Fast Generation of Prime Numbers on Portable Devices: An Update

Marc Joye[1,⋆] and Pascal Paillier[2]

[1] Thomson R&D France
Technology Group, Corporate Research, Security Laboratory
1 avenue Belle Fontaine, 35576 Cesson-Sévigné, France
marc.joye@thomson.net
[2] Gemalto, Security Labs
34 rue Guynemer, 92447 Issy-les-Moulineaux Cedex, France
pascal.paillier@gemalto.com

Abstract. The generation of prime numbers underlies the use of most public-key cryptosystems, essentially as a primitive needed for the creation of RSA key pairs. Surprisingly enough, despite decades of intense mathematical studies on primality testing and an observed progressive intensification of cryptography, prime number generation algorithms remain scarcely investigated and most real-life implementations are of dramatically poor performance.
We show simple techniques that substantially improve all algorithms previously suggested or extend their capabilities. We derive fast implementations on appropriately equipped portable devices like smart-cards embedding a cryptographic coprocessor. This allows onboard generation of RSA keys featuring a very attractive (average) processing time.
Our motivation here is to help transferring this task from terminals where this operation usually took place so far, to portable devices themselves in near future for more confidence, security, and compliance with network-scaled distributed protocols such as electronic cash or mobile commerce.

Keywords: Public-key cryptography, RSA, primality testing, prime number generation, embedded software, efficient implementations, cryptoprocessors, smart cards, PDAs.

1 Introduction

Undoubtedly, the lack of *efficient* prime number generators severely restricts the development of public-key cryptography in embedded environments. Several algorithms that generate prime numbers do exist, some of them being well-known and popular [5, 6, 8, 17], but most of them are hardly adapted to the computational context of portable devices like smart cards or PDAs, where memory capabilities and processing power are somewhat limited. A noticeable exception is found in a recent heuristic algorithm by Joye, Paillier and Vaudenay [13].

⋆ This work was done while the author was with Gemalto (formerly Gemplus).

In this paper, we improve their algorithm in multiple directions. First, we give a more general description with extended parameter choices that fit any (crypto-)processor architecture. Second, we present new techniques that speed up the entire process and reduce the standard statistical deviation, especially in the generation of so-called units. Third, we consider the issue of length extendability, that is, algorithmic solutions for obtaining primes of arbitrary and dynamically chosen bitsize.

The way prime numbers are selected during (e.g., RSA) key generation is critical towards the security of generated key pairs. Therefore we investigate the mathematical properties fulfilled by our improved algorithms. Using an analogue of Gallagher's empiric law on the distribution of primes in arithmetic progressions [10, 11], we accurately evaluate the output entropy of our generators. We also analyze the probability that two outputs are identical, i.e., that one gets the same prime number when running the generation twice with randomly selected independent inputs. It is shown that the output entropy is nearly optimal (the entropy loss is < 0.61 bits compared to uniform distribution) and that collisions remain extremely unlikely.

The prime number generation algorithms we consider here find their main application in the generation of RSA keys on embedded platforms. This context of use implies the additional condition on a prime q being generated, that $q - 1$ be coprime to a prescribed public RSA exponent e. We show how our algorithm may automatically fulfill this latter condition at negligible cost, at least for small or smooth values of e. Further, as an additional application of our techniques, we show how to efficiently generate a random safe (resp. quasi-safe) prime. This answers a problem left open in [13].

The rest of the paper is organized as follows. In the next section, we present our improved prime generation algorithms. We then provide a security analysis in Section 3. In Section 4, we apply our techniques to the generation of RSA keys and of safe primes. Finally, we conclude in Section 5.

2 Efficient Generation of Prime Numbers

This section describes efficient (trial-division free as opposed to [3, 6, 8, 17]) algorithms for producing a prime q uniformly distributed in some given interval $[q_{\min}, q_{\max}]$ or a sub-interval thereof; $q_{\min}$ and $q_{\max}$ being two arbitrarily chosen integers and $q_{\min} < q_{\max}$. Our proposal actually consists of a pair of algorithms: the prime generation algorithm itself and an algorithm for generating invertible elements, also called *units* [13]. We assume that a random number generator is available, and that some fast (pseudo-)primality (resp. compositeness [2, 4, 14, 20, 22, 25, 19]) testing function T is provided as well.

Parameter setup. Let $0 < \varepsilon \leqslant 1$ denote a quality parameter (a typical value for ε is 10^{-3}). Let also ϕ denote Euler's totient function. Our setup phase requires to choose a product of primes, $\Pi = \prod_i p_i$, such that there exist integers t, v, w satisfying

(P1) $1-\varepsilon < \dfrac{w\Pi - 1}{q_{\max} - q_{\min}} \leqslant 1$;

(P2) $v\Pi + t \geqslant q_{\min}$;

(P3) $(v+w)\Pi + t - 1 \leqslant q_{\max}$;

(P4) the ratio $\phi(\Pi)/\Pi$ is as small as possible .

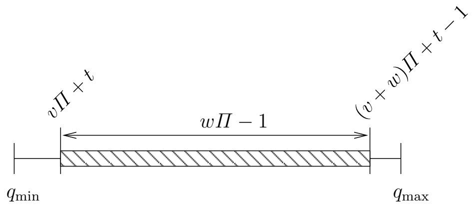

Fig. 1. ε-approximated output domain.

The primes output by our algorithm lie, in fact, in the sub-interval $[v\Pi + t, (v+w)\Pi + t - 1] \subseteq [q_{\min}, q_{\max}]$ as illustrated on Fig. 1. The error in the approximation is captured by the value of ε meaning that a smaller value for ε gives better results (cf. Property (P1)). The minimality of the ratio $\phi(\Pi)/\Pi$ in Property (P4) ensures that Π contains a maximum number of distinct primes and that these primes are as small as possible. Given any tuple $(q_{\min}, q_{\max}, \varepsilon)$, computing the tuple (Π, v, w, t) that best matches Properties (P1)–(P4) is experimentally easy.

Prime number generation. We now proceed to describe our prime number generation algorithm in its most generic version, as depicted on Fig. 2.

Parameters: t, v, w and $a \in (\mathbb{Z}/m\mathbb{Z})^* \setminus \{1\}$
Output: a random prime $q \in [q_{\min}, q_{\max}]$

---

1. Compute $l \leftarrow v\Pi$ and $m \leftarrow w\Pi$
2. Randomly choose $k \in (\mathbb{Z}/m\mathbb{Z})^*$
3. Set $q \leftarrow [(k-t) \bmod m] + t + l$
4. If $(\mathsf{T}(q) = \texttt{false})$ then
   (a) Set $k \leftarrow a \cdot k \pmod{m}$
   (b) Go to Step 3
5. Output q

Fig. 2. Generic prime generation algorithm for $q \in [q_{\min}, q_{\max}]$.

The first step requires the random selection of an integer $k \in (\mathbb{Z}/m\mathbb{Z})^*$ (see Section 2.2) where $m = w\Pi$ is a smooth integer. At this stage, it is worthwhile noticing that since $a \in (\mathbb{Z}/m\mathbb{Z})^*$, k remains coprime to m and also to Π throughout the algorithm —remember that Π contains a large number of prime factors by Property (P4). This, in turn, implies that q is coprime to Π as $q \equiv [(k-t) \bmod m] + t + l \equiv k \pmod{\Pi}$ and $k \in (\mathbb{Z}/\Pi\mathbb{Z})^*$. Hence, this technique ensures *built-in* coprimality of our prime candidate q with a large set of small prime numbers. Consequently, the probability under which q is prime at `Step 3` is in fact quite high. When q is found to be composite, a new candidate is derived by "recycling" q in a way that preserves its coprimality to Π.

2.1 An implementation example

The previous algorithm is actually very general and can be adapted in numerous ways, depending on hardware capabilities of the targeted processor architecture. Public-key crypto-processors generally allow super-fast (modular) additions, subtractions and multiplications over large integers, and this renders other types of computations comparatively prohibitive, unless specific hardware is integrated to support these. We now give a possible implementation to illustrate this, in which we attempt to increase our algorithm's performance to its uppermost level while running on a general-purpose crypto-processor. Other choices of parameters may lead to better results on specific platforms.

A first improvement is to choose $w = 1$ and to let the value of t varying as a random multiple of Π, say $t = b\Pi$ for some integer b, instead of fixing it. This allows to compute modulo Π, resulting in faster arithmetic. Also, the constant a may be chosen such that performing a multiplication by a modulo m turns out to be a somewhat trivial operation. In the end, the best possible choice is $a = 2$, because multiplying by 2 then amounts to a single bit shift or addition, possibly followed by a subtraction. Unfortunately, 2 must belong to $(\mathbb{Z}/m\mathbb{Z})^*$ and owing to Property (P4), 2 is a factor of Π, a contradiction. A simple trick here consists in choosing m odd (so that $2 \in (\mathbb{Z}/m\mathbb{Z})^*$) and in slightly modifying the above framework in order to ensure that a prime candidate q is always odd. We require $\Pi = \prod_i p_i$ (with $p_i \neq 2$) and integers $b_{\min}$, $b_{\max}$, v satisfying:

(P1) $1 - \varepsilon < \dfrac{(b_{\max} - b_{\min} + 1)\Pi - 1}{q_{\max} - q_{\min}} \leqslant 1\,;$

(P2) $v\Pi + b_{\min}\Pi \geqslant q_{\min}\,;$

(P3) $(v+1)\Pi + b_{\max}\Pi - 1 \leqslant q_{\max}\,;$

(P4) the ratio $\phi(\Pi)/\Pi$ is as small as possible .

Putting it all together, we obtain the algorithm shown on Fig. 3.[3] Note that if $k{+}t{+}l$ is even then $\Pi - k{+}t{+}l$ is odd since $\Pi - k{+}t{+}l \equiv \Pi + (k{+}t{+}l) \equiv \Pi \equiv 1 \pmod 2$. Hence, as before, any candidate q belonging to our search sequence is

[3] Stricly speaking, the algorithm of Fig. 3 is a particular case of the generic algorithm of Fig. 2 only if, at `Step 6(b)`, we go to `Step 3` (instead of `Step 4`).

coprime to 2Π: we get $\gcd(q, 2) = 1$ as q is odd. Also, $\gcd(q, \Pi) = 1$ as $q \equiv \pm k \pmod{\Pi}$ and $\pm k \in (\mathbb{Z}/\Pi\mathbb{Z})^*$.

```
Parameters: Π odd, b_min, b_max, v
Output:     a random prime q ∈ [q_min, q_max]

1. Compute l ← vΠ
2. Randomly choose k ∈ (Z/ΠZ)*
3. Randomly choose b ∈ {b_min, ..., b_max} and set t ← bΠ
4. Set q ← k + t + l
5. If (q even) then q ← Π − k + t + l
6. If (T(q) = false) then
   (a) Set k ← 2k (mod Π)
   (b) Go to Step 4
7. Output q
```

Fig. 3. Faster prime generation algorithm.

2.2 Generation of units

All prime generation algorithms presented in this paper require the random selection of some element $k \in (\mathbb{Z}/m\mathbb{Z})^*$ in the spirit of [13]. This section provides an algorithm that efficiently produces such an element with uniform output distribution. We base our design on the next two propositions, making use of Carmichael's function λ.

Proposition 1 (Carmichael [7]). *Let $m > 1$ and let k be any integer modulo m. Then $k \in (\mathbb{Z}/m\mathbb{Z})^*$ if and only if $k^{\lambda(m)} \equiv 1 \pmod{m}$.* □

Proposition 2. *Let k, r be integers modulo m and assume $\gcd(r, k, m) = 1$. Then*

$$[k + r(1 - k^{\lambda(m)}) \bmod m] \in (\mathbb{Z}/m\mathbb{Z})^* \ .$$

Proof. Let $\prod_i {p_i}^{\delta_i}$ denote the prime factorization of m. Define $\omega(k, r) := [k + r(1 - k^{\lambda(m)}) \bmod m] \in \mathbb{Z}/m\mathbb{Z}$. Let p_i be a prime factor of m. Suppose that $p_i \mid k$ then $\omega(k, r) \equiv r \not\equiv 0 \pmod{p_i}$ since $\gcd(r, p_i)$ divides $\gcd(r, \gcd(k, m)) = \gcd(r, k, m) = 1$. Suppose now that $p_i \nmid k$ then $k^{\lambda(m)} \equiv 1 \pmod{p_i}$ and so $\omega(k, r) \equiv k \not\equiv 0 \pmod{p_i}$. Therefore for all primes $p_i \mid m$, we have $\omega(k, r) \not\equiv 0 \pmod{p_i}$ and thus $\omega(k, r) \not\equiv 0 \pmod{{p_i}^{\delta_i}}$, which, invoking Chinese remaindering, concludes the proof. □

We benefit from these facts by devising the unit generation algorithm shown on Fig. 4.

```
Parameters: m and λ(m)
Output:     a random unit k ∈ (ℤ/mℤ)*

1. Randomly choose k ∈ [1, m[
2. Set U ← (1 − k^λ(m)) mod m
3. If (U ≠ 0) then
   (a) Choose a random r ∈ [1, m[
   (b) Set k ← k + rU (mod m)
   (c) Go to Step 2
4. Output k
```

Fig. 4. Our unit generation algorithm.

This algorithm is self-correcting in the following sense: as soon as k is relatively prime to some factor of m, it remains coprime to this factor after the updating step $k \leftarrow k+rU$. This is due to Proposition 2. What happens in simple words is that, viewing k as the vector of its residues $k \bmod {p_i}^{\delta_i}$ for all ${p_i}^{\delta_i} \mid m$ (i.e., the RNS representation of k based on m, see [9]), non-invertible coordinates of k are continuously re-randomized until invertibility is reached for all of them. This ensures that the output distribution is strictly uniform provided that the random number generator is uniformly distributed over $[1, m[$.

2.3 Efficiency

A complexity analysis for generating an n_0-bit prime q is easily driven from the work of [13]. The expected number of calls to T, i.e., the number of primality or compositeness tests required on average, heuristically amounts to

$$n_0 \cdot \ln 2 \cdot \frac{\phi(\Pi)}{\Pi} = \mathrm{O}\left(\frac{n_0}{\ln n_0}\right) \quad .$$

Naturally the exact, concrete efficiency of our implementation also depends on hardware-related features. In any case, in practice, a spectacular execution speed-up[4] is generally observed in comparison with usual, incremental and trial-division-based prime number generators. It can be shown that the unit generation requires about 2.15 modular exponentiations $x \mapsto x^{\lambda(m)} \bmod m$ where the bitsize of $\lambda(m)$ is much smaller than the bitsize of m, and experimentally never exceeds $|m|/3$. For instance, one has $|\lambda(m)| \simeq 160$ when $|m| = 512$. Note also that all computations fall into the range of operations easily and efficiently performed by any crypto-processor.

We note that many previous works such as [24, 16, 15] make use of trial-divisions up to a large bound to decrease the number of calls to T. This common

[4] which usually amounts to one order of magnitude.

technique is hardly adapted to cryptoprocessors where each and every modular reduction may impose a prior, time-prohibitive modulus-dependent initialization. Experience shows that practical smart-card implementations are found to impressively benefit from our above algorithm in comparison to these.

2.4 Length extendability

So far, our implementation parameters are Π, a, the tuple (v, w, t) and $\lambda(m)$ with $m = w\Pi$. These values are chosen once and for all and heavily depend on $q_{\min} = \lceil 2^{n_0-1/2} \rceil$ and $q_{\max} = 2^{n_0}$, if n_0 denotes the bitsize of prime numbers being generated. Now, the feature we desire here (and this is motivated by code size limitations embedded platforms usually have to work with), consists in the ability to use the parameters sized for n_0 to generate primes numbers of bitsize $n \neq n_0$. A performance loss is acceptable compared to the situation when parameters are generated for both lengths.

We propose an implementation solving that problem for any $n \geqslant n_0$, provided that a was chosen odd and that arithmetic computations can still be carried out over n-bit numbers on the processor taken into consideration. It is an extended version of the algorithm depicted on Fig. 2. We exploit the somewhat obvious, following facts:

1. Letting $q_{\max}(x) = 2^x$ and $q_{\min}(x) = \lceil 2^{x-1/2} \rceil$, we have of course $q_{\max}(n) = q_{\min}(n_0)2^{n-n_0}$ and $q_{\min}(n) \approx q_{\min}(n_0)2^{n-n_0}$;
2. Given $\Pi(n_0)$ chosen as per Section 2, we take
$$\begin{cases} \Pi(n) = \Pi(n_0) \\ v(n) = v(n_0)2^{n-n_0} \\ w(n) = w(n_0)2^{n-n_0} \\ t(n) = t(n_0)2^{n-n_0} \end{cases},$$
hence $l(n) = l(n_0)2^{n-n_0}$ and $m(n) = m(n_0)2^{n-n_0}$;
3. $a(n) = a(n_0)$, hence $a(n) \in (\mathbb{Z}/m(n)\mathbb{Z})^*$ since $a(n_0)$ is taken odd;
4. Given $\lambda(n_0) = \lambda(m(n_0))$, it is easy to see that denoting $\lambda(n) = \lambda(n_0)2^{n-n_0}$, we have again $\lambda(n) = \lambda(m(n))$, or at least $\lambda(n) \propto \lambda(m(n))$ which is a sufficient condition for the unit generation algorithm to be effective.

These transformations happen to preserve Properties (P1), (P2) and (P3) we required earlier, with $\varepsilon(n) = \varepsilon(n_0)$. It is easy to see that all parameters for some bitsize n may, as a direct consequence, be replaced by the respective parameters computed for n_0 multiplied by 2^{n-n_0}, except for $\Pi(n) = \Pi(n_0)$. By performing this replacement, we just accept to live with sub-optimized performances because the ratio $\phi(\Pi(n))/\Pi(n)$ will not be chosen minimal. Still, our algorithm will output n-bit primes in a correct manner, for any dynamic choice of $n \geqslant n_0$, with a 1-bit granularity.

Parameters:	$l(n_0) = v(n_0)\Pi(n_0)$, $m(n_0) = w(n_0)\Pi(n_0)$, $t(n_0)$, $a(n_0) \in (\mathbb{Z}/m(n_0)\mathbb{Z})^* \setminus \{1\}$, n_0
Input:	bitsize $n \geqslant n_0$
Output:	a random prime $q \in [q_{\min}(n), q_{\max}(n)]$

1. Set $m \leftarrow m(n_0)2^{n-n_0}$, $t \leftarrow t(n_0)2^{n-n_0}$ and $l \leftarrow l(n_0)2^{n-n_0}$
2. Randomly choose $k \in (\mathbb{Z}/m\mathbb{Z})^*$
3. Set $q \leftarrow [(k-t) \bmod m] + t + l$
4. If $(\mathsf{T}(q) = \texttt{false})$ then
   (a) Set $k \leftarrow a(n_0)\,k \pmod m$
   (b) Go to Step 3
5. Output q

Fig. 5. Our scalable prime generation algorithm.

Our extended algorithm is depicted on Fig. 5. In `Step 1`, the random unit generation is carried out with parameters $m(n_0)2^{n-n_0}$ and $\lambda(n_0)2^{n-n_0}$ instead of $m(n_0)$ and $\lambda(n_0)$. This does not affect the algorithm whatsoever. Another observation is that the order of $a(n)$ modulo $m(n)$ is necessarily larger than (or equal to) the order of $a(n_0)$ modulo $m(n_0)$. It is therefore large enough for all our choices of n provided that $a(n_0)$ was correctly chosen in the first place.

3 Security Analysis

We outline in this section a mathematical analysis of our generic prime generation algorithm (Fig. 2). The results are easily transposable to the other prime generation algorithms presented in this paper. We answer the following critical questions:

Question 1. Are output primes well distributed? How much entropy is there in the output distribution?

Question 2. What is the probability that the same prime is output for two independently selected input values?

3.1 Output entropy

We accurately evaluate the entropy H of the output distribution which, following Brandt and Damgård's methodology [5], is considered as a quality measure of a prime number generator.

Theorem 1. *Let $H_{\max}$ be the maximal possible value of H. Then, under Hardy and Littlewoods' prime r-tuple conjecture* [11] *and Gallagher's heuristic* [10], *we* *have for any* $n \geqslant 256$*,*

$$H_{\max} - H < \frac{1-\gamma}{\ln 2} = 0.609949$$

where γ *is the Euler-Mascheroni constant* [22]. □

Theorem 1 shows that the entropy loss with respect to a perfectly uniform generator is less that 0.61 bit for any prime bitlength. Due to lack of space, we omit the proof here and refer the reader to the extended version of this work for more detail [12].

Table 1 represents the concrete values for H, $H_{\max}$ and $\rho = (H_{\max} - H)/H_{\max}$ for various bitlengths n. We see that the output entropy of our generator is

Table 1. Output entropy H as a function of n.

n	256	384	512	640	768	896	1024
$H_{\max}$	246.767	374.179	501.762	629.439	757.176	884.953	1012.76
H	246.194	373.596	501.173	628.847	756.581	884.356	1012.16
$H_{\max} - H$	0.572795	0.583093	0.588773	0.592377	0.594834	0.59669	0.598092
ρ (%)	0.23212	0.155833	0.117341	0.094111	0.078559	0.067426	0.0590557

similar to the one of random search, in which one sets candidate q to successive random numbers until q is prime. Our figures show that

- asymptotically, the output entropy gets arbitrarily close to its maximal possible value, and
- the gap is already negligibly small for concrete bitsizes of practical interest $256 \leqslant n \leqslant 1024$.

3.2 Collision probability

Theorem 2. *We denote by* ν *the probability that the same prime number is output twice for two uniformly and independently distributed random inputs. Then*

$$\nu < \frac{\ln 2}{1 - \frac{1}{\sqrt{2}}} \cdot n \cdot 2^{-n+1} \quad .$$

□

Again, we refer to the extended version of this paper [12] for a detailed proof of Theorem 2 and related insights. Table 2 displays ν for common values of n.

Table 2. Collision probability.

n	128	256	384	512	1024
$\nu \leqslant$	$1.91 \cdot 10^{-75}$	$3.30 \cdot 10^{-152}$	$4.28 \cdot 10^{-229}$	$4.93 \cdot 10^{-306}$	$5.49 \cdot 10^{-614}$

As a result, from Theorems 1 and 2, we conclude that our prime generation algorithms are *provably reliable.*

4 Concrete Cryptographic Applications

We apply the prime number generators above to the concrete generation of RSA primes, in which the public exponent e is fixed and set to a standard value. We also consider the case of safe primes as they underly many variants of RSA and other popular cryptosystems.

4.1 Generating RSA Primes

This section deals with the generation of an RSA prime q. Let $e = \prod_i {e_i}^{\nu_i}$ denote the prime factorization of a given public exponent e. Because the RSA primitive (see Appendix A) induces a permutation (i.e., $\gcd(e, \lambda(N)) = 1$), it turns out that q must be such that $\gcd(e_i, q-1) = 1$ for each prime e_i dividing e.

First, let us assume that $e_i \mid \Pi$ for all i. This happens in the most popular scenario where e is some small prime (like 3 or 17) or when e is chosen smooth. Let α be an integer such that

$$\gcd(\alpha, m) = 1 \text{ and } \operatorname{order}(\alpha \bmod e_i, e_i) = e_i - 1 \text{ for each } e_i \mid e \ . \tag{1}$$

In practice, the choice of a value for α may be done easily using Chinese remaindering. Note that for such an α, we get that $\operatorname{order}(\alpha, e_i)$ is simultaneously even for all prime factors $\{e_i\}_i$. We define $e^+ = \gcd(e, \Pi) = \prod_i e_i$ and denote by k_0 the initial value for k that the unit generation algorithm of Fig. 4 gets by invoking the random number generator in `Step 1`. It is easily seen that if we force

$$k_0 \equiv \alpha \pmod{e^+} \ , \tag{2}$$

then the unit k eventually output by the algorithm will also verify that $k \equiv \alpha \pmod{e^+}$. This is due to the algorithm's self-correctness. We then adapt the generic prime generation algorithm by choosing $a = \alpha^2$. By doing so, every candidate q generated by the sequence will satisfy

$$q \equiv \alpha^{2j+1} \pmod{e^+} \ ,$$

for some integer j, because $e^+ \mid \Pi$. Hence we can never have $q \equiv 1 \pmod{e_i}$ since α is of even order modulo e_i and q is an odd power of α. Consequently, $q \not\equiv 1 \pmod{e_i}$ for all i, which implies $\gcd(q-1, e) = 1$.

So our technique works when $e_i \mid \Pi$ for all i, that is, when e has only small prime factors. To deal with cases when $e_i \nmid \Pi$ for some $e_i \mid e$, we face the following options:

- either e is a prime number itself (like Fermat's fourth prime $2^{16} + 1$) and we add the verification step

  $$q - 1 \stackrel{?}{\not\equiv} 0 \pmod{e}$$

  before or after the primality test T is applied; or

– e is not prime but its factorization is known. We already know that $q \not\equiv 1 \pmod{e_i}$ when $e_i \mid \Pi$, so we have to ensure that the same holds when $e_i \nmid \Pi$. To do this, we simply check that $q - 1 \not\equiv 0 \pmod{e_i}$ for all prime factors $e_i \nmid \Pi$, or equivalently (but preferably) invoke Proposition 1 and make sure that

$$(q-1)^{\lambda(e^-)} \equiv 1 \pmod{e^-} ,$$

where $e^- = \prod_i e_i$ for all $e_i \nmid \Pi$.

In both cases, unfortunately, adding at least one additional test to the implementation cannot be avoided.

Finally, forcing $k_0 \equiv \alpha \pmod{e^+}$ in Eq. (2) is easily done by picking a random number r and setting $k_0 = \alpha + e\,r \pmod{m}$.

4.2 Generating safe and quasi-safe primes

We now show how to apply our generic techniques to the specific case of generating safe primes or quasi-safe primes. A *safe prime* is a prime q such that $(q-1)/2$ is also a prime. More generally, a *d-quasi-safe prime* is a prime q such that $(q-1)/2^d$ is prime.

All the point here resides in the way the search sequence is carried out. It should ideally verify that each and every candidate q be such that both q and $(q-1)/2$ are always coprime to Π. It is somewhat easy to guarantee that for q by ensuring (like in previous sections) that

$$q \equiv ak \pmod{\Pi}$$

for some $a, k \in (\mathbb{Z}/m\mathbb{Z})^*$. However, the later constraint on $(q-1)/2$ is a bit more delicate. Our need here is to ensure that for each prime divisor p_i of Π, $p_i \neq 2$,

$$q \not\equiv 1 \pmod{p_i} .$$

Our idea is to make sure that $q \bmod p_i$ just cannot be an element of $\mathrm{QR}(p_i)$, the subgroup of quadratic residues modulo p_i. Doing so, we ensure that $q \not\equiv 1 \pmod{p_i}$. We proceed in the following way. First, the constant a is chosen in $\mathrm{QR}(m)$. Next, we choose once for all a parameter $u \in (\mathbb{Z}/m\mathbb{Z})^*$ such that

$$\forall(\text{odd})\ p_i \mid \Pi : \quad u \notin \mathrm{QR}(p_i) . \tag{3}$$

From there on, the initial unit k (to avoid confusion, we denote it by k_0) is chosen as $k_0 = u\,\chi^2 \bmod m$ for some random $\chi \in (\mathbb{Z}/m\mathbb{Z})^*$. Then, as before, we have at iteration j

$$q = [(a^j\,k_0 - t) \mod m] + t + l .$$

It is now easy to see that for each and every odd prime $p_i \mid \Pi$, $q \equiv a^j\,u\chi^2 \pmod{p_i}$ has a Legendre symbol different from 1, and consequently $q - 1$ cannot be 0 modulo p_i, i.e., $q - 1$ is coprime to Π.

When $2^\tau \mid m$ for some $\tau \geqslant 2$, we have to make sure, in addition to the above, that $q \equiv 3 \pmod 4$ meaning that the last two bits in the binary representation of q are forced to $\ldots 11_2$, thereby ensuring that $(q-1)/2$ is an odd number and consequently that $(q-1)/2 \in (\mathbb{Z}/\Pi\mathbb{Z})^*$. This is done by forcing $k \equiv 3 \pmod 4$ and $a \equiv 1 \pmod 4$.

The resulting algorithm is described on Fig. 6.

`Parameters:`	$l = v\Pi$, $m = w\Pi$, $m' = m/2^\tau$,
	t, $a \in \mathrm{QR}(m)$ and u as above
`Output:`	a random prime $q \in [q_{\min}, q_{\max}]$ with $(q-1)/2$ prime

1. Randomly choose $\chi \in (\mathbb{Z}/m\mathbb{Z})^*$
2. Set $k \leftarrow 4u\chi^2 + 3m' \mod m$
3. Set $q \leftarrow [(k-t) \bmod m] + t + l$
4. If $(\mathsf{T}(q) = \texttt{false}$ or $\mathsf{T}((q-1)/2) = \texttt{false})$ then
   (a) Set $k \leftarrow ak \pmod m$
   (b) Go to Step 3
5. Output q

Fig. 6. Safe-prime generation algorithm for $q \in [q_{\min}, q_{\max}]$.

It is straightforward to extend our algorithm to the case of d-quasi-safe prime numbers whenever $d < \tau$. In this case, the constraint $q \equiv 3 \pmod 4$ has to be extended to $q \equiv 2^d + 1 \pmod{2^{d+1}}$.

A note on efficiency. Heuristically, about

$$\left(n_0 \cdot \ln 2 \cdot \frac{\phi(\Pi)}{\Pi} + 1\right)\left(n_0 \cdot \ln 2 \cdot \frac{\phi(\Pi)}{\Pi}\right)$$

primality tests are required for generating a n_0-bit safe prime q. This is ≈ 25 times faster than incremental search algorithms (where we iterate $q \leftarrow q+2$ until q and $(q-1)/2$ are simultaneously prime) for 512-bit numbers. Another obvious benefit of our technique resides in its simplicity when compared to classical algorithms.

5 Conclusion

We devised and analyzed simple computational techniques that improve the work of [13] in multiple ways. It is argued that our algorithms present much better performances than previous, classical methods.

We also would like to stress that our prime generation algorithm may support additional modifications *mutatis mutandis* in order to simultaneously reach other properties on q — for instance forcing the last bits of q to fit the Rabin-Williams cryptosystem with even public exponents. Independently, some applications require that the pair of primes satisfy specific properties such as being strong or compliant with ANSI X9.31 recommendations [1]. We refer the reader to [13] for a collection of mechanisms allowing to produce such primes. We point out that our improvements may coexist perfectly with these.

We also proposed a specific implementation for generating safe prime numbers which really boosts real-life execution performances. We emphasize that, implementing our techniques, a complete RSA key generation process can be executed on any given crypto-enhanced embedded processor in nearly all circumstances and with extremely attractive running times.

References

1. ANSI X9.31. Public-key cryptography using RSA for the financial services industry. American National Standard for Financial Services, draft, 1995.
2. A.O.L. Atkin and F. Morain. Elliptic curves and primality proving. *Mathematics of Computation*, vol. 61, pp. 29–68, 1993.
3. D. Boneh and M. Franklin. Efficient generation of shared RSA keys. In *Advances in Cryptology – CRYPTO '97*, vol. 1294 of Lecture Notes in Computer Science, pp. 425–439, Springer-Verlag, 1997.
4. W. Bosma and M.-P. van der Hulst. Faster primality testing. In *Advances in Cryptology – CRYPTO '89*, vol. 435 of Lecture Notes in Computer Science, pp. 652–656, Springer-Verlag, 1990.
5. J. Brandt and I. Damgård. On generation of probable primes by incremental search. In *Advances in Cryptology – CRYPTO '92*, vol. 740 of Lecture Notes in Computer Science, pp. 358–370, Springer-Verlag, 1993.
6. J. Brandt, I. Damgård, and P. Landrock. Speeding up prime number generation. In *Advances in Cryptology – ASIACRYPT '91*, vol. 739 of Lecture Notes in Computer Science, pp. 440–449, Springer-Verlag, 1991.
7. R.D. Carmichael. *Introduction to the Theory of Groups of Finite Order*, Dover, 1956.
8. C. Couvreur and J.-J. Quisquater. An introduction to fast generation of large prime numbers. *Philips Journal of Research*, vol. 37, pp. 231–264, 1982.
9. C. Ding, D. Pei, and A. Salomaa. *Chinese Remainder Theorem*, Word Scientific, 1996.
10. P.X. Gallagher. On the distribution of primes in short intervals. *Mathematica*, vol. 23, pp. 4–9, 1976.
11. G.H. Hardy and J.E. Littlewood. Some problems of 'Partitio Numerorum' III: On the expression of a number as a sum of primes. *Acta Mathematica*, vol. 44, pp. 1–70, 1922.
12. M. Joye and P. Paillier. Fast generation of prime numbers on portable devices: An update. *Extended version of this work*. Available on http://eprint.iacr.org.
13. M. Joye, P. Paillier, and S. Vaudenay. Efficient generation of prime numbers. In *Cryptographic Hardware and Embedded Systems – CHES 2000*, vol. 1965 of Lecture Notes in Computer Science, pp. 340–354, Springer-Verlag, 2000.
14. D.E. Knuth. *The Art of Computer Programming - Seminumerical Algorithms*, vol. 2, Addison-Wesley, 2nd ed., 1981.

15. C. Lu and A.L.M. Dos Santos. A note on efficient implementation of prime generation in small portable devices. *Computer Networks*, vol. 49, pp. 476–491, 2005.
16. C. Lu, A.L.M. Dos Santos, and F.R. Pimentel. Implementation of fast RSA key generation on smart cards. In *17th ACM Symposium on Applied Computing*, pp. 214–221, ACM Press, 2002.
17. U. Maurer. Fast generation of prime numbers and secure public-key cryptographic parameters. *Journal of Cryptology*, vol. 8, pp. 123–155, 1995.
18. A.J. Menezes, P.C. van Oorschot, and S.A. Vanstone. *Handbook of Applied Cryptography*, CRC Press, 1997.
19. L. Monier. Evaluation and comparison of two efficient probabilistic primality testing algorithms. *Theoretical Computer Science*, vol. 12, pp. 97–108, 1980.
20. H.C. Pocklington. The determination of the prime or composite nature of large numbers by Fermat's theorem. *Proc. of the Cambridge Philosophical Society*, vol. 18, pp. 29–30, 1914.
21. J.-J. Quisquater and C. Couvreur. Fast decipherment algorithm for RSA public-key cryptosystem. *Electronics Letters*, vol. 18, pp. 905–907, 1982.
22. H. Riesel. *Prime Numbers and Computer Methods for Factorization*, Birkhäuser, 1985.
23. R.L. Rivest, A. Shamir, and L.M. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, vol. 21, pp. 120–126, 1978.
24. R.D. Silverman. Fast generation of random, strong RSA primes. *Cryptobytes*, vol. 3, pp. 9–13, 1997.
25. R. Solovay and V. Strassen. A fast Monte-Carlo test for primality. *SIAM Journal on Computing*, vol. 6, pp. 84–85, 1977.

A The RSA Primitive

RSA is certainly the most widely used public-key cryptosystem today. We give hereafter a short description of the RSA primitive and refer the reader to the original paper [23] or any textbook in cryptography (e.g., [18]) for further details.

Let $N = pq$ be the product of two large primes. We let e and d denote a pair of public and private exponents, satisfying

$$e\,d \equiv 1 \pmod{\lambda(N)}\ ,$$

with $\gcd(e, \lambda(N)) = 1$ and λ being Carmichael's function. As $N = pq$, we have $\lambda(N) = \mathrm{lcm}(p-1, q-1)$. Given $x < N$, the public operation (e.g., message encryption or signature verification) consists in raising x to the e-th power modulo N, i.e., in computing $y = x^e \bmod N$. Then, given y, the corresponding private operation (e.g., decryption of a ciphertext or signature generation) consists in computing $y^d \bmod N$. From the definition of e and d, we obviously have that $y^d \equiv x \pmod N$. The private operation can be carried out at higher speed through Chinese remaindering (CRT mode [21, 9]). Computations are independently performed modulo p and q and then recombined. In this case, private parameters are $\{p, q, d_p, d_q, i_q\}$ with

$$\begin{cases} d_p = d \bmod (p-1)\,, \\ d_q = d \bmod (q-1)\,, \text{ and} \\ i_q = q^{-1} \bmod p\,. \end{cases}$$

We then obtain $y^d \bmod N$ as

$$\mathrm{CRT}(x_p, x_q) = x_q + q\left[i_q(x_p - x_q) \bmod p\right]\,,$$

where $x_p = y^{d_p} \bmod p$ and $x_q = y^{d_q} \bmod q$. We expect a theoretical speed-up factor close to 4 (see [21]), compared to the standard, non-CRT mode.

Thus, an *RSA modulus* $N = pq$ is the product of two large prime numbers p and q. If n denotes the bitsize of N then, for some $1 < n_0 < n$, p must lie in the range $\left[\lceil 2^{n-n_0-1/2}\rceil, 2^{n-n_0}\right]$ and q in the range $\left[\lceil 2^{n_0-1/2}\rceil, 2^{n_0}\right]$ so that $2^{n-1} < N = pq < 2^n$. For security reasons, so-called balanced moduli are generally preferred, which means $n = 2n_0$.