Fast Primality Proving on Cullen Numbers

Tsz-Wo Sze (szetszwo@cs.umd.edu)

September 20, 2009

Abstract

We present a unconditional deterministic primality proving algorithm for Cullen numbers. The expected running time and the worst case running time of the algorithm are $\tilde{O}(\log^2 N)$ bit operations and $\tilde{O}(\log^3 N)$ bit operations, respectively.

1 Introduction

A Cullen number is a number of the form

$$C_n = 2^n \cdot n + 1 \qquad \text{for } n \geq 1.$$

If C_n is a prime, it is called Cullen prime. These numbers were studied by James Cullen in 1905 [7]. The known Cullen primes include those with $n = 1$, 141, 4713, 5795, 6611, 18496, 32292, 32469, 59656, 90825, 262419 and 361275 [6, 10].

Theorem 1.1. *The primality of C_n can be determined in $\tilde{O}(\log^3 C_n)$ bit operations.*

Theorem 1.2. *There is a primality proving algorithm for C_n such that the expected running time and the worst case running time of the algorithm are $\tilde{O}(\log^2 C_n)$ bit operations and $\tilde{O}(\log^3 C_n)$ bit operations, respectively.*

In this paper, we prove the two theorems above without assuming any unproven hypothesis. We present two deterministic primality proving algorithms, Algorithm 2.2 and Algorithm 4.1 which are able to determine the primality of positive integers $N = 2^e t + 1$ with t odd and $2^e > t > 0$. The idea behind these algorithms was first introduced in [15] and [16].

For Cullen numbers, $N = C_n$, the expected running time and the worst case running time of the Algorithm 4.1 are $\tilde{O}(\log^2 N)$ bit operations and

$\tilde{O}(\log^3 N)$ bit operations, respectively. To the best of our knowledge, Algorithm 4.1 is the fastest among all the known deterministic primality proving algorithms which are applicable to Cullen numbers. The running time of the AKS algorithm [2] and Lenstra-Pomerance's modified AKS algorithm [9] are $\tilde{O}(\log^{7.5} N)$ and $\tilde{O}(\log^6 N)$, respectively. The Adleman-Pomerance-Rumely algorithm [1] runs in sub-exponential time. All the algorithms mentioned above have been proven unconditionally to be deterministic. With extra assumptions such as the Extended Riemann Hypothesis, we have the following results: The elliptic curve primality proving algorithm [4, 8] runs in $\tilde{O}(\log^5 N)$. The running time of Miller's algorithm [12] is $\tilde{O}(\log^4 N)$. The AKS algorithm can be improved [5, 11] to $\tilde{O}(\log^4 N)$. By Theorem 2.1, it is easy to construct an $\tilde{O}(\log^4 N)$ algorithm which checks congruence equation (2.1) with $2 \leq a \leq k$, where the positive integer k is $O(\log^2 N)$ by the results from Ankeny [3].

Algorithm 4.1 is also applicable to Fermat numbers, $N = 2^{2^n} + 1$. Comparing Algorithm 4.1 with the well known deterministic primality proving algorithm, Pépin's algorithm [13], both algorithms have the same expected running time, $\tilde{O}(\log^2 N)$. For the worst case running time, Pépin's algorithm is also $\tilde{O}(\log^2 N)$ but Algorithm 4.1 is $\tilde{O}(\log^3 N)$.

2 Primality Proving

Let $N = 2^e t + 1$ for some integer $e > 1$, t odd and $2^e > t > 0$. We show a fast primality proving algorithm for this form of numbers in this section. The main idea behind the algorithm is Theorem 2.1, Proth's Theorem. For a proof of Proth's Theorem, see [17].

Theorem 2.1 (Proth's Theorem)**.** *Let $N = 2^e t + 1$ for some odd t with $2^e > t$. If*

$$a^{(N-1)/2} \equiv -1 \pmod{N} \tag{2.1}$$

for some a, then N is a prime.

Our primality proving algorithm, Algorithm 2.2, tries to find an integer a satisfying equation (2.1). If such a is found, Theorem 2.1 asserts that N is a prime. The integer a is served as a primality certificate of N. If N is composite, such integer a does not exist. The algorithm detects this situation. Therefore, the algorithm is deterministic.

For integer x, a quadratic residue modulo N, denote a fixed square root of x modulo N by $\sqrt{x} \pmod{N}$.

Algorithm 2.2 (Primality Proving)**.** *The input is* $N = 2^e t + 1$ *for some integer* $e > 1$*,* t *odd and* $2^e > t > 0$*. This algorithm returns* PRIME *if* N *is a prime. Otherwise, it returns* COMPOSITE*.*

I. Try finding $a_2 = \sqrt{-1} \pmod N$ by Algorithm 2.3.
If Algorithm 2.3 halts due to N composite, return COMPOSITE.

II. For each $(3 \le j \le e)$ {
Try computing $a_j = \sqrt{a_{j-1}} \pmod N$ by Algorithm 3.3.
If Algorithm 3.3 halts due to N composite,
return COMPOSITE.
}

III. return PRIME with a_e as a primality certificate.

We discuss the first step below and the second step, the crucial step of Algorithm 2.2, in next section.

Step I can be computed as follows: Suppose N is a prime. Let H be the subgroup of $\mathbb{F}_N^\times$ with $2t$ elements, where $\mathbb{F}_N^\times$ denotes the multiplicative group of the finite field with N elements. For $1 \le i \le 2t+1$, there exists $i \notin H$. We have $i^{2t} \not\equiv 1 \pmod N$. If $i^{2^k t} \equiv -1 \pmod N$ for some $1 \le k \le e-1$, then $i^{2^{k-1}t} \equiv \sqrt{-1} \pmod N$.

If $j^{2t} \equiv 1 \pmod N$ for all $1 \le j \le 2t+1$, we deduce that N is composite since there are $2t+1$ elements with order dividing $2t$. For some $1 \le i \le 2t+1$, if $i^{2t} \not\equiv 1 \pmod N$ but $i^{2^k t} \not\equiv -1 \pmod N$ for all $1 \le k \le e-1$, then either

(1) $i^{2^e t} \not\equiv 1 \pmod N$, or

(2) $i^{2^k t} \not\equiv -1 \pmod N$ and $i^{2^{k+1} t} \equiv 1 \pmod N$ for some $1 \le k \le e-1$.

In case (1), N is composite by Fermat's Little Theorem. In case (2), $\gcd(i^{2^k t} - 1, N)$ is a non-trivial factor of N and so N is composite.

Algorithm 2.3 (Computing $\sqrt{-1} \pmod N$)**.** *The input is* $N = 2^e t + 1$ *for some integer* $e > 1$ *and odd* t*. If* N *is a prime, this algorithm returns* $\sqrt{-1} \pmod N$*. Otherwise, this algorithm either returns an integer congruent to* $\sqrt{-1} \pmod N$ *or halts due to* N *composite.*

I.1. Compute $b_j = j^{2t} \pmod N$ for $1 \le j \le 2t+1$.

I.2. If $b_j \equiv 1 \pmod N$ for all $1 \le j \le 2t+1$,
halt due to N composite.

I.3. Suppose $b_i \not\equiv 1 \pmod{N}$ for some $1 \le i \le 2t+1$.
Compute $b_i^{2^k} \pmod{N}$ for $0 \le k \le e-2$.

I.4. If $b_i^{2^k} \not\equiv -1 \pmod{N}$ for all $0 \le k \le e-2$,
halt due to N composite.

I.5. Suppose $b_i^{2^{k_0}} \equiv -1 \pmod{N}$ for some $0 \le k_0 \le e-2$.
Return $i^{2^{k_0} t} \pmod{N}$.

Algorithm 2.3 takes $\tilde{O}((t \log t + \log N) \log N)$ bit operations since steps I.1, I.2 take $\tilde{O}(t \log t \log N)$ bit operations and steps I.3, I.4, I.5 take $\tilde{O}(e \log N) = \tilde{O}(\log^2 N)$. The square root of $-1 \pmod{N}$ may also be computed by Schoof's square root algorithm [14], which takes $\tilde{O}(\log^9 N)$ bit operations.

3 Taking Square Roots

Given $\sqrt{-1} \pmod{N}$, a square root of a fixed value, computed in the previous section, we show how to compute the square roots of an arbitrary value when N is a prime. For more details of computing square roots with this idea, see [15] and [16].

Suppose N is a prime. Given a quadratic residue $\beta \pmod{N}$ with $1 < \beta < N-1$, we are going to find a square root of β modulo N. Suppose

$$\beta \equiv \alpha^2 \pmod{N} \qquad \text{for some integer } \alpha.$$

Define two sets G'_α and G_α as

$$G'_\alpha = \{[a] \,:\, a \not\equiv \pm\alpha \pmod{N}\}, \text{ and} \tag{3.1}$$

$$G_\alpha = G'_\alpha \cup \{[\infty]\}. \tag{3.2}$$

We denote the elements in G_α by $[\,\cdot\,]$ for avoiding confusion with the elements in $\mathcal{Z}$, where

$$\mathcal{Z} = \mathbb{Z} \cup \{\infty\},$$

where $\mathbb{Z}$ is the set of integers. Two elements $[a_1], [a_2] \in G'_\alpha$ are equal if and only if $a_1 \equiv a_2 \pmod{N}$. Therefore, there are exactly $N-2$ and $N-1$ elements in G'_α and G_α, respectively.

Further, define an operator $*$ as following: For any $[a] \in G_\alpha$ and any $[a_1], [a_2] \in G'_\alpha$ with $a_1 + a_2 \not\equiv 0 \pmod{N}$,

$$[a] * [\infty] = [\infty] * [a] = [a], \tag{3.3}$$

$$[a_1] * [-a_1] = [\infty], \tag{3.4}$$

$$[a_1] * [a_2] = \left[(a_1 a_2 + \alpha^2)(a_1 + a_2)^{-1}\right], \tag{3.5}$$

where x^{-1} denotes the multiplicative inverse of $x \pmod N$ for integer x with $\gcd(x, N) = 1$. Interestingly, $(G_\alpha, *)$ is a well-defined cyclic group. Instead of giving a direct proof, we show that G_α is isomorphic to $\mathbb{F}_N^\times$, the multiplicative group of the finite field with N elements.

Theorem 3.1. *$(G_\alpha, *)$ is a cyclic group isomorphic to $\mathbb{F}_N^\times$.*

Proof. Define a bijective mapping

$$\psi : G_\alpha \longrightarrow \mathbb{F}_N^\times, \qquad [\infty] \longmapsto 1, \qquad [a] \longmapsto (a+\alpha)(a-\alpha)^{-1} \tag{3.6}$$

with inverse mapping

$$\psi^{-1} : \mathbb{F}_N^\times \longrightarrow G_\alpha, \qquad 1 \longmapsto [\infty], \qquad b \longmapsto \left[\alpha(b+1)(b-1)^{-1}\right]. \tag{3.7}$$

A straightforward calculation shows that ψ is a homomorphism. □

In the rest of the paper, we drop the symbol $*$ and denote the group operation of G_α by multiplication. Algorithm 3.2 below shows how to perform multiplication in G_α. In Algorithm 3.2, the integer N may be a prime or a composite number since the algorithm is going to be used for proving the primality of N.

Algorithm 3.2 (Group Operation)**.** *The inputs are $N, \beta \in \mathbb{Z}$ and $a_1, a_2 \in \mathcal{Z}$ such that, for $i = 1, 2$, if $a_i \neq \infty$, then $a_i^2 \not\equiv \beta \pmod N$. If N is a prime, the input β is guaranteed to be a quadratic residue modulo N and this algorithm returns $a \in \mathcal{Z}$ such that $[a] = [a_1][a_2] \in G_\alpha$. Otherwise, this algorithm either returns some $a' \in \mathcal{Z}$ or halts due to N composite.*

1. If $a_1 = \infty$, return a_2.
2. If $a_2 = \infty$, return a_1.
3. If $a_1 + a_2 \equiv 0 \pmod N$, return ∞.
4. If $\gcd(a_1 + a_2, N) \neq 1$,
   halt due to N composite.
5. Compute $a \equiv (a_1 a_2 + \beta)(a_1 + a_2)^{-1} \pmod N$.
6. If $a^2 \equiv \beta \pmod N$,
   halt due to N composite.
   Otherwise, return a.

Algorithm 3.2 basically follows the group operation definitions (3.3), (3.4) and (3.5). It also handles the case if N is a composite number. In such case, G_α is no longer a well-defined group. If the algorithm halts in Step 4, a non-trivial factor of N is discovered and so N is a composite number. If the algorithm halts in Step 6, we have $(a_1a_2+\beta)^2 \equiv \beta(a_1+a_2)^2 \pmod N$, which implies $(a_1^2-\beta)(a_2^2-\beta) \equiv 0 \pmod N$. Since $a_i^2 \not\equiv \beta \pmod N$ for $i=1,2$ by the assumption, $a_1^2-\beta$ and $a_2^2-\beta$ are zero divisors, which means that N is composite.

Note that the value of α is not required as an input of Algorithm 3.2. Equipped with G_α and Algorithm 3.2, we are ready to describe how to compute square roots modulo N, which is the main ingredient of the Step II in Algorithm 2.2. In Algorithm 3.3 below, the notation $[x]^y$ means using Algorithm 3.2 and the successive squaring method to compute $[x]$ to the power y. Algorithm 3.3 halts due to N composite as soon as Algorithm 3.2 does, if it is the case.

Algorithm 3.3 (Taking Square Root Modulo N)**.** *The inputs are integers N, β and b such that $1<\beta<N-1$ and $b^2 \equiv -1 \pmod N$, where $N = 2^e t+1$ for some integer $e>1$ and odd t as before. If N is a prime, the input β is guaranteed to be a quadratic residue modulo N and this algorithm returns $\sqrt{\beta} \pmod N$. If N is a composite number, this algorithm either returns an integer congruent to $\sqrt{\beta} \pmod N$ or halts due to N composite.*

II.1. Check easy cases:

- (a) If $\gcd(b+1, N) \neq 1$,
  halt due to N composite.
- (b) If $j^2 \equiv \beta \pmod N$ for some $1 \leq j \leq 2t$, return j.

II.2. Find $[a]$ such that $[a]^2 \neq [\infty]$ and $[a]^4 = [\infty]$:

- (a) Compute $[c_j] = [j]^{2t}$ for $1 \leq j \leq 2t$.
- (b) If $[c_j] = [\infty]$ for all $1 \leq j \leq 2t$,
  halt due to N composite.
- (c) Suppose $[c_i] \neq [\infty]$ for some $1 \leq i \leq 2t$.
  Compute $[c_i]^{2^k}$ for $0 \leq k \leq e-2$.
- (d) If $[c_i]^{2^k} \neq [0]$ for all $0 \leq k \leq e-2$,
  halt due to N composite.
- (e) Suppose $[c_i]^{2^{k_0}} = [0]$ for some $0 \leq k_0 \leq e-2$.
  Compute $[a] = [i]^{2^{k_0}t}$.

II.3. Compute α:

(a) Compute $\alpha \equiv a(b-1)(b+1)^{-1} \pmod{N}$.

(b) If $\alpha^2 \not\equiv \beta \pmod{N}$,
halt due to N composite.
Otherwise, return α.

Correctness: Step II.1 checks two easy cases. Step II.1.(a) checks whether the inverse of $b+1 \pmod{N}$ exists. If $d = \gcd(b+1, N) \neq 1$, then d must be a non-trivial factor of N because if $b+1 \equiv 0 \pmod{N}$, then $b^2 \equiv 1 \not\equiv -1 \pmod{N}$. In Step II.1.(b), if j is a square root of β modulo N for some $1 \leq j \leq 2t$, we are done.

Step II.2 is similar to Algorithm 2.3. If N is a prime, both of them compute an order 4 element: Algorithm 2.3 computes a square root of $-1 \pmod{N}$, an order 4 element in $\mathbb{F}_N$, while an order 4 element $[a]$ in G_α is computed in Step II.2. Note that G_α is isomorphic to $\mathbb{F}_N^\times$ as shown in Theorem 3.1. The identity element of G_α is $[\infty]$. The order 2 element in G_α is $[0]$.

Suppose N is a prime. Denote the subgroup of G_α with $2t$ elements by H_α. If $[c_j] = [j]^{2t} = [\infty]$ for all $1 \leq j \leq 2t$, all the $2t+1$ elements in the set $\{[\infty]\} \cup \{[1], [2], \cdots, [2t]\}$ have order dividing $2t$ in G_α, which leads to a contradiction. Suppose $[c_i] \neq [\infty]$ for some $1 \leq i \leq 2t$. In Step II.2.(d), if $[c_i]^{2^k} \neq [0]$ for all $0 \leq k \leq e-2$, then $[i]^{2^{e-1}t} \neq [0]$, which implies $[i]^{|G_\alpha|} \neq [\infty]$, which is impossible if N is a prime. Therefore, N is composite if the algorithm halts at Step II.2.(b) or Step II.2.(d).

If N is a prime, the order of the element $[a] \in G_\alpha$ computed in Step II.2.(e) is exactly 4 since $[a]^2 = [0] \neq [\infty]$ and $[a]^4 = [\infty]$. On the other hand, $\pm b$ are the only order 4 elements in $\mathbb{F}_N^\times$. By Theorem 3.1, we have $\psi([a]) \equiv \pm b \pmod{N}$. In Step II.3, the integer $\alpha \equiv \pm a(b-1)(b+1)^{-1} \pmod{N}$ is a square root of β by the construction of G_α. If $\alpha^2 \not\equiv \beta \pmod{N}$, it leads to a contradiction, which means that N is a composite number.

Running Time Analysis: All the powers $[x]^y$ are computed by Algorithm 3.2 and the successive squaring method. It takes $\tilde{O}(\log y \log N)$ bit operations for computing $[x]^y$.

Step II.1 takes $\tilde{O}(t \log N)$ bit operations. In Step II.2, the running time of parts (a) and (b) together is $\tilde{O}(t \log t \log N)$, parts (c) and (d) together take $\tilde{O}(e \log N) = \tilde{O}(\log^2 N)$ bit operations, and part (e) takes $\tilde{O}(\log^2 N)$

bit operations. Therefore, Step II.2 takes $\tilde{O}((t \log t + \log N) \log N)$ bit operations in total. For Step II.3, it only takes $\tilde{O}(\log N)$ bit operations. The overall running time of the Algorithm 3.3 is $\tilde{O}((t \log t + \log N) \log N)$.

Finally, we show Theorem 1.1 below.

Proof of Theorem 1.1. By definition, Cullen number $C_n = 2^n \cdot n + 1$. Let $n = 2^m t$ for some positive integer m and t with t odd. Then, $C_n = 2^e t + 1$, where $e = n + m$. It is easy to see that $2^e > t$. If $n = 1$, the Cullen number $C_1 = 3$ is a prime. For $n > 1$, the primality of C_n can be determined by Algorithm 2.2. We show the correctness and analyze the running time below. Let $N = C_n$.

Correctness: If either Step I or Step II in Algorithm 2.2 returns COMPOSITE, the input N must be COMPOSITE by Algorithm 2.3 and Algorithm 3.3, respectively. Note that if N is a prime, the integer a_{j-1} in Step II is a quadratic residue modulo N for all $3 \leq j \leq e$.

Otherwise, Step III returns PRIME. In this case, N is indeed a prime by Theorem 2.1 with $a = a_e$, where a_e is computed in Step II.

Running Time Analysis: The order of magnitudes of e and t are both $O(\log N)$. Step I takes $\tilde{O}((t \log t + \log N) \log N) = \tilde{O}(\log^2 N)$ bit operations by Algorithm 2.3. By Algorithm 3.3, Step II takes $\tilde{O}(e(t \log t + \log N) \log N) = \tilde{O}(\log^3 N)$ bit operations. Step III can be done in $\tilde{O}(\log N)$ bit operations. Therefore, the running time of Algorithm 2.2 is $\tilde{O}(\log^3 N)$ bit operations. □

4 Randomized Primality Proving

Algorithm 2.2 first tries computing $a_2 \equiv \sqrt{-1} \pmod N$, and then it repeatedly takes square roots to obtain a_3, a_4, $\cdots$, a_e such that $a_j \equiv \sqrt{a_{j-1}} \pmod N$ for $3 \leq j \leq e$. If N is a prime, all the computations success and it ends up with a_e, a quadratic non-residue modulo N. Since it totally takes $e - 1$ square roots and each square root computation requires $\tilde{O}(\log^2 N)$ bit operations, the total running time for computing a_e is $\tilde{O}(e \log^2 N) = \tilde{O}(\log^3 N)$ bit operations. These square root computations dominate the running time of the entire algorithm.

In this section, we improve Algorithm 2.2 by repeatedly taking square roots on a randomly chosen integer, instead of the fixed integer -1. We first randomly choose an integer a. Then, we compute $\sqrt{a} \pmod N$, $\sqrt{\sqrt{a}}$

(mod N) and so on. If N is a prime, this process ends up with a quadratic non-residue modulo N.

For prime $N = 2^e t+1$, the multiplicative group $\mathbb{F}_N^\times$ being cyclic tells that most of elements in $\mathbb{F}_N^\times$ have order with large 2-part. Only a few number of square root computations are required in order to obtain a quadratic non-residue from these elements. In fact, there are half of the total number of elements in $\mathbb{F}_N^\times$ are quadratic non-residues modulo N. The order of a quadratic non-residue is divisible by 2^e. In general, for $1 \leq k \leq e$, there are exactly $2^{k-1}t$ elements having order divisible by 2^k but not 2^{k+1}. Only $e-k$ square root computations are required for obtaining a quadratic non-residue from these $2^{k-1}t$ elements. We present the randomized primality proving algorithm below.

Algorithm 4.1 (Randomized Primality Proving)**.** *The input is $N = 2^e t+1$ for some integer $e > 1$, t odd and $2^e > t > 0$. This algorithm returns* PRIME *if N is a prime. Otherwise, it returns* COMPOSITE.

(i) Find an integer b_k randomly such that 4 divides the of order $b_k \pmod N$:

  (a) Randomly choose an integer $1 < a < N-1$
  until $a^{2t} \not\equiv 1 \pmod N$.
  If there are $2t-1$ distinct integers $1 < a < N-1$
  such that $a^{2t} \equiv 1 \pmod N$, return COMPOSITE.

  (b) Compute $a_0 \equiv a^t \pmod N$.
  If $a_0^{2^e} \not\equiv 1 \pmod N$, return COMPOSITE.

  (c) Find the least $k > 0$ such that $a_0^{2^k} \equiv 1 \pmod N$.
  If $a_0^{2^{k-1}} \not\equiv -1 \pmod N$, return COMPOSITE.

  (d) Set $b_2 \equiv a_0^{2^{k-2}} \pmod N$.
  Set $b_k = a_0$.

(ii) For each $(1 + k \leq j \leq e)$ {
  Try computing $b_j = \sqrt{b_{j-1}} \pmod N$ by Algorithm 3.3.
  If Algorithm 3.3 halts due to N composite,
  return COMPOSITE.
}

(iii) return PRIME with b_e as a primality certificate.

Correctness: In Step (i)(a), we randomly choose an integer a from the open interval $(1, N-1)$ without replacement until $a^{2t} \not\equiv 1 \pmod N$. If

there are $2t-1$ distinct integers a in $(1, N-1)$ such that $a^{2t} \equiv 1 \pmod{N}$, then these $2t-1$ distinct integers together with 1 and $N-1$ are totally $2t+1$ distinct integers with modulo N order dividing $2t$. Therefore, N is composite. In Step (i)(b), if $a_0^{2^e} \equiv a^{N-1} \not\equiv 1 \pmod{N}$, then N is composite by Fermat's Little Theorem. In Step (i)(c), the least positive integer k with $a_0^{2^k} \equiv 1 \pmod{N}$ exists since $a_0^{2^e} \equiv 1 \pmod{N}$. We also have $k \geq 2$ because $a^{2t} \not\equiv 1 \pmod{N}$ by Step (i)(a). If $a_0^{2^{k-1}} \not\equiv -1 \pmod{N}$, then $\gcd(a_0^{2^{k-1}} - 1, N)$ is a non-trivial factor of N, and so N is composite. If Step (i)(a), (i)(b) and (i)(c) do not return COMPOSITE, we end up in Step (i)(d) that $b_2 \equiv a_0^{2^{k-2}} \equiv \pm\sqrt{-1} \pmod{N}$ and $b_k = a_0$ with $b_k^{2^{k-1}} \equiv -1 \pmod{N}$. Note that b_2 is required as an input of Algorithm 3.3 used in Step (ii). In the later of this section, we will show that the value of k is large with high probability.

Step (ii) and (iii) are similar to the Step II and III in Algorithm 2.2 except that Step (ii) begins taking square roots with b_k. If Algorithm 3.3 does not halt due to N composite, the invariant $b_j^{2^{j-1}} \equiv -1 \pmod{N}$ is maintained in the loop for $k \leq j \leq e$. If b_e is obtained, N is a prime by Theorem 2.1 with $a = b_e$.

Running Time Analysis: Step (i)(a) requires $\tilde{O}(t \log t \log N)$ bit operations. Steps (i)(b), (i)(c) and (i)(d) together take $\tilde{O}(\log^2 N)$ bit operations. The running time of Step (ii) is $\tilde{O}(m(t \log t + \log N) \log N)$ bit operations, where m is the maximum possible number of iterations in the loop. Step (iii) can be done in $\tilde{O}(\log N)$ bit operations. The entire algorithm is dominated by Step (ii). The total running time is $\tilde{O}(m(t \log t + \log N) \log N)$ bit operations.

The value of m depends on the integer a chosen randomly in Step (i)(a). It is easy to see that the worst case is $m = O(\log N)$. Thus, the worst case running time is $\tilde{O}((t \log t + \log N) \log^2 N)$. We will show the expected maximum number of iterations is indeed a constant, i.e. $E(m) = O(1)$. As a consequence, the expected running time is $\tilde{O}((t \log t + \log N) \log N)$.

Let $v_2(x)$ be the 2-adic valuation function. For positive integer $x = 2^r s$ with s odd, we have $v_2(x) = r$, which is the exponent of the 2-part of x. Let $\mathrm{ord}_p(a)$ be the order of $a \pmod{p}$ for prime p and $a \not\equiv 0 \pmod{p}$. We show in Lemma 4.2 below that the expected value of $v_2(\mathrm{ord}_p\, a)$ for a randomly chosen integer a is bounded below by $v_2(p-1) - 1$.

Lemma 4.2. *Let $p = 2^{e'} t' + 1$ be an odd prime for some odd t' and $e' \geq 1$. Let a be an integer randomly chosen from the open interval $(1, p-1)$ such that $a^{2d} \not\equiv 1 \pmod{p}$ for some positive divisor d of t'. Then the expected*

value

$$E(v_2(\mathrm{ord}_p\, a)) > e' - 1.$$

Proof. By counting the number of integers $a \in (1, p-1)$ such that $v_2(\mathrm{ord}_p\, a) = i$ for $i = 0, 1, \cdots, e'$, we have

$$\begin{aligned} \sum_{\substack{1<a<p-1 \\ a^{2d} \not\equiv 1 \pmod p}} v_2(\mathrm{ord}_p\, a) &= 0 \cdot (t' - d) + 1 \cdot (t' - d) + \sum_{i=2}^{e'} i \cdot 2^{i-1} t' \\ &= (e' - 1)(p - 1) + t' - d. \end{aligned}$$

Then, the expected value is

$$\begin{aligned} E(v_2(\mathrm{ord}_p\, a)) &= e' - 1 + \frac{2d(e' - 1) + t' - d}{p - 1 - 2d} \\ &> e' - 1. \end{aligned}$$

□

Expected Maximum Number of Iterations, $E(m)$: Suppose N is a prime. Recall that a is a randomly chosen integer in Step (i)(a) such that $a^{2t} \not\equiv 1 \pmod N$ and $k = v_2(\mathrm{ord}_N\, a)$. By Lemma 4.2 with $d = t$, we have $E(k) = E(v_2(\mathrm{ord}_N\, a)) > e - 1$. Therefore, $E(m) = E(e - k) = O(1)$.

Suppose N is composite. Let p be a prime divisor of N such that $v_2(p-1)$ is the minimum among all the prime divisors of N. Write $p = 2^{e'} t' + 1$. Clearly, we have $e' \le e$. If the algorithm does not discover N composite in Step (i), the maximum number of iterations is bounded above by e', i.e. $m \le e' \le e$. Let a be the integer chosen in Step (i)(a). If p divides a, then Step (i)(b) will return COMPOSITE since $a_0^{2^e} \equiv a^{2^e t} \not\equiv 1 \pmod N$. Suppose p does not divide a. By Lemma 4.2 with $d = \gcd(t, t')$, we have $E(v_2(\mathrm{ord}_p\, a)) > e' - 1$, which implies $E(m) = O(1)$.

As a conclusion, the expected maximum number of iterations in Step (ii) is $O(1)$ for prime or composite N.

Proof of Theorem 1.2. Algorithm 4.1 can be used to determine the primality of Cullen numbers C_n. For $N = C_n$, the expected running time and the worst case running time of Algorithm 4.1 are $\tilde{O}(\log^2 C_n)$ bit operations and $\tilde{O}(\log^3 C_n)$ bit operations, respectively. □

For the number in the form $N = 2^e t + 1$, there is a well known fast probabilistic primality proving algorithm, which randomly chooses an integer a

and then checks the congruence equation (2.1). The process is repeated until an a is found such that

$$a^{(N-1)/2} \not\equiv 1 \pmod{N}. \tag{4.1}$$

For such a, if $a^{(N-1)/2} \equiv -1 \pmod{N}$, then a is prime by Theorem 2.1. Otherwise, a is composite by Fermat's Little Theorem. There is a high probability that such a can be found within a few iterations. The expected running time, which is also $\tilde{O}(\log^2 N)$ bit operations, is a little bit better than Algorithm 4.1 by a small constant factor. However, the number of iterations to find such a is $O(N)$ in the worst cases. Therefore, the algorithm has an exponential running time in the worst cases.

In practice, Algorithm 4.1 can be combined with the probabilistic algorithm mentioned in the previous paragraph. An integer a is randomly chosen until a satisfies equation (4.1) or the number of iterations reaches to a fixed limit m. If all the m chosen a's do not satisfy equation (4.1), we select the one with order having the largest 2-part among those a's and use it in Algorithm 4.1. Such hybrid approach has roughly the same exact expected running time as the probabilistic algorithm and has the same order of worst case running time as Algorithm 4.1.

5 A Numerical Example

Consider

$$N = C_{141} = 393050634124102232869567034555427371542904833.$$

With Algorithm 2.2, we first compute

$$a_2 = 328337527527414723914576799806385366095264867 \tag{5.1}$$

such that $a_2^2 \equiv -1 \pmod{N}$. Then, we compute $a_3, a_4, \cdots, a_{141}$ and obtain

$$\begin{aligned} a_3 &= 34894726410835542200345415804166056711193393, \\ a_4 &= 191997998663833236900292517250656100336076727, \\ &\vdots \\ a_{141} &= 367816872098652281367044660748960111937242897 \end{aligned}$$

such that $a_i^2 \equiv a_{i-1} \pmod{N}$ for $i = 3, 4, \cdots, 141$. These congruence equations imply

$$a_i^{2^{i-1}} \equiv -1 \pmod{N}.$$

By Theorem 2.1, the Cullen number C_{141} is a prime and the integer a_{141} is a primality certificate of C_{141} since

$$a_{141}^{(N-1)/2} \equiv (-1)^{141} \equiv -1 \pmod{N}.$$

We use Algorithm 4.1 to determine the primality of C_{141} again. Suppose $a = 2$ is chosen in Step (i)(a). Then, $a_0 \equiv a^{141} \pmod{N}$. We find $k = 139$ in Step (i)(c) since $a_0^{2^{138}} \equiv -1 \pmod{N}$ and $a_0^{2^{139}} \equiv 1 \pmod{N}$. In Step (i)(d), we get

$$\begin{aligned} b_2 &\equiv a_0^{2^{k-2}} \pmod{N}, \\ b_{139} &= 27875931498163278926919647840810451882475 52. \end{aligned}$$

Since we already have b_{139}, only two iterations are needed in Step (ii) for computing

$$\begin{aligned} b_{140} &= 37295148844985067101576087682628780309282 8048, \\ b_{141} &= 16222971329271189512283344470163234024593 2509. \end{aligned}$$

We have shown that C_{141} is a prime again with b_{141} as a primality certificate of C_{141}.

References

[1] Leonard M. Adleman, Carl Pomerance, and Robert S. Rumely. On distinguishing prime numbers from composite numbers. *Ann. of Math.*, 117(1):173–206, January 1983.

[2] Manindra Agrawal, Neeraj Kayal, and Nitin Saxena. PRIMES is in P. *Ann. of Math. (2)*, 160(2):781–793, 2004.

[3] Nesmith C. Ankeny. The least quadratic non residue. *Ann. of Math.*, 55(1):65–72, January 1952.

[4] A. O. L. Atkin and F. Morain. Elliptic curves and primality proving. *Math. Comp.*, 61(203):29–68, 1993.

[5] Daniel J. Bernstein. Proving primality in essentially quartic random time. *Math. Comp.*, 76(257):389–403, 2007.

[6] Chris Caldwell. The top twenty: Cullen primes, 2008. `http://primes.utm.edu/top20/page.php?sort=Cullen`.

[7] James Cullen. Question 15897. *Educ. Times*, page 534, December 1905.

[8] S Goldwasser and J Kilian. Almost all primes can be quickly certified. In *Proceedings of the 18th Annual ACM Symposium on the Theory of Computing*, pages 316–329, 1986.

[9] Hendrik W. Lenstra Jr. and Carl Pomerance. Primality testing with Gaussian periods, 2005. Preliminary version. Available at `http://www.math.dartmouth.edu/~carlp/PDF/complexity12.pdf`.

[10] Wilfrid Keller. New Cullen primes. *Math. Comp.*, 64(212):1733–1741, 1995.

[11] Preda Mihăilescu and Roberto M. Avanzi. Efficient "quasi"-deterministic primality test improving AKS. Available at `http://www-math.uni-paderborn.de/~preda/`.

[12] Gary L. Miller. Riemann's hypothesis and tests for primality. In *STOC '75: Proceedings of seventh annual ACM symposium on Theory of computing*, pages 234–239, New York, NY, USA, 1975. ACM Press.

[13] P. Pépin. Sur la formule $2^{2^n} + 1$. *Comptes Rendus Acad. Sci. Paris*, 85:329–333, 1877.

[14] René Schoof. Elliptic curves over finite fields and the computation of square roots mod p. *Math. Comp.*, 44(170):483–494, April 1985.

[15] Tsz-Wo Sze. *On Solving Univariate Polynomial Equations over Finite Fields and Some Related Problems*. PhD thesis, University of Maryland, 2007. Available at `http://hdl.handle.net/1903/7632`.

[16] Tsz-Wo Sze. On taking square roots without quadratic nonresidues over finite fields, 2009. Available at `http://arxiv.org/abs/0812.2591`.

[17] Hugh C. Williams. *Édouard Lucas and Primality Testing*, volume 22 of *Canadian Mathematical Society Series of Monographs and Advanced Texts*. Wiley-Interscience, 1998.