Efficient polynomial time algorithms computing industrial-strength primitive roots

Jacques Dubrois, Jean-Guillaume Dumas

Efficient polynomial time algorithms computing industrial-strength primitive roots

Jacques Dubrois* and Jean-Guillaume Dumas*

September 14, 2004

Abstract

E. Bach, following an idea of T. Itoh, has shown how to build a small set of numbers modulo a prime p such that at least one element of this set is a generator of $\mathbb{Z}/p\mathbb{Z}$[2, 16]. E. Bach suggests also that at least half of his set should be generators. We show here that a slight variant of this set can indeed be made to contain a ratio of primitive roots as close to 1 as necessary. We thus derive several algorithms computing primitive roots correct with very high probability in polynomial time. In particular we present an $O(\sqrt{\frac{1}{\epsilon}}log^2(p) + log^4(p))$ algorithm providing primitive roots with probability of correctness greater than $1 - \epsilon$ and several $O(log^\alpha(p))$, $4 \leq \alpha \leq 4.959$ algorithms computing "Industrial-strength" primitive roots with probabilities e.g. greater than the probability of "hardware malfunctions".

1 Introduction

Primitive roots are generators of multiplicative group of the invertibles of a finite field. We focus in this paper only on prime finite fields, but actually the proposed algorithms can work over extension fields or any other multiplicative groups.

Primitive roots are of intrinsic use e.g. for secret key exchange [7] or pseudo random generators [4]. The classical method of generation of such generators is by trial, test and error. Indeed within a prime field with p elements they are quite numerous ($\phi(\phi(p)) = \phi(p-1)$ among $p-1$ invertibles are generators, see e.g. [5] for more details). The problem resides in the test to decide whether a number g is a generator or not. The first idea is to test every g^i for $i = 1..p-1$ looking for matches. Unfortunately this is exponential in the size of p. The classical acceleration is then to factor $p-1$ and test whether one of the $g^{\frac{p-1}{q}}$ is

*Université de Grenoble, laboratoire de modélisation et calcul, LMC-IMAG BP 53 X, 51 avenue des mathématiques, 38041 Grenoble, France. {Jacques.Dubrois, Jean-Guillaume.Dumas}@imag.fr .

1 for q a divisor of $p-1$. If this is the case then g is obviously not a generator. On the contrary, one has proved that the only possible order of g is $p-1$. Unfortunately again, factorization is still not a polynomial time process. Therefore no polynomial time algorithm computing primitive roots is known.

However, there exists polynomial time methods isolating a polynomial size set of numbers containing at least one primitive root. Shoup's [23] algorithm is such a method. Elliot and Murata [8] also gave polynomial lower bounds on the least primitive root modulo p. One can also generate elements with exponentially large order even though not being primitive roots [13]. Our method is in between those two approaches.

As reported by Bach [2], Itoh's breakthrough was to use only a partial factorization of $p-1$ to produce primitive roots with high probability [16]. Bach then used this idea of partial factorization to give the actual smallest known set deterministically containing one primitive root[2]. Moreover, he suggested that his set contained at least half primitive roots.

In this paper, we propose to use a combination of Itoh's and Bach's algorithms producing a polynomial time algorithm generating primitive roots with a very small probability of failure. Such generated numbers will be denoted by "Industrial-strength" primitive roots. We also have a guaranteed lower bound on the order of the produced elements. We mainly deal with the analysis of the actual ratio of primitive roots within a variant of Bach's full set. As this ratio is close to 1, selecting a random element within this set produces a fast and effective method computing primitive roots. The probability of failure, i.e. of selecting a non primitive root can therefore be made arbitrarily low.

We present in section 2 our algorithm and the main theorem counting the ratio of primitive roots. As this ratio and provable bounds on it highly depends on ω the number of prime factors of a given composite, section 3 shows some refinement of known bounds for ω. Then practical implementation details and effective ratios are discussed section 4. Section 5 presents following a tentative analysis of the composite case. We conclude section 6 with applications of primitive root generation, accelerated by our probabilistic method. Among this applications are Diffie-Hellman key exchange, ElGamal cryptosystem, Blum-Micali pseudo random bit generation, and a new probabilistic primality test based on Lucas' deterministic procedure.

2 The variant of Itoh/Bach's algorithm

We now present our variant of Itoh/Bach's algorithm. The salient features of our approach when compared to Bach's are that:

1. We partially factor, but *with known lower bound on the remaining factors to be found.*
2. *We do not require the primality* of the chosen elements.
3. *We consider the whole set of candidates* instead of only the first ones.

Now, when compared to Itoh's method, we use a deterministic process producing a number with a very high order and which has a high probability of being primitive. On the contrary, Itoh selects a random element but uses a polynomial process to prove that this number is a primitive root with high probability [16]. The difference here is that with our version we use low order terms to build higher order elements whereas Itoh discards the randomly chosen candidates and restarts all over at each failure.

As we show next, with this requirements we are able to prove very high probabilities to find primitive roots.

We now prove that this algorithm is correct and give its running complexity. In order to prove the correctness, we prove the following theorem. We have to admit that Itoh, independently and very differently, proves the same within his [16, Theorem 1].

Theorem 1. *The ratio of primitive roots within the returned values of Algorithm 1 is* $\frac{\phi(Q)}{Q-1}$.

Proof. We let $p-1 = kQ$. a as in the algorithm has order k (see e.g. [2]). First, let $B = \{b^k, b \in Z/pZ^*$ and b^k has order $Q\}$. We have $|B| = \phi(Q)$. Indeed the distinct elements of B are exactly all the elements in Z/pZ^* of order Q: if y is of order Q, then there exists some generator g for which $y = g^{uk}$, b is thus g^u.

We now need to count how many b are such that b^k is of order Q, or equivalently how many b are such that ab^k is a primitive root, for a of order k ?

Well, we let $g = ab^k$ be a primitive root. Then, $\exists v$ so that $a = g^{vQ}$ and $b = g^{t_1}$. From there we now that $1 \equiv vQ + t_1 k\ [p-1]$. Now we set i so that $0 \leq t_0 = t_1 - iQ \leq Q$; this gives that $\forall j \in \{0..k-1\}$, $t_0 + jQ < P - 1$ and $a.(g^{t_0+jQ})^k \equiv g$, since $k(t_0 + jQ) = k(t_1 + (j-i)Q) \equiv kt_1 \equiv 1 - vQ\ [p-1]$. We thus have k possible b_i for which $ab_i^k = g$. This proves that k distinct b_i produce the same primitive root. Moreover, two distinct primitive roots can of course only be produced by distinct b_i. This proves that $k\phi(Q)$ distinct b_i produce a primitive root.

Algorithm 1: Probabilistic Primitive Root

Input: A prime $p \geq 3$
Input: A failure probability $0 < \epsilon < 1$
Output: A number, primitive root with probability greater than $1 - \varepsilon$.
begin
  Compute B such that $(1 + \frac{2}{p-1})(1 - \frac{1}{B})^{\log_B \frac{p-1}{2}} == 1 - \varepsilon$.
  Partially factor $p - 1$, so that $p - 1 = 2^{e_1} p_2^{e_2} .... p_h^{e_h} Q$ where $p_i < B$ and Q is free of prime $< B$ (*for that use e.g.* $2\sqrt{B}$ *loops of Pollard's rho method*).
  foreach $1 \leq i \leq h$ **do**
    By trial and error, randomly choose α_i verifying: $\alpha_i^{\frac{p-1}{p_i}} \not\equiv 1 \; mod \; p$.
  Set $a = \prod_{i=1}^{h} \alpha_i^{\frac{p-1}{p_i^{e_i}}} \; mod \; p$.
  if *Factorization is complete* **then**
    Set Probability of correctness to 1.
    return a.
  else
    Refine Probability of correctness to $(1 + \frac{1}{Q-1})(1 - \frac{1}{B})^{\log_B Q}$.
    By trial and error, randomly choose b verifying: $b^{\frac{p-1}{Q}} \not\equiv 1$.
    return $g = ab^{\frac{p-1}{Q}}$.
end

There remains to count from how many b_i we can select. Well, this is $p - 1$ minus the number of b_i for which $b_i^k \equiv 1$. As before, 1 has exactly k distinct pre images. We thus conclude for the ratio to be $\frac{k\phi(Q)}{p-1-k} = \frac{\phi(Q)}{Q-1}$. □

Corollary 2. *Algorithm 1 is correct.*

Proof. Theorem 1 shows that the probability to get a primitive root is exactly $\frac{\phi(Q)}{Q-1}$. We thus only need to show that $\frac{\phi(Q)}{Q-1} > 1 - \varepsilon$.
Let $Q = \prod_{i=1}^{\omega(Q)} {q_i}^{f_i}$ where $\omega(Q)$ is the number of distinct prime factors of Q. Then
$\phi(Q) = \prod_{i=1}^{\omega(Q)} \phi({q_i}^{f_i}) = Q \prod_{i=1}^{\omega(Q)} (1 - \frac{1}{q_i})$.
Thus $\frac{\phi(Q)}{Q-1} = (1 + \frac{1}{Q-1}) \prod_{i=1}^{\omega(Q)} (1 - \frac{1}{q_i})$. Now, since any factor of Q is bigger than

B, we have:

$$\prod_{i=1}^{\omega(Q)}(1-\frac{1}{q_i}) > \prod_{i=1}^{\omega(Q)}(1-\frac{1}{B}) = (1-\frac{1}{B})^{\omega(Q)}.$$

To finish, we minor $\omega(Q)$ by $\log_B(Q)$. This gives the probability refinement. Since Q is not known at the beginning, one can minor it there by $\frac{p-1}{2}$ since $p-1$ must be even whenever $p \geq 3$. □

Remark 3. *Of course, one can dynamically refine B as more and more small factors of $p-1$ are known. Indeed Q starts with value $\frac{p-1}{2}$ (and can immediately be set to $\frac{p-1}{2^{e_1}}$) but is also reduced each time a new factor is known. The end refinement would then become unnecessary. Algorithm 1 is given instead for the sake of simplicity.*

Remark 4. *Instead of looking for distinct α_i for the different p_i, one can take one α_i at random. Then, the idea is to check for its order with respects to all of the p_i and not only one. Suppose that $\alpha^{\frac{p-1}{p_1}} \not\equiv 1$ and $\alpha^{\frac{p-1}{p_2}} \not\equiv 1$ Then α can serve for both primes and a be set to $\alpha^{\frac{p-1}{p_1^{e_1}p_2^{e_2}}}$. The search then continues as in algorithm 1.*

Theorem 5. *Algorithm 1 has a worst case complexity of*

$$O\left(\frac{1}{\varepsilon}\log^2(p) + \log^4(p)\log(\log(p))\right)$$

and, when Pollard's rho algorithm is used, an average running time of

$$O\left(\sqrt{\frac{1}{\varepsilon}}\log^2(p) + \log^3(p)\log(\log(p))\right)$$ [1].

Proof. For the computation of B, we use a Newton-Raphson's approximation.

The second step depends on the factorization method. Both complexities here are given by the application of Pollard's rho algorithm. Indeed Pollard's rho would require at worst $L = 2\lceil B \rceil$ loops and $L \approx 3.5482\lceil\sqrt{B}\rceil$ on the average thanks to the birthday paradox [20]. Now each loop of Pollard's rho is a squaring and a gcd, both of complexity $O(\log^2 p)$. We conclude by the fact that $(1+\frac{2}{p-1})(1-\frac{1}{B})^{\log_B \frac{p-1}{2}} = 1-\varepsilon$ so that $B \leq \frac{1}{\varepsilon}$.

[1] Using fast integer arithmetic [12, Corollary 11.10] this can become

$$O\left(\sqrt{\frac{1}{\varepsilon}}\log(p)\log^2(\log(p))\log(\log(\log(p))) + \log^2(p)\log^2(\log(p))\log(\log(\log(p)))\right)$$

For the remaining steps, there is at worst $\log p$ distinct factors, thus $\log p$ distinct a_i, but only $\log\log p$ on the average [15, Theorem 430]. Each one requires an exponentiation which can be performed with $O(\log^3 p)$ operations using fast binary modular exponentiation. Now, to get a correct a_i, at most $O(\log\log p)$ trials should be necessary (see e.g. [24, Theorem 6.18]). However, by an argument similar to that of theorem 1, less than $1 - \frac{1}{p_i}$ of the α_i are such that $\alpha_i^{\frac{p-1}{p_i}} \equiv 1$. This gives an average number of trials of $1 + \frac{1}{p_i}$, which is bounded by a cnstant. This gives $\log \times \log^3 \times \log\log$ in the worst case (distinct factors $\times$ exponentiation $\times$ number of trials) and only $\log\log \times \log^3 \times 2$ on the average. □

Of course, the only problem with this algorithm is that it is not polynomial. Indeed the partial factorization up to factors of any given size is still exponential. This gives the non polynomial factor $\sqrt{\frac{1}{\varepsilon}}$. Other factoring algorithms with better complexity could also be used, provided they can guaranty a bound on the unfound factors.

Anyway, we now try to reduce the actual values of B in order to produce efficient algorithms. The first idea, of next section, is to improve the bounds. Then, the trick of the following section is to factor only to a logarithmic size of factors in order to guaranty a polynomial time algorithm.

3 About the number of prime divisors

In the previous section, we have seen that the probability to get a primitive root out of our algorithm is greater than $\left(1-\frac{1}{B}\right)^{\omega(Q)}$ for Q the remaining unfactored part with no divisors less than B. The running time of the algorithm, and in particular its non-polynomial behaviour depnds on B and on ω. The problem is that ω is in general quite small. The bound we used in the preceding section, $\log_B(p-1)$, is then much too large. In this section, we thus provide tighter probability estimates for some small B and large Q.

Theorem 6. *Let $B \in \mathbb{N}$, $Q \in \mathbb{N}$ such that no prime lower than B divides Q then:*

$$\omega(Q) \leq \log_B(Q) \qquad \forall B \geq 2 \tag{1}$$

$$\omega(Q) \leq \frac{1.0956448}{\log_B(ln(Q))}\log_B(Q) \qquad \forall B \geq 2^{10} \tag{2}$$

$$\omega(Q) \leq \frac{1.0808280}{\log_B(ln(Q))}\log_B(Q) \qquad \forall B \geq 2^{15} \tag{3}$$

$$\omega(Q) \leq \frac{1.0561364}{\log_B(ln(Q))}\log_B(Q) \qquad \forall B \geq 2^{20} \tag{4}$$

Proof. Of course, (1) is a large upper bound on the number of divisors of Q and therefore a bound on the number of prime divisors. Now for the other bounds, we refine Robin's bound on ω [22, Theorem 11]: which is $\omega(n) \leq \frac{1.3841}{ln(ln(n))}ln(n)$. Let $N_k = \prod_{i=1}^{k} p_i$ where p_i is the i-th prime. Now, we let k be such that $\frac{N_k}{N_{\pi(B)}} \leq Q < \frac{N_{k+1}}{N_{\pi(B)}}$. Then $\omega(Q) \leq \omega\left(\frac{N_k}{N_{\pi(B)}}\right) = k - \pi(B)$ since no prime less than B can divide Q. We then combine this with the fact that $X \hookrightarrow \frac{ln(X)}{X}$ is decreasing for $X > e$ to get: $\omega(Q) \leq \frac{F(k,B)}{\log_B(ln(Q))}\log_B(Q)$ where $F(k,B) = (k-\pi(B)).\frac{\log\left(\log\left(\frac{N_k}{N_{\pi(B)}}\right)\right)}{\log\left(\frac{N_k}{N_{\pi(B)}}\right)}$. We then replace both N_k in $F(k,B)$ using e.g. classical bounds on $\theta(p_k) = ln(N_k)$ [22, Theorems 7 & 8]:

$$\theta(p_k) \geq k(ln(k) + ln(ln(k)) - 1 + ln(ln(k))/ln(k) - 2.1454/ln(k)) \tag{5}$$

$$\theta(p_k) \leq k(ln(k) + ln(ln(k)) - 1 + ln(ln(k))/ln(k) - 1.9185/ln(k)) \tag{6}$$

We therefore obtain a function $\widetilde{F}(k,B)$ explicit in k and B. The values given in the theorem are the numerically computed maximal values of $\widetilde{F}(k,B)$ as a function in k for $B \in \{2^{10}, 2^{15}, 2^{20}\}$. The claim for the greater values of B then follows from the fact that $\widetilde{F}(k,B)$ is decreasing in B. □

It is noticeable that the last estimates are more interesting than $\log_B(Q)$ only when $B^{\widetilde{F}(k,B)} < ln(Q)$. Those estimates are then only useful for very large Q (e.g. more than 10^5 bits for $B = 2^{15}$).

4 Polynomial time heuristics

4.1 Efficient generation of industrial-strength primitive roots

Despite the small theoretical improvements of section 3, our lower bound for the actual probability of correctness is much too low. For that reason, we propose another algorithm with an attainable number of loops for the partial factorization. Therefore, the algorithm is efficient and we provide experimental data showing that it also has a very good behavior with respect to the probabilities.

Algorithm 2: Polynomial-time Generation of Primitive Root

Input: A prime $p \geq 3$
Output: A number, primitive root of p with high probability.
begin
  Apply Algorithm 1 with $B \leq \log^2(p)\log^2(\log(p))$.
end

Lemma 7. *With Pollard's rho factoring, Algorithm 2 has an*

$$O\left(\log^3(p)\log(\log(p))\right)$$

average bit complexity.

Proof. The proof is trivial in view of that of theorem 5. Just replace B by $\log^2(p)\log^2(\log(p))$ and use $L = \sqrt{B}$. □

In practice, *L should be chosen not higher than a million !* Indeed, the following experimental data (see figures 1 and 2) shows that a probability of $1 - 2^{-40}$ is easily guaranteed. We choose Q with known factorization and compute $\frac{\phi(Q)}{Q-1}$. Figure 1 shows the very good behavior of our algorithm when the number of distinct factors is small. Now, figure 2 seems to show that no probability less than $1 - 2^{-40}$ is possible even with L as small as 2^{20}. This fact will be explained in the following.

4.2 Industrial-strength primitive roots with best polynomial complexity

Provided that one is ready to accept a fixed probability, further theoretical improvements on the asymptotic complexity can be made. Indeed, Don Knuth said *"For the probability less than $(\frac{1}{4})^{25}$ that such a 25-times-in-row procedures gives the wrong information about n. This is less than one chance in a quadrillion; even if we certified a billion different primes with such a procedure, the expected*

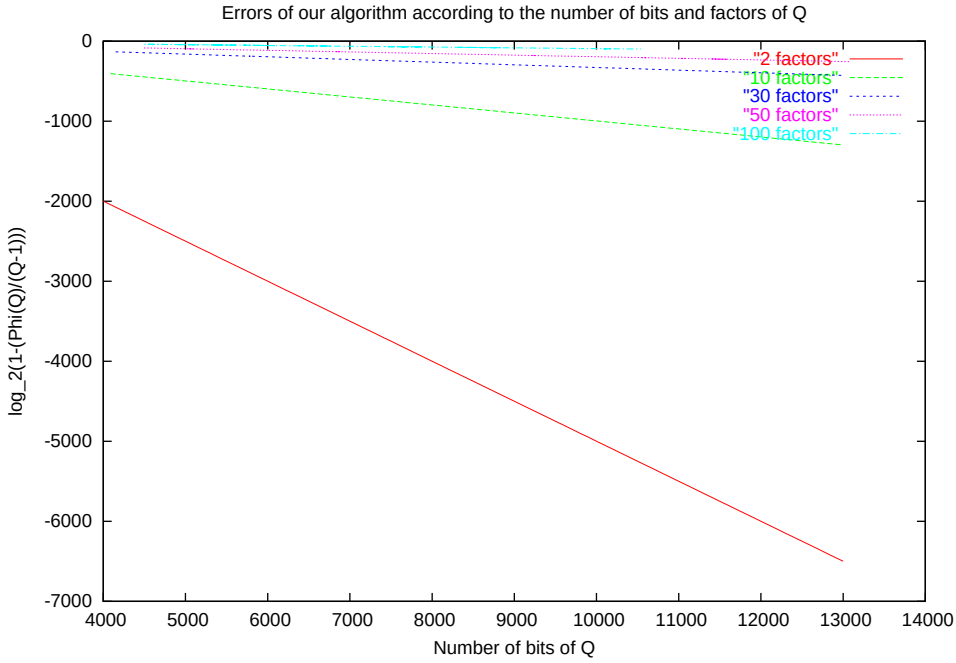

Figure 1: Actual probability of failure of Algorithm 2 with $L = 2^{20}$

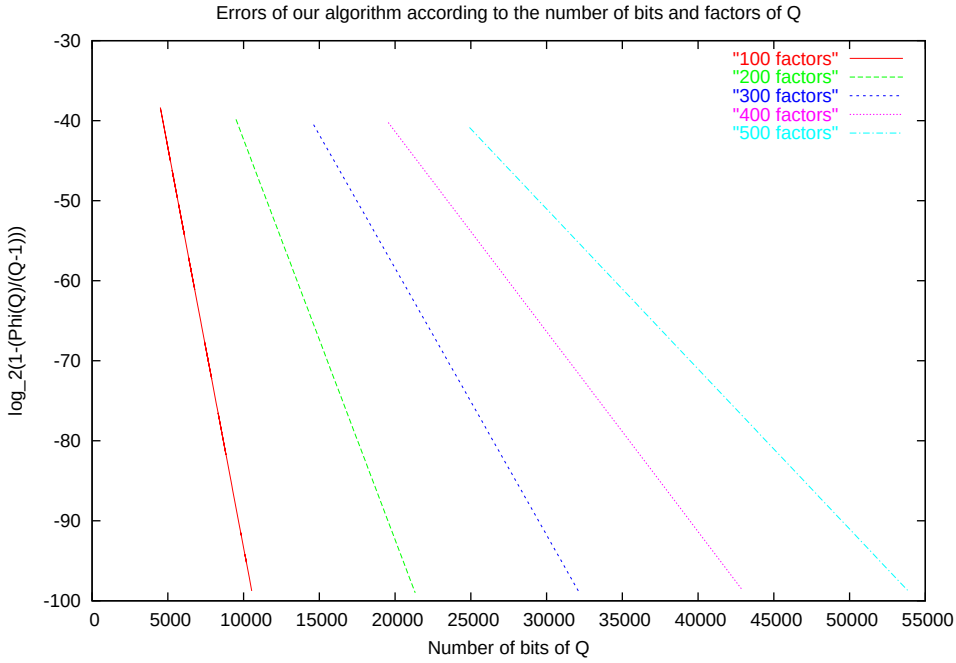

Figure 2: Actual probability of failure for Q with more distinct factors

number of mistakes would be less than 1/1000000. It's much more likely that our computer has dropped a bit in its calculations, due to hardware malfunctions or cosmic radiations, than that algorithm P has repeatedly guessed wrong." [17]. We thus provide a version of our algorithm guarantying that the probability of incorrect answer is lower than 2^{-50}.

Algorithm 3: Industrial-strength Primitive Root

Input: A prime $p \geq 3$
Output: A number, primitive root of p with probability higher than $1 - 2^{-50}$ (resp. $1 - 2^{-40}$).
begin
  if $p < 2^{512}$ *(resp.* $p < 2^{30}$*)* **then**
    Factor $p - 1$ completely and produce a deterministic primitive root.
  else
    Apply Algorithm 1 with $B = log^{5.918016}p$ (resp. $B = log^{5.909920}p$).
end

Theorem 8. *With Pollard's rho factoring, Algorithm 3 is correct and has an average* $O(\log^{4.959008} p)$ *(resp.* $O(\log^{4.954960} p)$*) asymptotic bit complexity*[2]*.*

Proof. First, we remark that 512-bits numbers are nowadays factorizable. So, whatever running time is needed to compute the factorization of numbers lower than 2^{512}, this is asymptotically a constant ! Still, if this is too hard to swallow, we also give the algorithm with a slightly smaller probability of correctness and a trivial factorization of 30-bits numbers. Now for $p > 2^{512}$ and $B = \log^{\alpha}(p)$, we just remark that $(1+\frac{2}{p-1})(1-\frac{1}{B})^{\log_B \frac{p-1}{2}}$ is increasing, so that it is bounded by its first value (for which $p = 2^{512}$). Now numerical approximation of α so that the latter is $1 - 2^{-50}$ gives $\alpha \approx 5.918$. The complexity exponent follows as it is $2 + \frac{\alpha}{2}$.
Now the exact same arguments apply for a probability $1 - 2^{-40}$, and then with very easy factorization: for a composite number of 30 bits, there exist a factor with less than 15 bits. Pollard's rho algorithm would then only require 180 loops on the average ! □

With this version, we see once more that a probability of $1-2^{-40}$ can indeed always be guaranteed. In other words, this proves that our algorithm is able to very efficiently produce industrial-strength primitive roots.

This is for instance illustrated when comparing our algorithm, implemented in C++ with GMP [14], to existing software (e.g. Pari-GP[3]) on an Intel PIV 2.4GHz. Such comparison is shown on figure 3.

[2]The worst case exponents then being 7.918016 (resp. 7.909920).
[3]http://pari.math.u-bordeaux.fr

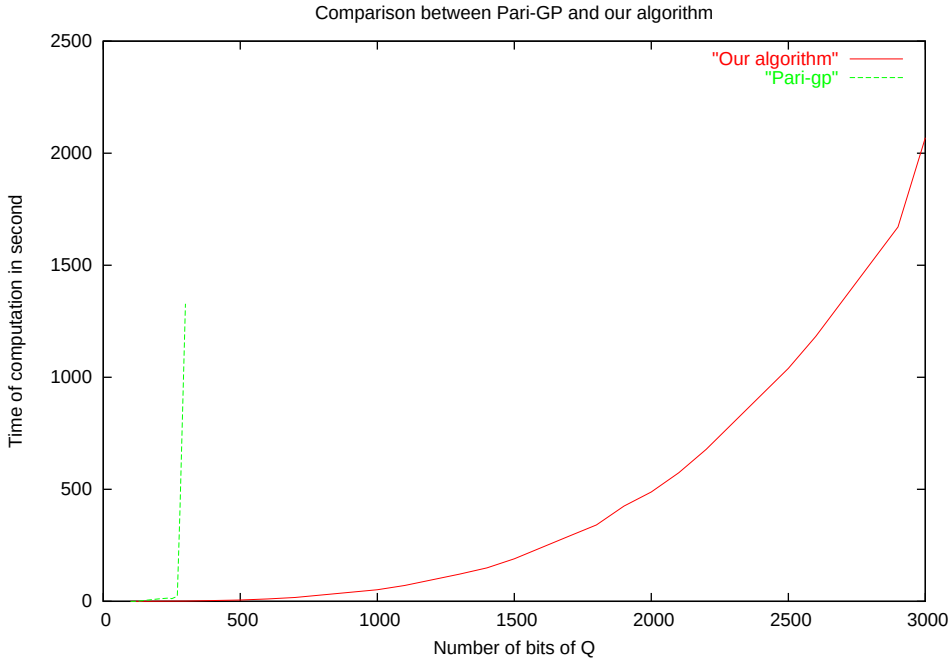

Figure 3: Comparing Pari-GP's generation of primitive roots

Of course, the comparison is not fair as other softwares are always factoring $p-1$ completely. Still we can see the huge progress in primitive root generation that our algorithm has enabled.

5 Tentative analysis of the algorithm for composite numbers

In this section we propose some analysis of the behavior of the algorithm for composite numbers. Indeed, our algorithm can also be used to produce high, if not maximal, order element modulo some composite number. This analysis is also used section 6.2 for the probabilistic primality test. It is well known that there exists primitive roots for every number of the form 2, 4, p^k or $2p^k$ with p an odd prime. On the other hand, Euler's theorem states that every invertible a within $\mathbb{Z}/p\mathbb{Z}^*$ verifies $a^{\varphi(n)} \equiv 1[n]$. Thus, for composite numbers n not possessing primitive roots, $\varphi(n)$ is not a possible order of an invertible. We therefore use Carmichael's lambda function, the maximal order of an invertible, defined e.g. in [17, 9, 3]:

Definition 9. *$\lambda(m)$ is the maximal order of an invertible element in the multiplicative group $(\mathbb{Z}/p\mathbb{Z}^*, \times)$.*

Of course, λ and φ coincide for 2, 4, p^k and $2p^k$, for p and odd prime. Then $\lambda(2^e) = 2^{e-2}$ for $e \geq 3$. Now, for the other cases, since $\varphi\left(\prod p_i^{k_i}\right) = \prod (p_i - 1)p_i^{k_i - 1}$ for distinct primes p_i, we obtain the similar following formula for λ:

$$\lambda\left(\prod p_i^{k_i}\right) = lcm\{\lambda(p_i^{k_i})\}$$

Eventually, we also obtain the following corollary of Euler's theorem:

Corollary 10. *Every invertible a within $\mathbb{Z}/p\mathbb{Z}^*$ verifies $a^{\lambda(n)} \equiv 1[n]$.*

Proof. Let $n = \prod p_i^{e_i}$, for distinct primes p_i. Then $\varphi(p_i^{e_i})$ divides $\lambda(n)$. This together with Euler's theorem shows that $a^{\lambda(n)} \equiv 1[p_i^{e_i}]$. The Chinese theorem thus implies that the latter is also true modulo the product of the $p_i^{e_i}$. □

This corollary shows that the order of any invertible must divide $\lambda(n)$. Now for n prime, the number of invertibles having order $d|p-1$ is exactly $\varphi(d)$ so that $\sum_{d|k} \varphi(d) = k$ for $k|p-1$. We have the following analogous for n a composite number:

Proposition 11. *The number of invertibles having order $d|\lambda(n)$ is*

$$\sum_{S_d} \prod_{j=1}^{\omega} \varphi(d_j)$$

for $n = p_1^{e_1} \dots p_\omega^{e_\omega}$ and $S_d = \{(d_1, \dots, d_\omega)$ s.t. $d_j|\varphi(p_j^{e_j})$ and $lcm\{d_j\} = d\}$.

Proof. By the Chinese theorem, an element has order d if and only if the lcm of its orders modulo the $p_j^{e_j}$ is d. Then there are exactly $\varphi(d_j)$ elements of order d_j modulo $p_j^{e_j}$. □

Let us have a look of this behavior on an example: let $n = 45$ so that $\varphi(45) = 6 \times 4 = 24$ and $\lambda(45) = 12$. We thus know that any order modulo 9 divides $\varphi(9) = 6$ and that any order modulo 5 divides $\varphi(5) = 4$. This gives the different orders of the 24 invertibles shown on table 1.

order	modulo 9	modulo 5	# of elements of that order modulo 45
1	1	1	**1**
	1	2	1
	2	1	1
	2	2	1
2			**3**
3	3	1	$\varphi(3) \times \varphi(1) =$ **2**
	1	4	$\varphi(1) \times \varphi(4) = 2$
	2	4	$\varphi(2) \times \varphi(4) = 2$
4			**4**
	6	1	$\varphi(6) \times \varphi(1) = 2$
	3	2	$\varphi(3) \times \varphi(2) = 2$
	6	2	$\varphi(6) \times \varphi(2) = 2$
6			**6**
	3	4	$\varphi(3) \times \varphi(4) = 4$
	6	4	$\varphi(6) \times \varphi(4) = 4$
12			**8**

Table 1: Elements of a given order modulo 45

It would be highly desirable to have tight bounds on those number of elements of a given order. Moreover, these bounds should be easily computable (e.g. not requiring some factorization !). We propose the following:

Proposition 12. *For d dividing $\lambda(n)$, the number of invertible elements of order d is at least $\varphi(d)$ and the number of invertibles of order less than or equal to d is at least d.*

Proof. At least one ω-uple $(d_1, \ldots, d_\omega)$ is possible, one for which the d_i are two by two co-prime. Then $\prod_{j=1}^{\omega} \varphi(d_j) = \varphi(d)$ which proves the first claim. The second is only the sum of the $\varphi(h)$ for $h|d$. □

This, together with the following experimental result should give a basis for the complete analysis of the composite case:

Conjecture 13. *For n odd, the number of elements of order $\lambda(n)$ is larger than $\varphi(\varphi(n))$.*

Ideas. The first idea is to use the previous proposition. Direct simplification shows that we can construct at least

$$\prod_{j=1}^{\omega} \varphi(\varphi(p_i^{e_i}))$$

distinct elements having order $\lambda(n)$. This is not sufficient. The second idea is that every maximal order element produces

$$\varphi(\lambda(n))$$

other maximal elements in its orbit: consider an element of maximal order a. Then if k is co-prime to λ then the order i of a^k must be λ. Indeed, if $a^{ki} \equiv 1 \mod n$, then $ku + \lambda v = 1$ implies that $a^{kiu} \equiv a^i \equiv 1$. As the order of a is λ and no larger order is possible, then $i == \lambda$. This is still insufficient.

What happens is that each of the distinct elements produced by the first idea generates its own orbit. The orbits are not distinct but neither are they equal. The full proof would be to augment the orbits with a converging process ... □

The bound is tight: $\varphi(\varphi(15)) = 4$ and only 2, 7, 8 and 13 have order $\lambda(15) = 4$. Now, this last result shows that actually quite a lot of elements are of maximal order modulo n. Using this fact, a modification of algorithm 1 can then produce with high probability an element of maximal order even though n is composite.

6 Applications

Of course, our generation can be applied to any application requiring the use of primitive roots. In this section we show the speed of our method compared to generation of primes with known factorization and propose a generalization of Miller-Rabin probabilistic test.

6.1 Faster pseudo random generators construction or key exchange

The use of a generator and a big prime is the core of many cryptographic protocols. Among them are Blum-Micali pseudo-random generators [4], Diffie-Hellman key exchange [7], etc.
In this section we just compare the generation of primes with known factorization [1], so that primitive roots of primes with any given size are computable. The idea in [4] is to iteratively and randomly build primes so that the factorizations of $p_i - 1$ are known. For cryptanalysis reasons their original method selects the primes and primitive roots bit by bit and is therefore quite slow. On figure 4 we then present also a third way, which is to generate the prime with known factorization as in [1], but then to generate the primitive root deterministically with our algorithm (since the factorization of $p-1$ is known). We compare this method with the following full-probabilistic way:

1. By trial and error generate a probable prime (e.g. a prime passing several Miller-Rabin tests [19]).
2. Generate a probable primitive root by Algorithm 2.

We see on figure 4 that our method is faster and allows for the use of bigger primes/generators.

6.2 Probabilistic version of Lucas primality test

The deterministic primality test of Lucas is actually the existence of primitive roots:

Theorem 14 (Lucas). *Let $p > 0$. If one can find an $a > 0$ such that $a^{p-1} \equiv 1 \, mod \, p$ and $a^{\frac{p-1}{q}} \not\equiv 1 \, mod \, p$, as soon as q divides $p-1$, then p is prime.*

We propose here as a probabilistic primality test to try to build a primitive root. If one succeeds then the number is prime with high probability else it is either proven composite or composite with a high probability.

Now for the complexity, we do not pretend to challenge Miller-Rabin test for speed ! Well, one often needs to perform several Miller-Rabin tests with distinct witnesses, so that the probability of being prime increases. Our idea is the following: since one tests several witnesses, why not use them as factors of our probable primitive root ! This idea can then be viewed as a generalization of Miller-Rabin: we not only test for orders of the form $\frac{n-1}{2^e}$ but also for each order

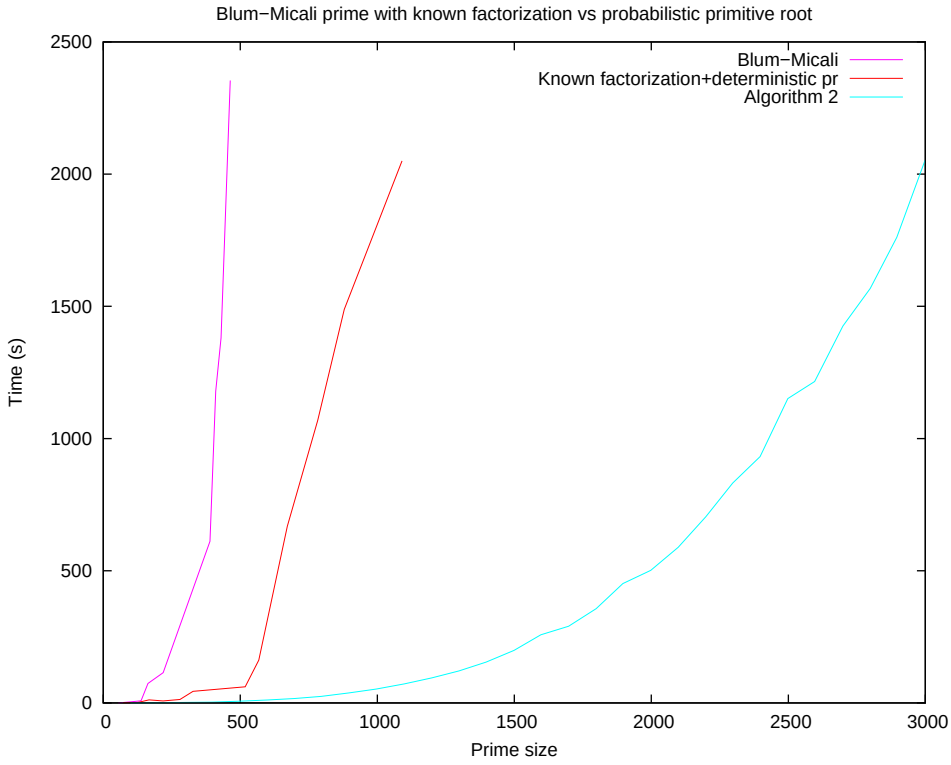

Figure 4: Comparing primes with known factorization to Industrial-strength primitive roots

of the form $\frac{n-1}{q^e}$ where q is a small prime factor of $n-1$. The effective complexity (save maybe from the partial factorization) will not suffer and the probability can jump as soon as an element with very high order is generated. The algorithm is then a slight modification of algorithm 1, where we let $F(B,Q) = 1-(1+\frac{1}{Q-1})(1-\frac{1}{B})^{\log_B Q}$:

Algorithm 4: Probabilistic Lucas primality test

Input: $n \geq 3$, odd.
Input: A failure probability $0 < \epsilon < 1$.
Output: **Whether** n is prime and a certificate of primality,
Output: **or** n is composite and a factor (or just a Fermat witness),
Output: **or** n is prime with probability of error less than ϵ,
Output: **or** n is composite with probability of error less than ϵ.
begin
- Set $P = 1$, $a = 1$, $Q = n - 1$ and $q = 2$.

while $Q > n^{\frac{2}{3}}$ **do**
- Randomly choose $\alpha \mod n$.
- **if** $gcd(\alpha, n) \neq 1$ ***or*** $\alpha^{n-1} \not\equiv 1[n]$ ***or*** $gcd(\alpha^{\frac{n-1}{q}} - 1, n) \notin \{1; n\}$ ***or*** $(q == 2$ *and* n *is not a strong pseudoprime to the base* $\alpha)$ **then**
  - **return** n *is composite.*
- **else if** $\alpha^{\frac{n-1}{q}} \equiv 1$ *mod* n **then**
  - Set $P = P/q$.
  - **if** $P \leq \epsilon$ **then**
    - **return** n *is probably composite with error less than* P.
- **else**
  - **-** Set e to the greatest power of q dividing Q.
  - **-** Set $Q = Q/q^e$.
  - **-** Set $a = a \times \alpha^{\frac{n-1}{q^e}}$.
  - **-** Set $k = k \cup \{q^e\}$.
  - **-** Refine B such that $F(B, Q) == 4\epsilon$.
  - **-** Find a new prime factor q of Q with $q < B$, otherwise set $q = Q$.

if *Every* q *was prime* **then**
- **return** n *is prime and* (a, k) *is a certificate.*

else
- **return** n *is probably prime with error less than* $F(B, q)$.

end

Remark 15. *The exponentiations by* $\frac{n-1}{q}$ *can in practice by factorized in a "Lucas-tree" [21, 6].*

Remark 16. *Algorithm 4 is correct for the primes and most of the composite numbers.*

Proof. Correctness for prime numbers is the correctness of the pseudo primitive root generation.
Now for composite numbers: the idea is that first of all, only Carmichael num-

bers will be able to pass the pseudo prime test several times. The ϵ times 4 then follows since at least one α passed the strong pseudoprime test. This reduces the possible Carmichael numbers able to pass our test. Then, for most of the Carmichael numbers, $\lambda(n)$ divides $n-1$ but, moreover, $\lambda(n)$ also divides $\frac{n-1}{q}$ for some q, factor of $n-1$. Therefore, $\alpha^{\frac{n-1}{q}}$ will always be one. If n is prime on the contrary, only $\frac{1}{q}$ elements will have order a multiple of q.

Now for the $n^{\frac{2}{3}}$ in the loop. The argument is the same as for the Pocklington theorem [6, Theorem 4.1.4] and the Brillhart, Lehmer and Selfridge theorem [6, Theorem 4.1.5]: let $n-1 = kQ$ and let p be a prime factor of n. The algorithm has found an a verifying $a^{n-1} \equiv 1 \mod n$. Hence, the order of $a^Q \mod p$ is a divisor of $\frac{n-1}{Q} = k$. Now, since $gcd(a^{\frac{n-1}{q}} - 1, n) = 1$ for each prime q dividing k, this order is not a proper divisor of k, so is equal to k. Hence, k must be a divisor of $p-1 = \varphi(p)$. We conclude that each prime factor of n must exceed k. From this, Pocklington's theorem states that if k is greater than $\sqrt{n}$, n is prime. And then, Brillhart-Lehmer-Selfridge theorem states that if k is in between $n^{\frac{1}{3}}$ and $n^{\frac{1}{2}}$ then n must be prime or composite with exactly two prime factors [6, Theorem 4.1.5]. But n has escaped our previous tests only if n is a Carmichael number. Fortunately, Carmichael numbers must have at least 3 factors [18, Proposition V.1.3]. Now, whenever Q is below $n^{\frac{2}{3}}$, k exceeds $n^{\frac{1}{3}}$ and then n must be prime otherwise n would have more than 3 factors each of those being greater than $n^{\frac{1}{3}}$. □

Here is an example of Carmichael number, 1729. $1728 = 2^6 3^3$, where $\lambda(1729) = 2^2 3^2$. Then $\frac{n-1}{q}$ is either 864 or 576 both of which are divisible by $36 = \lambda(1729)$. Therefore, our test will detect 1729 to be probably composite with any probability of correctness. Figure 5 shows that this algorithm is highly competitive with repeated applications of GMP's strong pseudo prime test (i.e. with the same estimated probability of correctness). Depending on the success of the partial factorization, our test can even be faster (timing presented on figure 5 are the mean time between 4 disctinct runs).

Haplessly, some Carmichael numbers will still pass our test. The following results, sharpening [10, lemma 1], explains why:

Theorem 17. *Let $n = p_1^{e_1} \dots p_\omega^{e_\omega}$. Let q be a prime divisor of $\varphi(n)$, and $(f_1, \dots, f_\omega)$ be the maximal values for which q^{f_i} divides $\varphi(p_i^{e_i})$. There are*

$$\varphi(n)\left(1 - \frac{1}{q^{\sum f_i}}\right)$$

invertible elements of order divisible by q (i.e. for which $\alpha^{\frac{\lambda(n)}{q}} \not\equiv 1 \mod n$).

Proof. By the Chinese remainder theorem, one can consider the moduli by $p_i^{e_i}$ separately. Suppose, without loss of generality, that $p_1^{e_1}$ is such that $f_1 > 0$. Otherwise all the f_i are 0 and the theorem is still correct. Consider a generator g of the invertibles modulo $p_1^{e_1}$. An element has q in its order if and only if its index with respect to g contains q^{f_1}. There are exactly $1 - \frac{1}{q^{f_1}}$ such elements among

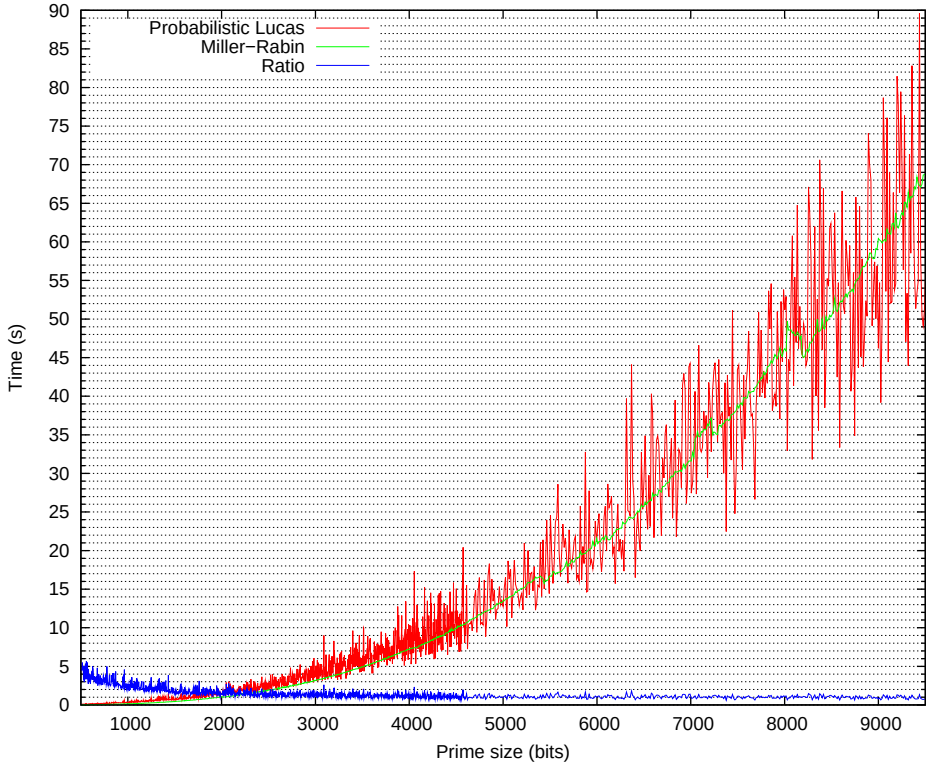

Figure 5: Probabilistic Lucas vs GMP's Miller-Rabin for primes with probability $< 10^{-6}$, on a PIV 2.4GHz

the elements of $\mathbb{Z}/p_1^{e_1}\mathbb{Z}$. By the Chinese theorem, among the elements having their order divisible by q modulo n, we have then identified $\varphi(n)\left(1-\frac{1}{q^{f_1}}\right)$ of them: the ones having their order modulo $p_1^{e_1}$ divisible by q. Now the others are among the $\varphi(n)\left(\frac{1}{q^{f_1}}\right)$ that remains. Just now consider those modulo $p_2^{e_2}$. If $f_2 == 0$ then we have not found any new element. Otherwise, $1-\frac{1}{q^{f_2}}$ of them are of order divisible by q. Well, actually, in both cases, we can state that $1-\frac{1}{q^{f_2}}$ of them are of order divisible by q. We have thus found some other elements: $\varphi(n)\left(\frac{1}{q^{f_1}}\right)\left(1-\frac{1}{q^{f_2}}\right)$. This added to the previously found elements makes $\varphi(n)\left(1-\frac{1}{q^{f_1}q^{f_2}}\right)$. Doing such a counting for each of the remaining $p_i^{e_i}$ gives the announced formula. □

For instance, take a Carmichael number still passing our test whenever $B \leq 1450$: $37690903213 = 229 \times 2243 \times 73379$. Well, $37690903212 = 19 \times 2^2 \times 3 \times 59 \times 1451 \times 1931$ and $\lambda(37690903213) = 19 \times 2^2 \times 3 \times 59 \times 1931$. Then, Q will be

1451×1931 and our algorithm will be able to find elements for which $\alpha^{\frac{n-1}{Q}} \not\equiv 1 \mod n$: those of which order is divisible by 1931. Unfortunately, there are quite a lot of them: $\varphi(n)\frac{1930}{1931} = 37489647840 \approx (1 - .0053396272268313 4975)n$. Thus, there are more than 5 chances over a thousand to choose an element α for which $\alpha^{\frac{n-1}{1451\times 1931}} \not\equiv 1 \mod n$. Even though this is much higher than $\frac{1}{Q}$ (if n was prime), this probability will not be detected abnormal by our algorithm.

7 Conclusion

We provide here a new very fast and efficient algorithm generating primitive roots. On the one hand, the algorithm has a polynomial time bit complexity when all existing algorithms where exponential. This is for instance illustrated when comparing it to existing software on figure 3.

On the other hand, our algorithm is probabilistic in the sense that the answer might not be a primitive root. We have seen in this paper however, that the chances that an incorrect answer is given are less important than say "hardware malfunctions". For this reason, we call our answers "Industrial-strength" primitive roots.

Then, we propose a new probabilistic primality test using this primitive root generation. This test can be viewed as a generalization of Miller-Rabin's test to other small prime factors dividing $n-1$. When a given probability of correctness is desirable for the test, our algorithm is heuristically competitive with repeated applications of Miller-Rabin's.

Acknowledgements

Many thanks to T. Itoh and E. Bach.

References

[1] Eric Bach. How to generate factored random numbers. *SIAM Journal on Computing*, 17(2):179–193, April 1988. Special issue on cryptography.

[2] Eric Bach. Comments on search procedures for primitive roots. *Mathematics of Computation*, 66(220):1719–1727, October 1997.

[3] Eric Bach and Jeffrey Shallit. *Algorithmic Number Theory — Efficient Algorithms*, volume I. MIT Press, Cambridge, USA, 1996. ISBN: 0-262-02405-5.

[4] Manuel Blum and Silvio Micali. How to generate cryptographically strong sequences of pseudo-random bits. *SIAM Journal on Computing*, 13(4):850–864, November 1984.

[5] David M. Burton. *Elementary number theory.* International series in Pure and Applied Mathematics. McGraw-Hill, quatrime edition, 1998.

[6] Richard Crandall and Carl Pomerance. *Prime Numbers, a computational perspective*. Springer, 2001.

[7] Whitfield Diffie and Martin E. Hellman. New directions in cryptography. *IEEE Transactions on Information Theory*, IT-22(6):644–654, 1976.

[8] Peter D. T. A. Elliott and Leo Murata. On the average of the least primitive root modulo p. *Journal of The london Mathematical Society*, 56(2):435–454, 1997.

[9] Paul Erdös, Carl Pomerance, and Eric Schmutz. Carmichael's lambda function. *Acta Arithmetica*, 58:363–385, 1991.

[10] John B. Friedlander, Carl Pomerance, and Igor Shparlinski. Period of the power generator and small values of Carmichael's function. *Mathematics of Computation*, 70(236):1591–1605, October 2001. See corrigendum [11].

[11] John B. Friedlander, Carl Pomerance, and Igor Shparlinski. Corrigendum to "Period of the power generator and small values of Carmichael's function". *Mathematics of Computation*, 71(240):1803–1806, October 2002. See [10].

[12] Joachim von zur Gathen and Jürgen Gerhard. *Modern Computer Algebra*. Cambridge University Press, New York, NY, USA, 1999.

[13] Joachim von zur Gathen and Igor Shparlinski. Orders of Gauss periods in finite fields. *Applicable Algebra in Engineering, Communication and Computing*, 9:15–24, 1998.

[14] Torbjörn Granlund. *The GNU multiple precision arithmetic library*, 2002. Version 4.1, `www.swox.com/gmp/manual`.

[15] Godfrey Harold Hardy and E. Maitland Wright. *An Introduction to the Theory of Numbers*. Oxford University Press, fifth edition, 1979.

[16] Toshiya Itoh and Shigeo Tsujii. How to generate a primitive root modulo a prime. Technical Report 009-002, IPSJ SIGNotes ALgorithms Abstract, 2001.

[17] Donald E. Knuth. *Seminumerical Algorithms*, volume 2 of *The Art of Computer Programming*. Addison-Wesley, Reading, MA, USA, troisième edition, 1997.

[18] Neal Koblitz. *A course in number theory and cryptography*, volume 114 of *Graduate texts in mathematics*. Springer-Verlag, Berlin, Germany / Heidelberg, Germany / London, UK /, etc., 1987.

[19] Gary L. Miller. Riemann's hypothesis and tests for primality. In *Conference Record of Seventh Annual ACM Symposium on Theory of Computation*, pages 234–239, Albuquerque, New Mexico, May 1975.

[20] John M. Pollard. A Monte Carlo method for factorization. *BIT Numerical Mathematics*, 15(3):331–334, 1975.

[21] Vaughan R. Pratt. Every prime has a succinct certificate. *SIAM Journal on Computing*, 4(3):214–220, 1975.

[22] Guy Robin. Estimation de la fonction de tchebycheff θ sur le k-ième nombre premier et grandes valeurs de la fonction $\omega(n)$ nombre de diviseurs premiers de n. *Acta Arithmetica*, XLII:367–389, 1983.

[23] Victor Shoup. Searching for primitive roots in finite fields. *Mathematics of Computation*, 58(197):369–380, January 1992.

[24] Samuel S. Wagstaff, Jr. *Cryptanalysis of numbers theoretic ciphers*. Chapman-Hall / CRC, 2003.