Designer Primes

Anna M. Johnston
amj at juniper dot net
Juniper Networks

December 8, 2020

Abstract

Prime integers are the backbone of most public key cryptosystems. Attacks often go after the primes themselves, as in the case of all factoring and index calculus algorithms. Primes are time sensitive cryptographic material and should be periodically changed. Unfortunately many systems use fixed primes for a variety of reasons, including the difficulty of generating trusted, random, cryptographically secure primes. This is particularly concerning in the case of discrete log based cryptosystems. This paper describes a variant of provable prime generation, intended for discrete logarithm based cryptography, based off Pocklington's theorem with improved efficiency, flexibility and security.

1 Change Your Prime!

Primes used in most discrete logarithm based public key cryptographic systems are fixed. The number field sieve, the attack driving size considerations in finite field cryptography, goes after the prime itself. Once a successful attack has occurred, discrete logarithms modulo this prime become relatively easy and every system based on this problem is broken. While the best attacks against these primes are currently infeasible, the fact that the same small set of primes is used everywhere and never changes makes these primes exceedingly high value targets. If the unthinkable happens, and one or more of these primes was successfully attacked, data from banks, industry, government, and personal accounts would be vulnerable.

Variable	Description
P	The potential prime (eventually proven prime) Designer Primes will generate. This will have the form $(Rh + 1)$.
R	A divisor of $(P - 1)$, with known factorization $R = \prod_{j=1}^{t} r_j^{mj}$ where r_j are distinct primes and m are positive integers.
r_j	Known prime divisors of R
m_j	Exponents of known prime divisors of R, with $m_j \in \mathbb{Z}^{<0}$.
h	Random component of P.
$h^{(j)}$	h may be quite large. If h needs to be split up for the stepping (section 3.1) procedure, it will be $$h = h^{(2)}2^{\nu_1+\nu_0} + h^{(1)}2^{\nu_0} + h^{(0)}$$ with $0 \leq h^{(j)} < 2^{\nu_j}$.
ν_j	The number of bits in each portion of h (see above)

Figure 1: Designer Primes Variables

The best way to mitigate this threat is to treat primes in cryptographic systems as time-sensitive cryptographic material. Just as symmetric secret keys should be periodically changed, so should the public primes used in finite field cryptography.

This paper describes Designer Primes, a new provable prime generation algorithm based off Pocklington's theorem [12]. The algorithm improves both security and efficiency using a technique which simultaneously eliminates the 'prime gap' (definition C.2) problem and improves the weeding out of composite integers.

1.1 Strong Provable Primes

Prime generation algorithms based off Pocklington's theorem are not only provable, but the generation process is more efficient and provides the necessary data to very quickly repeat the primality proof [9]. A quick primality proof is essential to security: using a composite modulus or subgroup order can severely undermine the security of discrete logarithm based public key cryptography[13].

The form of a prime integer p and the group $(\mathbb{Z}/p\mathbb{Z})^*$ contributes to the strength and capabilities of many public key cryptosystems. If the order of $\mathbb{Z}/p\mathbb{Z}^*$ is *smooth* (definition C.3) then discrete logarithms over $\mathbb{Z}/p\mathbb{Z}^*$, along with the cryptosystems based upon them, can be broken [15] [18].

Furthermore, a cryptographic systems functionality may require the prime to have certain properties. For example, systems based on the difficulty of computing an r^{th} root [8] [1] (where r is a large prime) in a finite group, r^2 must divide the order of the group. If the group is $(\mathbb{Z}/p\mathbb{Z})^*$, then $p = hr^2+1$ for some integer h.

Pocklington's theorem (theorem 4.1.3 in [3]) gives a simple primality test for integers of the form $P = (hR+1)$ where the complete factorization of R is known[1]. This theorem is the cornerstone of Designer Primes, with sieving and linear feedback shift registers (LFSR's) employed to improve efficiency and randomness.

2 Pocklington's Theorem and Prime Generation

Pocklington's theorem (theorem A.2) enables a simple primality check on an integer P if enough of the factorization of $(P-1)$ is known. Let:

$$P = hR + 1 \tag{1}$$

$$R = \prod_{k=1}^{t} r_k^{m_k} \tag{2}$$

where h is a random integer, r_k are distinct prime integers, and m_k are positive integers. If certain size constraints hold on h in comparison to R, primality can be proven with only $(t+1)$ modulo P exponentiations[2].

[1] A watered down version of Pocklington's theorem in [7] was used in provable prime generation algorithms described in NIST FIPS 186-3.

[2] Composite integers are usually revealed in a single exponentiation.

2.1 Pocklington Test for Primality and Extensions

Pocklington's theorem (theorem A.2) tells us that if $P = hR + 1$ passes a few tests, then all factors of P 'look' like P. In other words, if q_j is a factor of P, then:

$$q_j = (Rh_j + 1) \text{ with } h_j \geq 1.$$

If P is composite, it must have at least two factors of this form. These factors force h to be fairly large — greater than R. If $h \leq R$, then P can not be composite and must be prime.

Extensions on the bounds of h (see section 3.3 and A.4) increase the flexibility of the algorithm. Increasing the bound on h increases the potential for entropy in the prime while reducing the computational cost of the algorithm by reducing necessary steps (i.e., faster growth) and required exponentiations.

2.2 Bootstrapping

Pocklington's theorem based prime generation starts with smaller, known prime numbers (R), finds larger primes using random h values which pass Pocklington's primality test (theorem A.3), repeating these steps to find successively larger primes. This bootstrapping technique is used in many existing algorithms, including [11], [2], [19], and [10].

2.3 Prime Properties

There are several properties required in cryptographic prime numbers.

If P is the prime being generated then:

Size: The size of the prime must generally be within a given range bit range: $2^B < P < 2^{B+1}$, with B determined by current security standards and user requirements. A larger B increases the cost of field based attacks such as the number field sieve [6] and its variants.

Group Structure: Security and functional requirements for the cryptosystem dictate the structure of the multiplicative group – i.e., constraints on the factorization of $(P-1)$. At a bare minimum there must be at least one large prime factor of $(P-1)$, r, with $r > 2^b$, with b determined by current security standards. A larger b increases the cost of both field based attacks (number field sieve) and group based attacks (for examples see [17], [15] [13]).

Other Attack Prevention: Designer Primes are intended for use in finite field based cryptography. If the primes generated will be used in factoring, the resulting prime P must have a large prime divisor of $(P+1)$ to avoid Lucas sequence based attacks [20]. There are no discrete logarithm attacks known using smooth (definition C.3) of $(P+1)$. Forcing these restrictions on primes used for finite field cryptography reduces randomness and increases computational costs, and should not be used.

2.4 Probability of finding a prime and maximal R

Pocklington's theorem tests for primality, but it does not guarantee that a prime of the given form (equation 1) exists. The probability of finding a prime of this form depends on the size of R and the desired size of P.

We can compute the probability of a prime existing of the given size and form using the prime number theorem [4]. If $\pi(x)$ is the number of primes less than x, the prime number theorem states:

$$\lim_{x\to\infty} \log(x)\,\pi(x)\,x^{-1} = 1 \tag{3}$$

or for very large x

$$\pi(x) \sim x\,(\log(x))^{-1} \tag{4}$$

Since the final cryptographic primes will be quite large ($> 2^{2048}$), we use the approximation for the number of primes (equation 4). The expected

number of primes $2^a < P < 2^b$ is approximately:

$$\left(\pi\left(2^b\right) - \pi(2^a)\right) \sim 2^a \left(2^{b-a}a - b\right) (ab\log(2))^{-1}.$$

If $R \approx 2^c$, then the number of primes with this form in this range is:

$$\left(\pi\left(2^b\right) - \pi(2^a)\right) 2^{-c} \sim 2^{a-c} \left(2^{b-a}a - b\right) (ab\log(2))^{-1}. \tag{5}$$

To improve our chances of finding a prime in the given range and increase the entropy in each prime, we'd like the expected number of primes (equation 5) to be greater than a positive value n, where $\lg(n)$ is a minimal bound on the maximal entropy of the prime.

$$n \leq 2^{a-c} \left(2^{b-a}a - b\right) (ab\log(2))^{-1},$$

The largest R we should use then is:

$$c < a + \lg\left(2^{b-a}a - b\right) - (\lg(a) + \lg(b) + \lg(\log(2))) - \lg(n). \tag{6}$$

As an example, if $a = 2048$ and $b = 2049$ and we want the expected number of primes in our search to be $n = 2^{256}$, then the maximal R would be 2^c with

$$c = 2048 + \lg(2047) - (11 + \lg(2049) - 0.5288) - 256 = 1781.527$$

In other words, we can generate a 2048-bit prime with 2^{256} expected primes in this range by choosing a $R < 2^{1909}$.

2.4.1 Safe is unsafe

Safe primes have the form $P = 2Q + 1$ where Q is also prime. These are often chosen for finite field cryptography due the maximal size of the multiplicative subgroup, which protects against subgroup based attacks [13], [14], [15], [17].

While a safe-prime has the advantage of a maximal multiplicative subgroup, this comes at a cost of greatly reduced entropy and increased generation costs when primes are regularly changed. There are far fewer safe

primes than generic cryptographically secure Pocklington primes, increasing the computational cost of generating and reducing the entropy contained in the prime, thus reducing the long term security when primes are regularly changed.

For example, let's say we want to find a 2048 bit primes. What must we do to generate a safe prime and one with a 1024-bit prime divisor of $(P-1)$?

We'll look at the simple finite field strong prime first, with $R = 2Q$. Given Q, the number of 2048-bit primes $P = 2Qh+1$ expected in the range $2^{2048} < P < 2^{2049}$ is:

$$\left(\pi\left(2^{2049}\right) - \pi\left(2^{2049}\right)\right) 2^{-1025} \approx 2^{1023} \left(4096 - 2049\right) \left(2048 \cdot 2049 \log(2)\right)^{-1} \approx 2^{1012}.$$

If Q is a 2047-bit prime and $P = 2Q + 1$, then this expected number drops to:

$$\left(\pi\left(2^{2048}\right) - \pi\left(2^{2049}\right)\right) 2^{-2048} \approx \left(4096 - 2049\right) \left(2048 \cdot 2049 \log(2)\right)^{-1} \approx 2^{-10.47}.$$

In other words, to generate a safe prime of this form, you'd expect to have to first generate around 2048 primes, each 2047 in size. In the strong prime case, there should be more than enough primes in the given range from a single 1024 bit prime.

2.5 Algorithm Outline

All Pocklington based prime generation algorithms have the same basic structure. To generate a prime P with $2^B < P < 2^{B+1}$ and a set of constraints on the multiplicative subgroup (i.e., the form of R), primes will be grown starting from a smaller known prime (or set of primes) to the desired prime. The single prime generation algorithm is called first with the known prime(s), then repeated using its output, until the desired size and shape of prime has been generated.

The following algorithm outlines the prime generation step. Input for the algorithm is the set of prime divisors in R, and the range for h. How

these values are chosen depend on the stage of the generation process and the desired form of the final prime. For example, the range of smaller primes should be bounded only by the testing requirements, while later primes should be bounded to insure appropriate sizes.

Algorithm 2.1: Generate single prime

Input:

$\{r_j, m_j \mid 1 \leq j \leq t\}$	Known factors for P, with $R = \prod_{j=1}^{t} r_j^{mj}$
h_L, h_U	Bounds for h with $h_L \leq h < h_U \leq R^2$

Output:

P	prime number, or 0 for FAILURE

I: Choose a random starting value $h_L \leq h_0 < h_U$ and set $h = h_0$.

II: While $P = (Rh + 1)$ fails primality test (section 3.2):

A: Increment h (section 3.1);

B: If $h = h_0$, then return 0 (FAILURE);

III: return P

End of Algorithm 2.1

Designer Primes differs from other Pocklington based generation techniques in how both the testing and stepping of h is performed.

- Stepping (section 3.1) is improved by using Galois registers to step through the h values. This insures every h value within the bounds can occur once and only once, as in a traditional additive stepping, but with (for all practical purposes) random step sizes. The random step size minimizes entropy loss in the generation process from the prime-gap bias, improving the entropy contained in the final output prime.

- Testing (section 3.2) is improved by tying it to the stepping process, eliminating trial division. Sieving (section 3.4) 'pre-checks' all potential prime numbers for divisibility by small primes making trial division unncessary.

3 Algorithm Details

3.1 Incrementing h

The value of h is incremented at step j of algorithm 2.1. The technique for incrementing h should satisfy several constraints:

Non-repeating: The range for h is generally very large, therefore the probability of a repeated h value is quite small. However, this is still a concern if (1) the range for h is small; (2) the randomizer is poor.

Random stepping: After choosing a random starting point for h, it could be stepped by one each time. However, large gaps exist between some primes but not between others. Single stepping h may bias the algorithm towards those primes on the edges of the gaps.

Bounded: Bounds on P are obtained through bounds, first on R, then on h.

3.1.1 Stepping through h values with Galois Registers

Using Galois registers to step h solves both the non-repeating and random requirements. Galois registers are a simple, deterministic way to imitate a random number generator. Its output passes many random tests, but cycles over all possible non-zero values. If a k bit register starts at an initial random point $0 < reg < 2^k$ it can be stepped $2^k - 1$ times before it repeats.

Galois registers use polynomials, polynomial arithmetic (figure 2) and modular polynomial arithmetic (figure 3) as described in appendix B

The following algorithm iterates through a Galois register, given a current non-zero initial and current state.

Algorithm 3.1: Galois Step Register

Input:

reg_0	starting register value
reg	current register value

Output: Modifies reg by stepping it with the fixed, degree k-polynomial.

I: Constants –

ply	integer representing lower k coefficients of a degree k primitive binary polynomial
$mask$	2^{k-1}

II: If $reg = 0$: Return ERROR — *register can not zero.*

III: if $reg\&mask = 0$: $reg = reg << 1$ — *Left shift register over one bit (multiply by 2)*

IV: else: $reg = ((reg \oplus mask) << 1) \oplus ply$ — *Left shift and reduce*

V: if $reg = reg_0$: Set $reg = 0$ — *ran out of possible reg values, except zero*

VI: return

End of Algorithm 3.1

A sieve (section 3.2) to reduce trial division costs will be performed over the size of the register. If the register is k-bits long, and 64-bit words are used to store the sieve data, then 2^{k-6}-words are needed for the sieve. For $k = 16$, the sieve will require a 1024 long array of 64-bit words.

The sieve restrictions bound the size of the Galois register. In most cases it will not cover the entire h range. Instead, h will be divided into sections, each k-bits long.

Let $h_L \leq h < h_U$. Then we can rewrite h as

$$h = h_L + h'$$

where $0 \leq h' < h_R$ and $h_R = (h_U - h_L)$.

The number of acceptable values for h' will be divided into k-bit words. These k-bit words will be initialized with non-zero values, then Galois stepped to find subsequent trials.

Let $s = \left\lfloor \frac{\lg h_R}{k} \right\rfloor$, then

$$h' = h^{(s)} 2^{sk} + \sum_{j=0}^{s-1} h^{(j)} 2^{jk}$$

where $0 \leq h^{(j)} < 2^k$ for $0 \leq j < s$ and $0 \leq h^{(s)} < \left\lfloor \frac{h_R}{2^{sk}} \right\rfloor$.

All words of h' are randomly initialized with a non-zero value within their bounds:

$$h'_0 = h_0^{(s)} 2^{sk} + \sum_{j=0}^{s-1} h_0^{(j)} 2^{jk}.$$

Stepping (algorithm 3.2) occurs in a clock like fashion, with the $(s-1)$ term stepping every time until all possible values have been tried. If more steps are necessary, the $(s-2)$ term is stepped once before returning to step the $(s-1)$ term again, and so on.

Algorithm 3.2: Extended Galois Stepping

Input:

$h_0 = h_0^{(s)} 2^{sk} + \sum_{j=0}^{s-1} h_0^{(j)} 2^{jk}$	starting register values, $h_0^{(j)} \neq 0$ for $0 \leq j \leq s$
$h' = h^{(s)} 2^{sk} + \sum_{j=0}^{s-1} h^{(j)} 2^{jk}$	current register value, $h^{(j)} \neq 0$ for $0 \leq j \leq s$

Output: Modifies h', returning an error if the boundaries have been exhausted.

I: Set $j = (s-1)$,

II: while $(j \geq 0)$ AND (the call to algorithm 3.1 with $\left(h_0^{(j)}, h^{(j)}\right)$ returns an error) : *loop until a new random value can be stepped into; note: this added stepping is VERY rarely needed.*

A: Subtract one from j: $j = j - 1$ — *move the index to try the next lowest k-bit register*

III: If $j < 0$: return ERROR — *No new random values can be found*

IV: Else return

End of Algorithm 3.2

3.2 Primality test

Pocklington's theorem (theorem A.2) and the resulting primality tests require two conditions (equation 9, 8) and bounds on h (theorem A.3, A.4). The conditions and bounds are used together with some common sense techniques. These are detailed in algorithm 3.3.

Algorithm 3.3: Primality Test

Input:

P	Prime to be tested
`wasSieved`	Boolean, true if sieve was used for small divisors
$\{r_j \mid 1 \leq j \leq t\}$	Divisors of $(P-1)$, ordered with $r_j^{m_j} > r_{j+1}^{m_{j+1}}$

Output:

prime status	prime or composite

I: Set `status` = `unsure`

II: If `wasSieved` = `false`: Test for small primes divisors – return composite if a divisor is found — *check for small divisors if needed*

III: Set $g = 2$ — *Base for Pocklington's tests*

IV: While `status` = `unsure`:

A: If $g^{P-1} \not\equiv 1 \bmod P$: Set `status` = `composite` — *Fails Fermat test (equation 8)*

B: $j = 1$ *index for prime divisors of R*

C: $\texttt{rem} = (P-1)$ *allows all powers of R to be removed from (P − 1)*

D: Set:$\texttt{divisors} = 1$ *Stores size of divisors: with (P − 1) = rem · divisors*

E: While $\texttt{status} = \texttt{unsure}$ and $\texttt{divisors} < \texttt{rem}^2$ and $j \leq t$:

1: Compute $x \equiv g^{P/r_j} - 1 \bmod P$ *Pocklington's second test (equation 9)*

2: If $\gcd(x, P) = 1$:

i: While $\texttt{rem} \equiv 0 \bmod r_j$: Set

- $\texttt{rem} = \texttt{rem}/r_j$ *remove from re*
- $\texttt{divisors} = \texttt{divisors} \cdot r_j$ *and pu*

ii: Else: Add one to g: $g = g + 1$, and set $\texttt{divisors} = 0$ *try a different is probably pri*

F: If $\texttt{rem} \leq \texttt{divisors}$: $\texttt{status} = \texttt{prime}$ *Satisfies simple Pocklington's test*

G: Else if $\texttt{rem}^2 \leq \texttt{divisors}$

1: $\texttt{status} = \text{PocklingtonExtension}\,(h, R)$ *Algorithm 3.4*

V: return status.

End of Algorithm 3.3

3.3 Extending Pocklington's Bound

Pocklington based prime generation test (theorem A.3) restricts the size of the random portion of the prime h, to less than the known portion, R. While this test allows the prime to double in size at each iteration, there is a simple extension which allows the prime to triple in size at each interation. Not only does this extension improve the growth rate of primes, but it increases entropy and reduces computation costs.

The extension (theorem A.4) changes the bound on h from $h \leq R$ to $h \leq R^2$. The added test is fairly simple. Assuming the integer passes the exponentiation tests and $h < R^2$, then if $\left((h \bmod R)^2 - 4\left\lfloor \frac{h}{R} \right\rfloor\right)$ is not a perfect square, P is prime.

Testing an integer to see if it is not a perfect square is fairly simple. If an integer x is a perfect square, then it must be a quadratic residue modulo q for any prime q. If x is not a quadratic residue for any prime q, then x is not a perfect square.

Algorithm 3.4 tests for quadratic residue modulo a set of small primes. If the integer is not a quadratic residue for even a single prime, it is not a perfect square and it passes the extension test.

Algorithm 3.4: Pocklington's Extension Test

Input:

h	Random portion of the prime
R	Known portion of the prime

Output: true if prime, false otherwise

I: Constants : A set of small primes $\{q_j \mid 0 \leq j < 1000\}$, with $q_0 = 5, q_1 = 7$ an so on.

II: set `status` = false

status of the prime test. As soon as one prime fails the quadratic residue test, the status changes to true

III: Set $b = (h \bmod R)^2$

IV: Set $a = 4 \cdot \left\lfloor \frac{h}{R} \right\rfloor$

V: If $b < a$: `neg`= true, `val`=$a - b$

Computing $b - a$, keeping track of the negative sign

VI: else `neg`= false, `val`=$b - a$

VII: set $\mathtt{j} = 0$

VIII: While $\mathtt{j} < 1000$ and $\mathtt{status} =$ false

A: Compute $v = \mathtt{val}^{(q_j-1)/2} \bmod q_j$

B: if `neg` is true:

1: if $(q_j - 1)/2$ is odd and $v = 1$: $\mathtt{status} =$ true — *$b - a$ is a quadratic non-residue*

2: else $(q_j - 1)/2$ is even and $v = q_j - 1$: $\mathtt{status} =$ true — *$b - a$ is a quadratic non-residue*

C: else if $v = (q_j - 1)$: $\mathtt{status} =$ true

D: $\mathtt{j} = \mathtt{j} + 1$

End of Algorithm 3.4

3.4 Sieving for Small Factors

Though there are several techniques for weeding out the composites with small prime factors, sieving is the most efficient when searching for large prime numbers. Let $S = \{q_i \mid 1 \leq i < s\}$ be the set of all small primes less than a chosen bound and relatively prime to R. Sieving quickly searches a set of possible h-values for those which $P = hR + 1$ are not divisible by any prime in S. It does this by creating an array A of bits, with each bit corresponding to a particular h. If the bit is 'on' (i.e., 1), no small primes divide P; if it is 'off' the integer is divisible by some small prime in S.

To create this array, work is done modulo $q_i \in S$. The first h such that P is divisible by q_i needs to be found. Recall that

$$P = \left(h_L + \sum_{j=0}^{s} h^{(j)} 2^j \right) R + 1.$$

Only one of the $h^{(j)}$ words will be sieved over at a time – generally $j = (s-1)$. The rest remain fixed. For the general case, assume that all words are fixed

except $j = w$. To find the first h value, set $P \equiv 0 \bmod q_i$.

$$0 \equiv \left(h_L + \sum_{j=0}^{s} h^{(j)} 2^{kj} \right) R + 1 \bmod q_i$$

$$h_i^{(w)} \equiv - \left(\left(R 2^{wk} \right)^{-1} + \sum_{\substack{j=0 \\ j \neq w}}^{s} h^{(j)} 2^{(j-w)k} + h_L 2^{-wk} \right) \bmod q_i$$

Not only is $(h_i R + 1)$ divisible by q_i using this value of $h^{(w)}$, but so is any $(hR + 1)$ where $h^{(w)} \equiv h_i^{(w)} \bmod q_i$.

Using a bit array the of a given length (say 2^k long), the sieve starts with all bits on – indicating good h-values. For each prime q_i, the solution $h_i^{(w)}$ is turned off, then $h_i^{(w)} + q_i$, $h_i + 2q_i$, and so on. The remaining bits on will be $h^{(w)}$ values whose corresponding P value is not divisible by any prime in S.

Algorithm 3.5: Basic Sieving of h-values

Input:

$S = \{q_i\}\, 1 \leq i < s$	small prime set
R	the known prime factorization of $P = hR + 1$
k	the number of bits to sieve
$w, \{h^{(j)}\}$	fixed portions of h, with $0 \leq j \leq s$, and $j \neq w$

Output:

A	k-long bit array with good h values 'on'

I: Create an k-long bit array A and set all k bits of A 'on' (set to 1)

II: for each prime $q_i \in S$

A: $h^{(w)}[i] = -\left(\left(R2^{wk}\right)^{-1} + \sum_{\substack{j=0 \\ j \neq w}}^{s} h^{(j)} 2^{(j-w)k} + h_L 2^{-wk}\right) \bmod q_i$

B: set $v = h^{(w)}[i]$

C: while $v < 2^k$:

1: turn off bit v of A;

2: Set $v = v + q_i$

III: return A

End of Algorithm 3.5

As an example, we'll solve for h_i values for $R = 1999$ and $S = \{2, 3, 5, 7, 11\}$. The first step is to solve for h_i, in this case $h_i(1999) + 1 \equiv 0 \bmod q_i$, or $\equiv (-1)(1999)^{-1} \bmod q_i$. The starting solutions are:

i	1	2	3	4	5
q_i	2	3	5	7	11
h_i	1	2	1	5	4

The sieve data A is stored as a bit array. In this example, the actual values are placed in each 'bit' position for clarity.

For this example, sieving is performed for the values from 200 to 249. The starting array will be:

200	201	202	203	204	205	206	207	208	209
210	211	212	213	214	215	216	217	218	219
220	221	222	223	224	225	226	227	228	229
230	231	232	233	234	235	236	237	238	239
240	241	242	243	244	245	246	247	248	249

For clarity again, instead of turning the cell 'off', the divisor

Starting with $q_1 = 2$, the first value divisible by 2 will be equivalent to $h \equiv h_1 \bmod 2$ and greater than or equal to 200: i.e. 201. Sieving out the

values divisible by $q_1 = 2$ gives:

200	2	202	2	204	2	206	2	208	2
210	2	212	2	214	2	216	2	218	2
220	2	222	2	224	2	226	2	228	2
230	2	232	2	234	2	236	2	238	2
240	2	242	2	244	2	246	2	248	2

For $q_2 = 3$, $h \equiv 2 \bmod 3$, which is 200. Sieving out the values divisible by $q_2 = 3$ gives:

3	2	202	2,3	204	2	3	2	208	2,3
210	2	3	2	214	2,3	216	2	3	2
220	2,3	222	2	3	2	226	2,3	228	2
3	2	232	2,3	234	2	3	2	238	2,3
240	2	3	2	244	2,3	246	2	3	2

sieving out those divisible by $q_3 = 5$, the smallest value greater than or equal to 200 and equivalent to 1 mod 5 is 201:

3	2,5	202	2,3	204	2	3,5	2	208	2,3
210	2,5	3	2	214	2,3	5	2	3	2
220	2,3,5	222	2	3	2	5	2,3	228	2
3	2,5	232	2,3	234	2	3,5	2	238	2,3
240	2,5	3	2	244	2,3	5	2	3	2

For $q_4 = 7$, the starting value must be equivalent to 5 mod 7. This starting value is $201 \equiv 5 \bmod 7$.

3	2,5,7	202	2,3	204	2	3,5	2	7	2,3
210	2,5	3	2	214	2,3,7	5	2	3	2
220	2,3,5	7	2	3	2	5	2,3	228	2,7
3	2,5	232	2,3	234	2	3,5,7	2	238	2,3
240	2,5	3	2,7	244	2,3	5	2	3	2

Finally, the starting value for $q_5 = 11$ must be equivalent to 4 mod 11. This

starting value is 202.

3	2,5,7	11	2,3	204	2	3,5	2	7	2,3
210	2,5	3	2,11	214	2,3,7	5	2	3	2
220	2,3,5	7	2	3,11	2	5	2,3	228	2,7
3	2,5	232	2,3	234	2,11	3,5,7	2	238	2,3
240	2,5	3	2,7	244	2,3	5,11	2	3	2

This leaves 10 remaining h values between 200 and 249. Six of these h-values generate prime numbers The other six are divisible by primes greater than 11.

A Pocklington's theorem and corollaries

The properties, given below, can be found in elementary number theory books such as [16].

Definition A.1 (Order of elements)**:** Let P and g be integers such that the $\gcd(P, b) = 1$. The order of g modulo P is the smallest positive integer d such that $g^d \equiv 1 \bmod P$.

Fact: If d is the order of an integer modulo a prime p then d divides $p-1$ ($p-1 = dx$ for some integer x). If the order of g is d in $\mathbb{F}_p$ and $g^x \equiv 1 \bmod p$, then d divides x or $x = dy$ for some integer y.

Theorem A.2 (Pocklington, 1914)**:** Let P, h, R be integers with

$$\begin{aligned} P &= hR + 1 \\ R &= \prod_{k=1}^{t} r_k^{m_k} \end{aligned} \tag{7}$$

where r_k are distinct prime integers. If there exists an integer g with

$$g^{hR} \equiv 1 \bmod P \tag{8}$$

$$\gcd\left(\left(g^{\frac{hR}{r_k}} \bmod P\right) - 1, P\right) = 1 \qquad \text{for all } 1 \le k \le t \tag{9}$$

then all prime factors of P are congruent to one modulo R.

Proof. Let q be a prime divisor of P and d be the multiplicative order of $g \bmod q$. Equation (8) gives us that $d \,|hR$ and equation (9) gives us that $g^{\frac{hR}{r_k}} \not\equiv 1 \bmod q$. Together this implies that $r_k^{m_k} \,|d$ for all k and that $R \,|d$. Since $d \,|(q-1)$ and $R \,|d$, we have $R \,|(q-1)$ and $q \equiv 1 \bmod R$.

□

If an integer passes the Pocklington's tests (equation 7) and either $h \le R$ (theorem A.3) or $h < R^2$ and a related integer is not a perfect square (theorem A.4), then P is prime.

Corollary A.3 (Provable Primality Test)**:** Let P be defined as in equation (7) with and assume that equations 8, 9 hold for all $1 \leq i \leq t$. If $h \leq R$ then P is a prime integer.

Proof. From theorem A.2 we know that any prime q dividing P satisfies $q \equiv 1 \bmod R$. Assume that P is composite with $q \,|P$.

1. Since $q \equiv 1 \bmod R$ and $q\frac{P}{q} = P \equiv 1 \bmod R \;\therefore\; \frac{P}{q} \equiv 1 \bmod R$ and

$$\begin{aligned} q &= h_0 R + 1 \\ \frac{P}{q} &= h_1 R + 1 \end{aligned}$$

   for some $h_0, h_1 > 0$.

2. Multiplying these two equations out gives:

$$\begin{aligned} q\frac{P}{q} &= (h_0 R + 1)(h_1 R + 1) \\ &= \left(h_0 h_1 R + (h_0 + h_1)\right) R + 1 \\ &= hR + 1 \end{aligned}$$

   Therefore, since $h_0, h_1 > 0$,

$$\begin{aligned} h &= h_0 h_1 R + h_0 + h_1 \\ h &> R \end{aligned}$$

3. The third assumption of the proof states that $h \leq R$, which contradicts the assumption that P was composite.

Therefore P is prime. □

The bound on Pocklington's prime text can be extended ([2], theorem 5 and [10], lemma 2) with a simple computation.

Corollary A.4 (Extension to Pocklington's)**:** Let $P, R,\ g$ be defined as in theorem A.2, satisfying equations (8) and (9); define

$$\beta \equiv h \bmod R \tag{10}$$

$$\gamma = \left\lfloor \frac{h}{R} \right\rfloor \tag{11}$$

If $R^3 \geq P$ and $\left(\beta^2 - 4\gamma\right)$ is not a perfect square, then P is prime.

Proof. 1. From theorem A.2, $P = \prod_{k=1}^{n} (Rh_k + 1)$ with $h_k \geq 1$. Since $R^3 \geq P$ and each factor $(Rh_k + 1) > R$, $n < 3$. If P is composite, $n = 2$ and

$$\begin{aligned} P = hR + 1 &= (Rh_1 + 1)(Rh_2 + 1) \\ &= R(Rh_1h_2 + (h_1 + h_2)) + 1 \\ h &= Rh_1h_2 + (h_1 + h_2) \qquad R^3 \geq N = Rh + 1 \\ \hline R^2 &> h \qquad\qquad R > h_1h_2 \end{aligned} \tag{12}$$

2. $h_1 + h_2 < R$ (by contradiction):

   Assume $(h_1 + h_2) \geq R$, and without loss of generality, that $h_1 \leq h_2$, with $(h_1 + h_2) = R + \delta$ with $\delta \geq 0$. We know

$$\begin{aligned} R > h_1h_2 &= h_1(R + \delta - h_1) \\ &> Rh_1 + \delta h_1 - h_1^2 \\ h_1^2 &> R(h_1 - 1) + \delta h_1 \qquad\qquad R > h_1h_2 \geq h_1^2 \end{aligned}$$

   If $h_1 > 1$, then $R > h_1^2 > R$, which is a contradiction. If $h_1 = 1$,

$$R^2 > h = \left(R^2 + R\delta - R\right) + (R + \delta) \geq R^2,$$

   which is also a contradiction. Therefore $(h_1 + h_2) < R$.

3. Since $h_1 + h_2 < R$, the reduced value $\beta \equiv h \bmod R$ is equal to $h_1 + h_2$ and $\gamma = \left\lfloor \frac{h}{R} \right\rfloor$ equals $h_1 h_2$.

4. Knowing the sum and product of h_1, h_2, we know that

$$(x - h_1)(x - h_2) = x^2 - \beta x + \gamma$$

therefore $h_1, h_2 \in \left\{ \left(\beta \pm \left(\beta^2 - 4\gamma \right)^{1/2} \right) / 2 \right\}$.

5. If $\left(\beta^2 - 4\gamma\right)$ is a perfect square, then R is composite and we know its factors. If it is not a perfect square, R is prime.

□

B Polynomial and Extension Field Arithmetic

- A polynomial f over $\mathbb{F}_2$ is a polynomial with all coefficients either one or zero, and the notation $f \in \mathbb{F}_2[x]$ is used to show that f is a polynomial. For example, $f(x) = x^3 + x + 1$.

- Binary polynomials are often stored as binary registers. For example, the polynomial $f(x) = x^3 + x + 1$ can be represented as

x^3	x^2	x^1	x^0
1	0	1	1

  Notice that it takes $k + 1$ bits to represent a degree k polynomial.

- Polynomials can be added or multiplied in the usual way, but with operations in $\mathbb{F}_2$. For example (in both polynomial and binary notation):

$$\left(x^2 + 1\right) + (x + 1) = x^2 + x \qquad [101] + [011] = [110]$$
$$\left(x^2 + x + 1\right)\left(x^3 + x^2 + 1\right) = x^5 + x + 1 \qquad [0111][1101] = [100011]$$

Figure 2: Binary Polynomial Representation and Operations

- If $f \in \mathbb{F}_2[x]$ has degree $deg(f) = k > 0$, then we can work modulo $f(x)$ (i.e., in $\mathbb{F}_2[x]/f(x)$). Polynomial operations are identical as in the normal $\mathbb{F}_2$ polynomials, but now we have equivalences defined by f. For example, if $f(x) = x^4 + x^3 + 1$ (degree $k = 4$) then $f(x) \equiv 0 \bmod f(x)$, and $x^4 \equiv x^3 + 1 \bmod f(x)$ (remember subtraction is the same as addition in binary). Every polynomial with degree k or grater can be reduced by replacing any x^{k+t} with $\left(x^3 + 1\right) x^t$.

$$\begin{aligned}\left(x^2 + x + 1\right)\left(x^3 + x^2 + 1\right) &= x^5 + x + 1 \\ &\equiv (x^3 + 1)x + x + 1 \\ &\equiv (x^3 + 1) + x + x + 1 \\ &\equiv [1000] \bmod [1101]\end{aligned}$$

- A polynomial $f \in \mathbb{F}_2[x]$ is *primitive* with degree $k > 0$ if

  1. f has no divisors in $\mathbb{F}_2[x]$ except 1 and itself.
  2. Every non-zero element in $\mathbb{F}_2[x]/f(x)$ can be represented as a power of x (i.e., x is a generator). For example, $f(x) = x^4+x^3+1$ is primitive:

j	$x^j \bmod f$			
	x^3	x^2	x^1	x^0
0	0	0	0	1
1	0	0	1	0
2	0	1	0	0
3	1	0	0	0
4	1	0	0	1

j	$x^j \bmod f$			
	x^3	x^2	x^1	x^0
5	1	0	1	1
6	1	1	1	1
7	0	1	1	1
8	1	1	1	0
9	0	1	0	1

j	$x^j \bmod f$			
	x^3	x^2	x^1	x^0
10	1	0	1	0
11	1	1	0	1
12	0	0	1	1
13	0	1	1	0
14	1	1	0	0

Figure 3: Binary Extension Field Operations

Tables of primitive polynomials can be found in many places; the following are but a few examples:

$$x^8 + x^5 + x^3 + x^2 + 1 \qquad x^{10} + x^3 + 1$$
$$x^{12} + x^{11} + x^{10} + x^6 + x^3 + x^2 + 1 \qquad x^{13} + x^{12} + x^9 + x^3 + 1$$

C Prime Generation Definitions and Comments

Prime integers are irregularly spaced.

Definition C.1 (Adjacent primes)**:** Two primes p_{i-1}, p_i are adjacent if $p_{i-1} < p_i$ and every integer q with $p_{i-1} < q < p_i$ is composite.

Definition C.2 (Prime Gap)**:** The prime gap G_i for adjacent primes p_{i-1}, p_i is the set of integers between the two primes: $G = \{k \in \mathbb{Z} \mid p_{i-1} < k \geq p_i\}$

Prime generation algorithms search for primes, usually choosing a random starting point and incrementing until a prime has been found. This initial guess is in some prime gap G_i, and the prime generated will be the prime p_i. Primes p_i with larger prime gaps G_i have a higher probability of being generated. This prime gap bias is eliminated with the stepping in the Designer Primes algorithm. The larger the prime gap for a prime, the higher the probability that the srimes p_i with larger gaps G_i have the higher the probability of a random start occurring in G_i. being generated.

Definition C.3 (Smooth Integers)**:** Smooth integer with respect to bound B: An integer S is smooth if it is divisible only by small primes q with each prime $q < B$. The order of $\mathbb{Z}/p\mathbb{Z}^*$ is $(p-1)$.

References

[1] Cheryl Beaver, Peter Gemmell, Anna Johnston, and William Newmann, *On the cryptographic value of the q^{th} root problem*, Proceedings of the International Conference on Information and Computer Security, Lecture Notes in Computer Science, Springer, 1999, Sydney, Australia, pp. 135–142.

[2] John Brillhart, D.H. Lehmer, and J.L. Selfridge, *New primality criteria and factorizations of* $2^m \pm 1$, Mathematics of Computation **29** (1975), no. 130, 620–647.

[3] R. Crandall and C. Pomerance, *Prime numbers: A computational perspective*, second ed., Springer-Verlag, 175 Fifth Avenue, New York, New York 10010, U.S.A., 2005.

[4] ———, *Prime numbers: A computational perspective*, second ed., ch. 1.1.5, p. 10, in theorem 1.1.4 [3], 2005.

[5] ———, *Prime numbers: A computational perspective*, second ed., ch. 4.1, p. 175, in theorem 4.1.3 [3], 2005.

[6] D.M. Gordon, *Discrete logarithms in $gf(p)$ using the number field sieve*, SIAM Journal on Discrete Mathematics (1993), no. 6, 124–138.

[7] G.H. Hardy and W.M. Wright, *An introduction to the theory of numbers*, fifth ed., Oxford Science Publications, New York, 1979.

[8] Anna Johnston and Peter Gemmell, *Authenticated key exchange provably secure against the man-in-the-middle attack*, Journal of Cryptology **15** (2002), no. 2, 139–148.

[9] Anna Johnston and Rathna Ramesh, *Prime proof protocol*, 2019 paper, to be published.

[10] Ueli M. Maurer, *Fast generation of prime numbers and secure public?? key cryptographic parameters*, Journal of Cryptology **8** (1995), 123–155.

[11] William J. Miller and Nick G. Trbovich, *Rsa public-key data encryption system having large random prime number generating microprocessor or the like*, 1982, US Patent assigned to Racal-Milgo Inc; expired 2000.

[12] Henry C. Pocklington, *The determination of the prime or composite nature of large numbers by fermat's theorem*, Proceedings of the Cambridge Philosophical Society, no. 18, University of Cambridge, 1914–1916, pp. 29–30.

[13] S.C. Pohlig and M.E. Hellman, *An improved algorithm for computing logarithms over $gf(p)$ and its cryptographic significance*, Transactions on Information Theory, no. 24, IEEE, 1978, pp. 106–110.

[14] J.M. Pollard, *Monte carlo methods for index computation (mod p)*, Mathematics of Computation, no. 32, 1978, pp. 918–924.

[15] ———, *Kangaroos, monopoly and discrete logarithms*, Journal of Cryptology **13** (2000), 437–447.

[16] Kenneth H. Rosen, *Elementary number theory and its applications*, third ed., Addison-Wesley Publishing Co., Reading, Massachusettes, 1993.

[17] Daniel Shanks, *Class number, a theory of factorization, and genera*, Proceedings of Symposia in Pure Mathematics **10** (1969), 415–440.

[18] ———, *Five number-theoretic algorithms*, Proceedings of the Second Manitoba Conference on Numerical Mathematics, no. VII, 1972, University of Manitoba, Winnipeg, Manitoba, pp. 51–70.

[19] J. Shawe-Taylor, *Generating strong primes*, Electronics Letters **22** (1986), 875–877.

[20] H. C. Williams, *A p+1 method of factoring*, Mathematics of Computation **39** (1982), no. 159, 225–234.