Efficient Generation of Prime Numbers*

Marc Joye[1], Pascal Paillier[1], and Serge Vaudenay[2]

[1] Gemplus Card International, France
{Marc.Joye,Pascal.Paillier}@gemplus.com
[2] École Polytechnique Fédérale de Lausanne, Switzerland
Serge.Vaudenay@epfl.ch

Abstract. The generation of prime numbers underlies the use of most public-key schemes, essentially as a major primitive needed for the creation of key pairs or as a computation stage appearing during various cryptographic setups. Surprisingly, despite decades of intense mathematical studies on primality testing and an observed progressive intensification of cryptographic usages, prime number generation algorithms remain scarcely investigated and most real-life implementations are of rather poor performance. Common generators typically output a n-bit prime in heuristic average complexity $O(n^4)$ or $O(n^4/\log n)$ and these figures, according to experience, seem impossible to improve significantly: this paper rather shows a simple way to substantially reduce the value of hidden constants to provide much more efficient prime generation algorithms. We apply our techniques to various contexts (DSA primes, safe primes, ANSI X9.31-compliant primes, strong primes, etc.) and show how to build fast implementations on appropriately equipped smart-cards, thus allowing on-board key generation.

Keywords. Prime number generation, key generation, RSA, DSA, fast implementations, crypto-processors, smart-cards.

1 Introduction

Traditional prime number generation algorithms asymptotically require $O(n^4)$ or $O(n^4/\log n)$ bit operations where n is the bit-length of the expected prime number. This complexity may even become of the order of $O(n^5/(\log n)^2)$ in the case of *constrained* primes, such as safe or quasi-safe primes for instance. These asymptotic behaviors,[1] according to experience, seem impossible to improve significantly. In this paper, we rather propose simple algebraic methods which substantially reduce the value of the hidden constants, thus providing much more efficient prime generation algorithms.

---

* Some parts presented in this paper are patent pending.

[1] assuming that multiplications modulo q are in $O(|q|^2)$. Theoretically, one could decrease this complexity by using multiplication algorithms such as Karatsuba in $O((|q|^{\log_2 3})$ or Schönhage-Strassen in $O(|q|\ \log|q|\ \log\log|q|)$.

We apply our techniques to various contexts such as DSA primes [9], strong primes [14] and ANSI X9.31-compliant primes [1], that is, real-life scenarios of well-recognized utility. As an illustration, we also reduce the number of rounds of Boneh and Franklin's [3] shared RSA keys protocol by a factor of nearly 10.

Finally, our techniques allow fast implementations on cryptographic smart-cards for on-board RSA [15] (or other schemes) key generation. Our motivation here is to help transferring this task from terminals to smart-cards themselves in the near future for more confidence, security, and compliance with network-scaled distributed protocols that include smart-cards, such as electronic cash or mobile commerce.

Notations. Throughout this paper, the following notations are used. Other notations will be introduced when needed.

Symbol	Signification
$\#\mathcal{A}$	cardinal of a set $\mathcal{A}$
$\lvert x\rvert$	bit-length of number x
$\{0,1\}^t$	set of t-bit numbers
$\mathbb{Z}_\Pi$	ring of integers modulo Π
$\mathbb{Z}_\Pi^*$	multiplicative group of $\mathbb{Z}_\Pi$
$\phi(\cdot)$	Euler's totient function
$\lambda(\cdot)$	Carmichael's function
$\mathrm{O}(\cdot)$	asymptotic bound
$\Omega(\cdot)$	asymptotic equivalence
$a \approx b$	a is approximatively equal to b
$a \gtrsim b$	$a \approx b$ and $a \geq b$
$a \lesssim b$	$a \approx b$ and $a \leq b$

2 Primality and Compositeness Tests

A lot of studies on primality testing have been carried out for years, and can be found in the literature devoted to the subject (e.g., see [7]). Computationally, we may distinguish *true primes* and *probable primes*: the difference being the way these are generated. A probable prime is usually obtained through a *compositeness test*. Such a test declares that a number is composite with probability 1 or prime with some probability < 1. Hence repeatedly running the test gives more and more confidence in the generated (probable) prime. Typical examples of compositeness tests include Fermat test, Solovay-Strassen test [16], and Miller-Rabin test [10, p. 379].

There also exist (true) *primality tests*, which declare a number prime with probability 1 (e.g., Pocklington's test [12] and its elliptic curve analogue [2], the Jacobi sum test [4]). However, these tests are generally more expensive or intricate.

To motivate further analysis, we hereafter assume that we are given some compositeness test T provided as a primality oracle of complexity $\tau(n) = \mathrm{O}(n^3)$

and of negligible error probability. Designing an efficient prime generation algorithm then reduces to the problem of knowing how to use T in order to produce a n-bit prime with a minimal number of calls to the oracle.

3 Generating Primes: Prior Art

3.1 Naive Generators

We refer to the naive prime number generator as the following:

1. pick a random n-bit odd number q
2. if $\mathsf{T}(q) = \texttt{false}$ then goto 1
3. output q

Fig. 1. Naive prime number generator.

Neglecting calls to the random number generator, the expected number of trials here is asymptotically equal to $(\ln 2^n)/2 \approx 0.347\, n$. Generating a 256-bit prime thus requires 89 trials in average.

The previous algorithm has an incremental variant, which is given below on Fig. 2.

1. pick a random n-bit odd number q
2. while $\mathsf{T}(q) = \texttt{false}$ do $q \leftarrow q + 2$
3. output q

Fig. 2. Naive incremental prime number generator.

It should be outlined that this second algorithm has not the same proven complexity [5]. A proper analysis actually has to exploit the properties of the distribution of prime numbers, in connection with Riemann's Hypothesis. The incremental generator is however commonly used and we recall that it was shown to fail with probability $\mathrm{O}(t^3\, 2^{-\sqrt{t}})$ after $\Omega(t)$ trials (see [11, p. 148]).

3.2 Classical Generation Algorithms

The naive incremental generator can be made more efficient by choosing the initial candidate q already co-prime to small primes. Usually, one defines $\Pi = 2 \cdot 3 \cdots 29$ and randomly chooses a n-bit number q satisfying $\gcd(q, \Pi) = 1$. If $\mathsf{T}(q) = \texttt{false}$ then q is updated as $q \leftarrow q + \Pi$ (note that the naive generator corresponds to the special case $\Pi = 2$). If Π is a constant independent from n and contains k distinct primes, we denote this probabilistic algorithm by $\mathsf{H}[n, k]$.

The next section presents two new algorithms. The first one, making use of look-up tables, produces random numbers *constructively* co-prime to small primes. The second algorithm, slightly slower, is space-optimized and particularly suited for smart-card implementations. Based on this, we construct a new prime generation algorithm in Section 5 and give timing results in Section 6. Finally, in Section 7, we apply these new techniques to particular contexts.

4 Generating Invertible Numbers modulo a Product of Primes

Common prime number generators generally include a stage of trial divisions by small primes. We investigate in this section a way of avoiding this stage by efficiently constructing candidates that already satisfy co-primality properties. We base our constructions on simple algebraic techniques.

4.1 A Table-based Method

Let $\Pi = \prod_{i=1}^{k} {p_i}^{\delta_i}$ be a n-bit product of the first k primes with some small exponents δ_i. Let $\Delta = \max_i \delta_i$. We denote by $x = (x_1, \ldots, x_k)_\equiv$ the modular representation of $x \in \mathbb{Z}_\Pi$, i.e., $x_i = x \bmod p_i^{\delta_i}$. For $i = 1, \ldots, k$, one then defines $\theta_i = (0, \ldots, 1, \ldots, 0)_\equiv$ where the "1" stands in i^{th} position. It is obvious to see that we always have

$$\forall x \in \mathbb{Z}_\Pi \qquad x = \sum_{i=1}^{k} x_i\, \theta_i \bmod \Pi \ ,$$

that is, the function $x \mapsto (x_i)$ is a bijection[2] from $\mathbb{Z}_{p_1^{\delta_1}} \times \cdots \times \mathbb{Z}_{p_k^{\delta_k}}$ into $\mathbb{Z}_\Pi$. This function also defines a bijection from $\mathbb{Z}^*_{p_1^{\delta_1}} \times \cdots \times \mathbb{Z}^*_{p_k^{\delta_k}}$ to $\mathbb{Z}^*_\Pi$, and it follows that

$$\begin{aligned} \forall x \in \mathbb{Z}_\Pi \qquad x \in \mathbb{Z}^*_\Pi &\iff x_i \in \mathbb{Z}^*_{p_i^{\delta_i}} \\ &\iff x_i^{\delta_i} \not\equiv 0 \pmod{p_i^{\delta_i}} \\ &\iff x_i^{\delta_i}\theta_i \not\equiv 0 \pmod{\Pi} \quad \text{for } i = 1, \ldots, k \ . \end{aligned} \tag{1}$$

As a consequence, it appears that $x \in \mathbb{Z}^*_\Pi$ can be built-up from numbers x_i as long as they verify Eq. (1) above. We then define $\mathcal{A}$ as a set of random sequences $\alpha = (\alpha_1, \alpha_2, \ldots)$ with $\alpha_i \in \{0, 1\}^t$. Equation (1) gives a natural way of surjectively transforming any $\alpha \in \mathcal{A}$ into an invertible number $\mathsf{g}(\alpha) \in \mathbb{Z}^*_\Pi$. The corresponding algorithm, g, is depicted on Fig. 3.

[2] This is the usual Chinese Remainder Theorem correspondence [8].

Precomputations: $\Pi = \prod_{i=1}^{k} p_i^{\delta_i}$, $t = C \cdot \max_i |p_i^{\delta_i}|$ $(C = 2)$, $\{\theta_i\}$
Input: a random sequence α
Output: an invertible number c modulo Π

1. $c = 0$
2. for $i = 1$ to k
   - 2.1 pick a random t-bit number α_i from α
   - 2.2 if $\alpha_i^{\delta_i}\theta_i \bmod \Pi = 0$ goto 2.1
   - 2.3 $c \leftarrow c + \alpha_i\,\theta_i \bmod \Pi$
3. output c

Fig. 3. Generator g of invertible numbers modulo Π.

Since we use t-bit numbers α_i and reduce them (implicitly) modulo $p_i^{\delta_i}$, there exists a bias (lying around 2^{-t}) leading to a non-uniform output distribution.[3] But this underlying bias may easily be made arbitrarily small by increasing t (which negatively affects the average complexity as well). We therefore suggest $t = C \cdot \max_i |p_i^{\delta_i}|$ as a good compromise, where the ratio C may be fixed to 2 for practical implementations. Further, we claim (for a negligible bias) that the function $\mathsf{g} : \mathcal{A} \to \mathbb{Z}_\Pi^*$ verifies

- (i) to be surjective;
- (ii) that for each element x of $\mathbb{Z}_\Pi^*$, the number of x's pre-images is about $\#\mathcal{A}/\#\mathbb{Z}_\Pi^* = \#\mathcal{A}/\,\phi(\Pi)$, and this guarantees the uniformity of g's outputs from its inputs;
- (iii) g has a low (time) complexity $\gamma(n) = \mathrm{O}(n^2)$.

4.2 Modular Search Method

As aforementioned, the algorithm g generates uniformly distributed elements of $\mathbb{Z}_\Pi^*$. Although the execution time of g happens to be excellent when using an arithmetic processor, the memory space needed to store the numbers $\{\theta_i\}$ may appear dissuasive, in particular on a smart-card where memory may be subject to strong size constraints. We propose here a simple alternative method based on Carmichael's theorem[4]

$$\forall c \in \mathbb{Z}_\Pi^* \qquad c^{\lambda(\Pi)} \equiv 1 \pmod{\Pi}\ ,$$

or more exactly on its converse:

Proposition 1. *$\forall c \in \mathbb{Z}_\Pi$, if $c^{\lambda(\Pi)} \equiv 1 \pmod{\Pi}$ then $c \in \mathbb{Z}_\Pi^*$.*

[3] It nevertheless seems a hard task to exploit it in some way for *a posteriori* secret key retrieval.

[4] If $\Pi = \prod_{i=1}^{k} p_i^{\delta_i}$ then $\lambda(\Pi) = \mathrm{lcm}[\lambda(p_i^{\delta_i})]_{i=1,\ldots,k}$, and $\lambda(p_i^{\delta_i}) = \phi(p_i^{\delta_i}) = p_i^{\delta_i-1}(p_i-1)$ for an odd prime p_i, $\lambda(2) = 1$, $\lambda(4) = 2$ and $\lambda(2^{\delta_i}) = \frac{1}{2}\,\phi(2^{\delta_i}) = 2^{\delta_i-2}$ for $\delta_i \geq 3$.

Proof. A number $0 \leq c < \Pi$ is in $\mathbb{Z}_\Pi^*$ if and only if, for all primes p dividing Π, we have $\gcd(c,p) = 1 \iff c^{p-1} \equiv 1 \pmod{p}$ so that $c^{\lambda(\Pi)} \equiv 1 \pmod{\Pi}$ by Chinese remaindering. □

This provides an easy co-primality test that requires a single modular exponentiation with exponent $\lambda(\Pi)$. Note that this technique only needs the storage of Π and $\lambda(\Pi)$, and is also particularly suitable for crypto-processors. In addition, since Π is smooth, $\lambda(\Pi)$ is optimally small. The obtained procedure is depicted below.

Precomputations: $\Pi = \prod_{i=1}^{k} p_i^{\delta_i}$ and $\lambda(\Pi)$
Output: an invertible number c modulo Π

1. pick an n-bit random number $c < \Pi$
2. while $c^{\lambda(\Pi)} \bmod \Pi \neq 1$ do $c \leftarrow c + 1$
3. output c

Fig. 4. Generator g' of invertible numbers modulo Π.

The previous algorithm can be improved via Chinese remaindering. Instead of testing the co-primality of c to Π, one checks the co-primality to some factor of Π, say π_1. If $\gcd(c, \pi_1) \neq 1$ (i.e., if $c^{\lambda(\pi_1)} \bmod \pi_1 \neq 1$) then we already know that $\gcd(c, \Pi) \neq 1$. Otherwise, we test the co-primality of c with another factor π_2 of Π, and so on for several factors π_i until $\prod_i \pi_i = \Pi$ (or is a multiple of Π).

Although the complexity of g' may appear greater than $\gamma(n)$, the comparison must take into account the computational features of the underlying crypto-processor (see Section 6). Of course, the implementer shall choose between generators g and g' (or a variant) according to the necessity of saving time (using g) or space (using g'). We consider in the following that this choice has been done once for all, and that a black-box generator (hereafter referred to as g) of elements of $\mathbb{Z}_\Pi^*$ is at disposal: we now have to deal with how to design a prime generation algorithm in which primitives T and g get optimally exploited.

5 An Efficient Prime Generation Algorithm

Generated primes are expected to lie in some target window $\mathcal{F} = [w_{\min}, w_{\max}]$, where $w_{\max} = 2^n - 1$ in most contexts, and $w_{\min}$ is equal to $2^{n-1} + 1$ when generating n-bit primes, or $\left\lceil \sqrt{2^{2n-1}+1} \right\rceil$ if the context imposes to obtain a strict $2n$-bit number when multiplying two so-generated primes (RSA moduli for instance).

The basic idea consists in utilizing g to produce a sequence of candidates that will be tested one by one until a prime is found. We now describe how we choose parameter Π. First, we find an integer η containing a maximum number

Output: a n-bit prime q

1. $c = \mathsf{g}()$
2. $q = c + \rho$
3. if $\mathsf{T}(q) = \texttt{false}$ then $c \leftarrow f_a(c)$ and goto 2.
4. output q

Fig. 5. $\mathsf{G}[n]$ – A basic prime number generator based on g.

of (different) primes (or more precisely minimizing the ratio $\phi(\eta)/\eta$) and such that there exist small integers $\varepsilon_{\min}$ and $\varepsilon_{\max}$ satisfying

$$\varepsilon_{\min}\eta \gtrapprox w_{\min} \quad \text{and} \quad \varepsilon_{\max}\eta \lessapprox w_{\max} - w_{\min} \enspace .$$

We then set

$$\Pi = \varepsilon_{\max}\,\eta \quad \text{and} \quad \rho = \varepsilon_{\min}\,\eta \enspace . \tag{2}$$

Once an invertible element $c^{(1)} \in \mathbb{Z}_\Pi^*$ is generated (using g), the first prime candidate is defined as

$$q^{(1)} = c^{(1)} + \rho \enspace .$$

Note that $\gcd(q^{(1)}, \eta) = \gcd(c^{(1)} + \rho, \eta) = \gcd(c^{(1)}, \eta) = 1$ since $c^{(1)} \in \mathbb{Z}_\Pi^*$; note also that $q^{(1)} \in \mathcal{F}$. We let $\mathcal{P}_0$ denote the set $(\mathbb{Z}_\Pi^* + \rho) \subseteq \mathcal{F}$, and $\mathcal{P}_c$ the set of primes belonging to $\mathcal{P}_0$. For avoiding systematic use of g, rejected candidates should optimally be transformed and re-used in order to continue the search. In this setting, the transition step $c^{(i+1)} = f_a(c^{(i)})$ uses the stability of $\mathbb{Z}_\Pi^*$ under multiplication by setting

$$c^{(i+1)} = f_a(c^{(i)}) = a\,c^{(i)} \bmod \Pi \quad \text{and} \quad q^{(i+1)} = c^{(i+1)} + \rho \;, \tag{3}$$

where a is a constant appropriately chosen in $\mathbb{Z}_\Pi^*$. We call $\mathsf{G}[n]$ the corresponding algorithm as illustrated in Fig. 5.

The produced *search sequence* $\{q^{(1)}, q^{(2)}, \ldots, q^{(d)}\}$ ends when $q^{(d)} \in \mathcal{P}_c$. Naturally, one has to make sure that the order of f_a (seen as a permutation over $\mathbb{Z}_\Pi^*$) is large enough, that is, a's order in $\mathbb{Z}_\Pi^*$ must be sufficiently large (since $c^{(i+1)} = a^i\,c^{(1)} \bmod \Pi$): otherwise the search sequence could possibly reach a cyclic set of values without ending.

By denoting $\sigma(n, a)$ and $\tau(n)$ the complexity of f_a and T respectively, and $\mathsf{Comp}(n)$ the average time complexity of $\mathsf{G}[n]$, it can be shown that

$$\mathsf{Comp}(n) = \gamma(n) + (\bar{d} - 1)\,\sigma(n, a) + \bar{d}\,\tau(n) \;, \tag{4}$$

where $\bar{d}$ denotes the average sequence length over many trials. Making the heuristic approximation that the random variables induced by the $q^{(i)}$s are independent and uniformly distributed, we get

$$\bar{d} = \frac{\#\mathcal{P}_0}{\#\mathcal{P}_c} \propto \frac{\phi(\eta)}{\eta} \enspace . \tag{5}$$

It can be shown that our heuristic algorithm $\mathsf{G}[n]$ outputs random n-bit primes in average time complexity $\mathrm{O}(n^4/\log n)$, although we do not give a proof of this fact here due to the lack of space.

¿From a practical viewpoint, since g and T are given, the only remaining degree of freedom resides in f_a. Note that $\sigma(n,a)$ is multiplied by a potentially big factor, $\#\mathcal{P}_0/\#\mathcal{P}_c$ in (4), so that decreasing $\sigma(n,a)$ leads to a proportional gain in $\mathsf{Comp}(n)$.

We now specialize Eq. (3) so that the transition step is very fast: the best possible value is $a = 2$. In this respect, we exclude $p_1 = 2$ from Π's factorization. The benefit is immediate due to the fact that for all $c \in \mathbb{Z}_\Pi$, $f_2(c) = 2c \bmod \Pi$ only requires *non-modular* additions: $f_2(c) = 2c$ or $2c - \Pi$. Note here that, since Π is even, $f_2(c)$ can be odd or even. Hence from Eq. (3), our candidate for primality $q = c + \rho$ can be even! So, in order to avoid useless tests, we suggest the following modification: we define Π as $\Pi = (\varepsilon_{\max} - 1)\eta$ and ρ as in Eq. (2). Next, $q = c + \rho$ is optionally added to η according to its parity so that the resulting q is always odd. Here is the final algorithm. We give practical values for η, Π and ρ to generate 512-bit primes in the next section.

Precomputations: parameters η, $\Pi = (\varepsilon_{\max} - 1)\eta$, and $\rho = \varepsilon_{\min}\,\eta$
Output: a n-bit prime q

1. $c = \mathsf{g}()$
2. $q = c + \rho$
3. if q is even then $q \leftarrow q + \eta$
4. if $\mathsf{T}(q) = \texttt{false}$ then $c \leftarrow 2c \bmod \Pi$ and goto 2
5. output q

Fig. 6. $\mathsf{GPrime}[n]$ – An optimized prime number generator.

Alternatively, one may use an even Π and fix a to some particular invertible value modulo Π so that multiplying by a requires very few operations (e.g., $a = 2^{16} + 1$).

6 Implementation Results

After having implemented g on Infineon's SLE66CX160S smart-card platform (8-bit CPU and 1100-bit arithmetic crypto-processor) for $n = 512$ and

$$
\begin{aligned}
\eta &= \texttt{b16bd1e084af628fe5089e6dabd16b5b80f60681d6a092fc} \\
& \texttt{b1e86d82876ed71921000bcfdd063fb90f81dfd} \\
& \texttt{07a021af23c735d52e63bd1cb59c93cbb398afd}_{16}\ , \\
\Pi &= 1729 \cdot \eta\ , \\
\rho &= 4180 \cdot \eta\ ,
\end{aligned}
$$

we compute a uniformly distributed[5] random invertible number modulo Π in less than 40 ms at 3.57 MHz. Algorithm on Fig. 5 with $a = 2$ runs in about 3.150 seconds in average to generate a 512-bit prime, which is in high accordance with Eq. (4) (T is a basic Fermat test with base 2 running in $\tau(512) \approx 90$ ms). As a direct consequence, this particularly fast smart-card implementation allows 1024-bit RSA keys on-board generation in less than 8 seconds in average. The generation of an invertible number using g requires about 2.7 KB of code memory (due to the storage of the θ_i). Such a large memory consumption can be avoided by replacing g with g', which only implies the storage of Π and

$$\lambda(\Pi) = \texttt{1dc6c203d4cc780033f9c5d8d97aa2468a54e3700}_{16} \ .$$

This implementation choice has a little impact on performances since the whole RSA key generation process still runs in less than 10 seconds in average. As a comparison with classical methods, we give on Fig. 6 the (heuristic) expected number of calls to T needed by $\mathsf{G}[n]$ and $\mathsf{H}[n, 10]$.

n	256	384	512	640	768	896	1024
$\mathsf{G}[n]$	18.72	26.12	33.29	40.25	46.90	53.56	59.98
$\mathsf{H}[n, 10]$	28.03	42.04	56.05	70.07	84.08	98.1	112.1

Fig. 7. $\mathsf{G}[n]$ *vs* $\mathsf{H}[n, 10]$ – Heuristic expected number of calls to the primality oracle T.

7 Applications

We now apply the previously analyzed tools to some particular contexts. We believe that these techniques constitute a serious improvement on current prime number generators in almost every circumstances, including while implementing ANSI X9.31 recommendations.

7.1 Generation of DSA Primes

Here we focus on the problem of generating a uniformly distributed random n-bit prime $p = 1 + qr$ for a given 160-bit prime q. Trial divisions are intended to check that the candidate p has no prime factor p_i for $i = 1, \ldots, k$. As before, we can advantageously generate r so that p automatically fulfills this condition. It suffices that

$$p \not\equiv 0 \pmod{p_i} \iff r \not\equiv -\frac{1}{q} \pmod{p_i} \qquad \text{for } i = 1, \ldots, k \ . \tag{6}$$

[5] Again, we consider the bias of Section 4 to be negligible.

Choosing $\Pi = p_1^{\delta_1} \cdots p_k^{\delta_k}$ with $|\Pi| = |r| = n - 160$, Eq. (6) can be rewritten as

$$r = -\frac{1}{q} + c \bmod \Pi \tag{7}$$

where $c \in \mathbb{Z}_\Pi^*$.

Based on Fig. 5, we therefore propose algorithm $\mathsf{GDSA}[n, q]$. Again, as in Section 5, $\mathsf{g}()$ generates elements of $\mathbb{Z}_\Pi^*$ and $f_a(c) = ac \bmod \Pi$ for some $a \in \mathbb{Z}_\Pi^*$.

Input: a 160-bit prime q
Output: a n-bit prime satisfying $p = 1 + rq$
1. compute $1/q = q^{\lambda(\Pi)-1} \bmod \Pi$
2. $c = \mathsf{g}()$
3. $r = (-1/q + c) \bmod \Pi$
4. $p = 1 + qr$
5. if $\mathsf{T}(p) = \texttt{false}$ then $c \leftarrow f_a(c)$ and goto 3
6. output p

Fig. 8. $\mathsf{GDSA}[n, q]$ – DSA prime generation algorithm based on g.

As a comparison with classical methods, we also give benchmarks for $\mathsf{GDSA}[n, q]$ and $\mathsf{H}[n, 10]$.

n	256	384	512	640	768	896	1024
$\mathsf{GDSA}[n, q]$	22.37	28.71	35.34	42.06	48.62	55.12	61.6
$\mathsf{H}[n, 10]$	28.03	42.04	56.05	70.07	84.08	98.1	112.1

Fig. 9. $\mathsf{GDSA}[n, q]$ – Heuristic expected number of calls to T.

7.2 Generation of Safe Primes

A prime p is said to be safe if $p = 1 + 2q$ where q is also prime. In order to generate a safe n-bit prime $p = 1 + 2q$, we have to produce a search sequence of pairs $(p^{(i)}, q^{(i)})$ in which $p^{(i)} = 1 + 2q^{(i)}$ and $p^{(i)}$, $q^{(i)}$ are both invertible modulo Π. This can be done by finding for $\Pi = p_1^{\delta_1} \cdots p_k^{\delta_k}$ a value close to 2^{n-2} with maximum k. As we know how to generate an element c of $\mathbb{Z}_\Pi^*$, we propose to test $q^{(1)} = c + \Pi$ and $p^{(1)} = 1 + 2c + 2\Pi$ for primality. By construction, since $c \in \mathbb{Z}_\Pi^*$, $q^{(1)}$ is indeed co-prime to Π and thus makes a good candidate for being a prime: this is however not the case for $p^{(1)}$. For solving this drawback, we propose to modify g into g_s as given in Fig. 10.

Precomputations: $\Pi = \prod_1^k p_i^{\delta_i}$, $t = C \cdot \max_i |p_i^{\delta_i}|$ $(C = 2)$, $\{\theta_i\}$
Input: a random sequence α
Output: a uniformly distributed invertible number $c \in \mathbb{Z}_\Pi^*$ with $1 + 2c \in \mathbb{Z}_\Pi^*$

1. $c = 0$
2. for $i = 1$ to k
   - 2.1 pick a t-bit random number α_i
   - 2.2 if $\alpha_i^{\delta_i}\theta_i \bmod \Pi = 0$ goto 2.1
   - 2.3 if $(1 + 2\alpha_i)^{\delta_i}\theta_i \bmod \Pi = 0$ goto 2.1
   - 2.4 $c \leftarrow c + \alpha_i\theta_i \bmod \Pi$
3. output c

Fig. 10. Generator g_s for safe prime generation.

Ouput: a safe n-bit prime $p = 1 + 2q$ with q prime

1. $c = \mathsf{g}_s()$
2. $q = c + \Pi$
3. $p = 1 + 2q$
4. if $\mathsf{T}(p) = \texttt{false}$ or $\mathsf{T}(q) = \texttt{false}$ goto 1
5. output p

Fig. 11. $\mathsf{Gsafe}[n]$ – safe prime generation algorithm.

¿From this modified generator, we naturally define an algorithm $\mathsf{Gsafe}[n]$ generating safe primes as shown on Fig. 11. Since it appears uneasy to find a low-cost transformation $(p^{(i+1)}, q^{(i+1)}) = f(p^{(i)}, q^{(i)})$ that respects co-primality to Π, g_s is recalled as many times as necessary.

7.3 Application to ANSI X9.31

In order to thwart certain classes of attacks on RSA, the ANSI recommends the use of prime factors satisfying particular properties as exposed in the specifications of X9.31. According to the standard, each prime factor q must be chosen such that

$$\begin{cases} q - 1 \text{ has a large prime divisor } u, \\ q + 1 \text{ has a large prime divisor } s, \end{cases}$$

where the respective sizes of u and s are chosen close to 100 bits. Primes numbers featuring this property will be called X9.31-compliant primes. We first note that, after having chosen parameters $\eta \approx 2^{99}$ and $\Pi = \rho = \eta$, our algorithm $\mathsf{G}[100]$ outputs two 100-bit prime numbers u and s with an expected complexity of 8.73 primality tests. We still have to generate a n-bit prime q such that

$$q = 1 + r_1 \cdot u = -1 + r_2 \cdot s \ ,$$

where r_1 and r_2 are integers of bit-size $n-100$. Hence $r_1 \equiv -\frac{2}{u} \pmod{s}$ and there must be an integer r_3 such that

$$q = 1 + u \cdot \left(-\frac{2}{u} \bmod s + r_3 \cdot s\right) .$$

By a reasoning similar to the one of Section 7.1, we are driven to produce candidates q of the preceding form with

$$r_3 = -\frac{1}{su} - \frac{-2u^{-1} \mod s}{s} + c \mod \Pi ,$$

where $c \in \mathbb{Z}_\Pi^*$ and Π is a product of small primes of total size close to $n-200$. Note also that the intermediate computations

$$\kappa = 1 + u(-2u^{-1} \mod s)$$

and

$$\mu = -\kappa(su)^{-1} \mod \Pi$$

of respective bit-sizes 200 and $n-200$ can be done easily in two exponentiations

$$u^{-1} = u^{s-2} \mod s$$

and

$$(su)^{-1} = (su)^{\lambda(\Pi)-1} \mod \Pi .$$

This motivates algorithm $\mathsf{Gx9.31}[n]$ illustrated on Fig. 12. As before, a is a constant chosen in $\mathbb{Z}_\Pi^*$ and $f_a(c) = ac \mod \Pi$. We also give the expected number of calls to the primality oracle T as a function of n on Fig. 13.

Precomputations: Π and $\lambda(\Pi)$
Output: a X9.31-compliant n-bit prime q

1. generate u and s using $\mathsf{G}[100]$
2. compute $\kappa \leftarrow 1 + u \cdot (-2u^{-1} \mod s)$ and $\mu \leftarrow -\kappa(su)^{-1} \mod \Pi$
3. $c \leftarrow \mathsf{g}()$
4. $r \leftarrow \mu + c \mod \Pi$ and $q \leftarrow \kappa + su \cdot r$
5. if $\mathsf{T}(q) = \texttt{false}$ then $c \leftarrow f_a(c)$ and goto 4
6. output q

Fig. 12. $\mathsf{Gx9.31}[n]$ – X9.31-compliant prime generation algorithm.

n	256	384	512	640	768	896	1024
$\mathsf{Gx9.31}[n]$	25.15	29.64	36.05	42.68	49.18	55.54	61.90

Fig. 13. $\mathsf{Gx9.31}[n]$ – Heuristic expected number of calls to the primality oracle T.

7.4 Generation of Strong Primes

A prime number q is said to be *strong* when

$$\begin{cases} q-1 \text{ has a large prime divisor } u, \\ q+1 \text{ has a large prime divisor } s, \\ u-1 \text{ has a large prime divisor } t. \end{cases}$$

The property of being strong therefore implies $X9.31$-compliance. Usually, the bit-sizes of u, s and t are chosen fixed to constant values and hence do not depend on the bit-size of q, n. We will take $|s| = |t| = 100$ and $|u| = 130$ here as an illustrative example, despite the fact that our technique remains fully generic towards these parameters.

Clearly, we can take advantage of the algorithm $\mathsf{Gx9.31}[n]$ of the preceding section and include the additional stage $u = \mathsf{GDSA}[130, t]$ before the search sequence takes place. This can be done by setting $\Pi \approx 2^{29}$ in $\mathsf{GDSA}[130, t]$. This gives the algorithm $\mathsf{Gstrong}$ described on Fig. 14.

Precomputations: Π and $\lambda(\Pi)$
Output: a n-bit strong prime q

1. generate s and t using $\mathsf{G}[100]$
   generate u using $\mathsf{GDSA}[130, t]$
2. compute $\kappa \leftarrow 1 + u \cdot (-2u^{-1} \mod s)$ and
   $\mu \leftarrow -\kappa(su)^{-1} \mod \Pi$
3. $c \leftarrow \mathsf{g}()$
4. $r \leftarrow \mu + c \mod \Pi$ and $q \leftarrow \kappa + su \cdot r$
5. if $\mathsf{T}(q) = \texttt{false}$ then $c \leftarrow f_a(c)$ and goto 4
6. output q

Fig. 14. $\mathsf{Gstrong}[n]$ – A strong prime generation algorithm.

We stress the fact that our technique features a dramatic performance improvement compared to classical methods. To illustrate this, we give a comparison of the average number of calls to T executed by $\mathsf{Gstrong}$ and the classical method, Gordon's algorithm.

n	256	384	512	640	768	896	1024
Gstrong[n]	30.34	30.82	36.7	43.1	49.55	56	62.21
Gordon	88.73	133.1	177.45	221.8	266.17	310.53	354.9

Fig. 15. Gstrong[n] *vs* Gordon – Heuristic expected number of calls to the primality oracle T.

7.5 Application in a Shared RSA Protocol

In [3], Boneh and Franklin proposed a shared RSA protocol which enables two parties with the help of a third party to generate a shared RSA key $N = pq$ and $de \equiv 1 \pmod{\phi(N)}$. In this protocol, N and e are public but p, q and d are shared through a secret sharing algorithm.

The Boneh-Franklin protocol enables the two parties to decrypt. One crucial step in this protocol resides in the protocol generating N. Basically, both parties A and B choose (p_A, q_A) and (p_B, q_B) respectively, proceed to a protocol such that they share $N = (p_A + p_B)(q_A + q_B)$, and check that $p = p_A + p_B$ and $q = q_A + q_B$ are simultaneously prime. Prior to this protocol, A and B check whether p and q are not divisible by small primes. In other words, they first generate some shared p and q which have no small prime factors $p_1, \ldots, p_m$ and start again until p and q are both prime. This leads to an expected number of $\left(\frac{e^{-\gamma}\log 2n}{\log p_m}\right)^2$ joint primality tests. As an example, Boneh and Franklin proposed for $n = 512$ that m should be close to 1024 which leads to a number of trial division steps of 32. Alternatively we propose to generate p and q as

$$p = (p'_A p'_B + (p''_A + p''_B)\Pi) \bmod \varepsilon\Pi \quad \text{and} \quad q = (q'_A q'_B + (q''_A + q''_B)\Pi) \bmod \varepsilon\Pi$$

where $\Pi = \prod_{i=1}^{k} p_i$, $\varepsilon\Pi \approx 2^n$ and p'_A, p'_B, q'_A and q'_B are random numbers co-prime to Π generated by generator g, and then to perform trial divisions by $p_{k+1}, \ldots, p_m$. For $m = 1024$ and $k = 74$, letting $\chi = \prod_{i=75}^{1024} p_i$, the number of trials is then $\chi/\,\phi(\chi)$ which is approximately 3 instead of 32. This drastically reduces the number of exchanged values in the protocol.

8 Conclusion

We introduced new algorithms for generating pseudo-random numbers with no small factors, and showed how to use them in designing prime number generation algorithms to improve related problems. We gave a sketchy expression of our main algorithm's complexity in heuristic terms: this complexity relates to the distribution of prime numbers in the arithmetic progression $a^i c \mod \Pi + \rho$ with $i \geq 0$ and $a, c \in \mathbb{Z}_\pi^*$. Therefore, an open question would be to provide a more formal investigation on the distribution of those primes, the same way Brandt and Damgård [5] characterized the naive incremental generator.

Acknowledgements

We are grateful to the referees for their useful comments.

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. 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.
8. C. Ding, D. Pei, and A. Salomaa. *Chinese Remainder Theorem*, Word Scientific, 1996.
9. FIPS 186. Digital signature standard. Federal Information Processing Standards Publication 186, US Department of Commerce/N.I.S.T., 1994.
10. D.E. Knuth. *The Art of Computer Programming - Seminumerical Algorithms*, vol. 2, Addison-Wesley, 2nd ed., 1981.
11. A.J. Menezes, P.C. van Oorschot, and S.A. Vanstone. *Handbook of Applied Cryptography*, CRC Press, 1997.
12. 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.
13. H. Riesel. *Prime Numbers and Computer Methods for Factorization*, Birkhäuser, 1985.
14. R.L. Rivest. Remarks on a proposed cryptanalytic attack on the M.I.T. public-key cryptosystem. *Cryptologia*, vol. 2, pp. 62–65, 1978.
15. 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.
16. R. Solovay and V. Strassen. A fast Monte-Carlo test for primality. *SIAM Journal on Computing*, vol. 6, pp. 84–85, 1977.