi)} - 1)\right] \pmod{\varPi} \tag{6}$$

where b is an element of large order in $\mathbb{Z}_{\varPi}{}^*$ (preferably of order $\lambda(\varPi)$).

The first and second improvements imply that only the value of σ (typically, a 64-bit value) and the different values of f for desired key lengths need to be stored in EEPROM-like memory. For RSA moduli up to 2048 bits, numerical experiments show that a upper bound for f is certainly 2^8 (hence f can be coded on one byte).

For example, in order to be able to produce RSA moduli ranging from 512 to 2048 bits with a granularity of 32 bits (they are 49 possible such key lengths), a card needs to store σ (8 bytes) and values for f for primes p and q ($2 \times 49 = 98$ bytes), that is, a total of 106 bytes (848 bits) in EEPROM-like memory.

A last trick to reduce the needed memory is to write in code-memory several values of Π (and the corresponding $\lambda(\Pi)$) for different key lengths by noting that a larger value for Π leads to smaller values for f.

3.2.3 Interoperability We now consider Condition (i), namely we want that RSA primes p and q verify the relation $\gcd(p-1, e) = \gcd(q-1, e) = 1$ where e denotes the public exponent.

From Eq. (5), we observe that a prime, say q, generated by the algorithm of Fig. 1 satisfies $q = a^{f-1}k_{(0)} \bmod \Pi + j\Pi$. Hence, provided that e divides Π, we have

$$q \equiv a^{f-1}k_{(0)} \pmod{e} \ . \tag{7}$$

Moreover, assuming that e is prime, the condition $\gcd(e, q-1) = 1$ reduces to $\gcd(e, q-1) \neq e \iff q \not\equiv 1 \pmod{e}$. Consequently, if public exponent e is a prime dividing Π then the condition $\gcd(e, q-1) = 1$ is fulfilled whenever $a^{f-1}k_{(0)} \not\equiv 1 \pmod{e}$. A way to achieve this consists in choosing $a \equiv 1 \pmod{e}$ (but of large order as an element of $\mathbb{Z}_{\Pi}{}^*$) and to force $k_{(0)}$ so that $k_{(0)} \not\equiv 1 \pmod{e}$. Hence, the resulting prime q satisfies $q \equiv k_{(0)} \not\equiv 1 \pmod{e}$, as desired.

The card does not know *a priori* the value of exponent e; the value of e is determined by the application. However, most applications (i.e., $> 95\%$) use for e values in the set $\{3, 17, 2^{16}+1\}$ (notice that all these values are prime). In order to cover the largest set of applications, we thus choose parameter a such that $a \equiv 1 \pmod{\{3, 17, 2^{16}+1\}}$, include $2^{16}+1$ in the factorization of Π, and force $k_{(0)}$ so that $k_{(0)} \not\equiv 1 \pmod{\{3, 17, 2^{16}+1\}}$. A possible candidate for a is the prime $R = 2^{64} - 2^{32} + 1$, provided that $\gcd(\Pi, R) = 1$. The condition on $k_{(0)}$ can be achieved by Chinese remaindering. More precisely, we need a value $K_{(0)}$ constructed from random seed σ such that $K_{(0)} \not\equiv 0, 1 \pmod{\{3, 17, 2^{16}+1\}}$. Given σ, we first construct two random integers in the respective ranges $[2, 2^4]$ and $[2, 2^{16}]$, say $\kappa_1 = \mathrm{MGF}_{2'}(\sigma)$ and $\kappa_2 = \mathrm{MGF}_{2''}(\sigma)$. Next, by Chinese remaindering (see Eq. (2)) modulo $e_1 := 17$ and $e_2 := 2^{16}+1$, we compute $\kappa_{1,2} = \kappa_2 + e_2[i_{1,2}(\kappa_1 - \kappa_2) \bmod e_1]$ where $i_{1,2} = 1/e_2 \bmod e_1$. Letting $e_0 := 3$, we compute $\kappa_{0,1,2} = \kappa_{1,2} + e_1e_2[i_{12,0}(2 - \kappa_{1,2}) \bmod e_0]$ where $i_{12,0} = 1/(e_1e_2) \bmod e_0$. (Observe that $\kappa_{0,1,2} \not\equiv 0, 1 \pmod{\{3, 17, 2^{16}+1\}}$.) From σ we construct a random integer in the range $[0, \pi)$ with $\pi = \Pi/(e_0e_1e_2)$, say $\kappa_\pi = \mathrm{MGF}_{2'''}(\sigma)$, and by Chinese remaindering modulo π and $e_0e_1e_2$, we finally define $K_{(0)}$ as

$$K_{(0)} = \kappa_{0,1,2} + e_0e_1e_2[i_{012,\pi}(\kappa_\pi - \kappa_{0,1,2}) \bmod \pi] \tag{8}$$

where $i_{012,\pi} = 1/(e_0e_1e_2) \bmod \pi$.

Consequently, we obtain an invertible element modulo Π, $k_{(0)}$, satisfying the condition of Eq. (7) for $e \in \{3, 17, 2^{16}+1\}$ as

$$k_{(0)} = \left[K_{(0)} + b^{\mathrm{MGF}_3(\sigma)}(K_{(0)}^{\lambda(\Pi)} - 1)\right] \pmod{\Pi} \ .$$

(This has to be compared to Eq. (6).)

It is worthwhile noticing here that for $e \in \{3, 17, 2^{16}+1\}$ (dividing $\varPi$), we have $k_{(0)} \equiv K_{(0)} \pmod{e}$ since $K_{(0)} \not\equiv 0 \pmod{e}$ by construction.

3.2.4 Off-line/On-line generation The system assumes the knowledge of a lower bound B_0 on the bit-length of the RSA primes being generated and a set $\mathcal{E}$ of public exponents likely be used in the applications. This determines the choice of parameter $\varPi$ (and so of $\lambda(\varPi)$). We also need two invertible elements modulo $\varPi$, a and b. Finally, we assume that we have at disposal a hash function H and a family of mask generating functions MGF_i (one may for example define $\mathrm{MGF}_i(x)$ as $\mathrm{MGF}(x\|i)$).

For concreteness, suppose that $B_0 = 256$ and $\mathcal{E} = \{e_0, e_1, e_2\}$ with $e_0 = 3$, $e_1 = 17$ and $e_2 = 2^{16}+1$. Then we can take $\varPi = (2^{16}+1) \cdot \prod_{i=1}^{41} p_i = (2^{16}+1) \cdot 2 \cdot 3 \cdots 179 < 2^{256}$. We choose $a = b = 2^{64} - 2^{32} + 1 := R$. Note that e_i divides $\varPi$ (for $i \in \{0,1,2\}$) and that $\gcd(R, \varPi) = 1$.

There are three algorithms. The first algorithm constructs constrained units; the second, off-line algorithm constructs values for the third, on-line algorithm generating RSA keys.

With the above system parameters, a possible implementation of the first algorithm is given below. The input is a string σ given by the calling algorithm.

```
Input:  parameter $\sigma$
Output: $k \in \mathbb{Z}_{\varPi}{}^*$

1. Compute $\kappa_1 \leftarrow \mathrm{MGF}_{2'}(\sigma) \pmod{(e_1 - 3)} + 2$
2. Compute $\kappa_2 \leftarrow \mathrm{MGF}_{2''}(\sigma) \pmod{(e_2 - 3)} + 2$
3. Compute $\kappa_\pi \leftarrow \mathrm{MGF}_{2'''}(\sigma) \pmod{\varPi/(e_0 e_1 e_2)}$
4. Compute $K_{(0)} \leftarrow \mathrm{CRT}(2, \kappa_1, \kappa_2, \kappa_\pi)$ as in Eq. (8)
5. Compute $t \leftarrow \mathrm{MGF}_3(\sigma) \pmod{\mathrm{ord}_{\varPi}(R)}$
6. Output $K_{(0)} + \left[K_{(0)} + R^t(K_{(0)}^{\lambda(\varPi)} - 1)\right] \pmod{\varPi}$
```

Fig. 2. Generation of $k \in \mathbb{Z}_{\varPi}{}^*$.

For a given bit-length ℓ_0, the next off-line algorithm (Fig. 3) produces c values f_z (for $z \in \{1, \ldots, c\}$) which will be used in the on-line generation ℓ_0-bit RSA primes valid with probability 1 when public exponent e lies in $\mathcal{E}$. Parameter σ_0 is a random string, proper to each card, and stored in EEPROM-like memory. A typical length for σ_0 is 64 bits.

An RSA application takes on input an RSA key-length ℓ and a public exponent e. If ℓ can be written as the sum of two available ℓ_0, then the next on-line algorithm (Fig. 4) is called for each of the two bit-lengths, ℓ_0, forming the RSA modulus $N = pq$, together with public exponent e. If no such decomposition

```
Input:  parameters $\ell_0, \sigma_0, c$
Output: $f_z$ for $z \in \{1, \ldots, c\}$

1. Compute $v \leftarrow \left\lceil \frac{\lfloor 2^{\ell_0 - 1/2} \rfloor}{\Pi} \right\rceil$ and $w \leftarrow \left\lfloor \frac{2^{\ell_0}}{\Pi} \right\rfloor$
2. Set $z = 1$
3. Define $\sigma \leftarrow H(\sigma_0, \ell_0, z)$
4. Compute $j \leftarrow \mathrm{MGF}_1(\sigma) \pmod{(w - v)} + v$ and set $l \leftarrow j\Pi$
5. Using the algorithm of Fig. 2, compute $k \in {\mathbb{Z}_\Pi}^*$
6. Set $f_z \leftarrow 0$ and $q \leftarrow k + l$
7. If ($q$ is not prime) then
   (a) Set $f_z \leftarrow f_z + 1$ and $k \leftarrow R\,k \pmod{\Pi}$
   (b) Go to Step 4
8. Output $f_z$
9. If ($z < c$) then set $z \leftarrow z + 1$ and go to Step 3
```

Fig. 3. Off-line algorithm.

```
Input:  parameters $\ell_0$, $e$ and
        the list $\{f_z\}_{1 \le z \le c}$, $z$
Output: a prime $q \in \big[\lceil 2^{\ell_0 - 1/2} \rceil, 2^{\ell_0} - 1\big]$ s.t.
        $\gcd(q - 1, e) = 1$

1. Compute $v \leftarrow \left\lceil \frac{\lfloor 2^{\ell_0 - 1/2} \rfloor}{\Pi} \right\rceil$ and $w \leftarrow \left\lfloor \frac{2^{\ell_0}}{\Pi} \right\rfloor$
2. Define $\sigma \leftarrow H(\sigma_0, \ell_0, z)$
3. Compute $j \leftarrow \mathrm{MGF}_1(\sigma) \pmod{(w - v)} + v$ and set $l \leftarrow j\Pi$
4. Using the algorithm of Fig. 2, compute $k \in {\mathbb{Z}_\Pi}^*$
5. Compute $k \leftarrow R^{f_z} k \pmod{\Pi}$
6. Set $q \leftarrow k + l$
7. If ($\gcd(e, q - 1) \neq 1$) then
   (a) Set $z \leftarrow z + 1$
   (b) If ($z > c$) output ‘‘Error’’; otherwise go to Step 2
8. Output $q$
```

Fig. 4. On-line algorithm.

exists then the off-line algorithm must be called to generate valid values or an error message must be output.

For security reasons, we insist that a prime can only be used *once* for a given application. When it is used, it must be marked as such (e.g., by removing the corresponding entry for z, f_z).

If the so-obtained prime, say q, does not satisfy the mandatory condition $\gcd(q-1, e) = 1$ then another value of z is tested. If there are no longer available values for z then the off-line algorithm must be called to generate valid values or an error message must be output.

4 Concluding Remarks

This paper presented a mixed off-line/on-line methodology for the generation of RSA keys, leading to on-board performances several orders of magnitude faster than state-of-the-art techniques. Two concrete realizations were proposed. The first solution has a faster on-line phase (at the expense of a slower off-line phase) and the second solution features a faster off-line phase. Further, these new fast off-line/on-line key generation algorithms enable "pay as you go" e-payment functions by reducing the cost of their infrastructure (cost of certificates and keys in card memory) while keeping the same security level than the one of classical key generation processes.

References

1. Bellare, M., and Rogaway, P. Optimal asymmetric encryption. In *Advances in Cryptology – EUROCRYPT '94* (1995), LNCS 950, Springer-Verlag, pp. 92–111.
2. Bellare, M., and Rogaway, P. The exact security of digital signatures - How to sign with RSA and Rabin. In *Advances in Cryptology – EUROCRYPT '96* (1996), LNCS 1070, Springer-Verlag, pp. 399–416.
3. Joye, M., and Paillier, P. Constructive methods for the generation of prime numbers. In *2nd Open NESSIE Workshop* (Egham, UK, Sept. 12–13, 2001).
4. Joye, M., Paillier, P., and Vaudenay, S. Efficient generation of prime numbers. In *Cryptographic Hardware and Embedded Systems – CHES 2000* (2000), LNCS 1965, Springer-Verlag, pp. 340–354.
5. Menezes, A. J., van Oorschot, P. C., and Vanstone, S. A. *Handbook of applied cryptography.* CRC Press, 1997.
6. PKIX. Public Key Infrastructure (X.509) series. Available at URL http://www.ietf.org/html.charters/pkix-charter.html.
7. Quisquater, J.-J., and Couvreur, C. Fast decipherment algorithm for RSA public-key cryptosystem. *Electronics Letters* **18** (1982), 905–907.
8. Rivest, R. L., Shamir, A., and Adleman, L. M. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM* **21**, 2 (1978), 120–126.