# Prime Proof Protocol

Anna M. Johnston, Rathna Ramesh
amj at juniper dot net
Juniper Networks

December 14, 2020

**Abstract**

Prime integers form the basis for finite field and elliptic curve cryptography, as well as many other applications. Provable prime generation guarantees primality and is more efficient than probabilistic generation, and provides components for an efficient primality proof. This paper details a protocol which takes in the proof components from the generation process, proves primality, and as an added benefit, supplies the user with a subgroup generator.

## 1 The Need for a Prime Proof Protocol

Primes form the bases for many existing asymmetric cryptographic systems. Attacks against these systems often focus on primes themselves, introducing the idea of cryptographically strong, weak, secure, or even broken prime integers. If primes can be broken, then they are similar to cryptographic keys. It is essential to the security of the systems to change this parameter regularly. We propose that cryptographic primes have life cycle, just as other cryptographic parameters.

Current cryptographic systems using primes have not had the ability to change primes regularly. Fixed primes are the norm in most asymmetric systems based off the discrete logarithm problem. One of the main reasons for this glaring security hole is that updating a prime isn't quite as easy as updating most cryptographic parameters. We can generate new primes easily enough, but exchanging new prime integers has some special challenges. A user who receives a new (possibly) prime integer $P$ for use in finite field based asymmetric system must be sure:

- $P$ is actually prime;
- $P$ has a cryptographically suitably sized subgroup, of order $r$[7];
- $r$ is prime;
- the generator integer used, $g$, has order $r$;

The protocol described in this paper does exactly this. It allows the proof of a prime to be transmitted and its primality and structure easily checked. The proof process also results in a known generator of the the order required.

## 2 Prime Generation Basics

There are two general categories of prime generation techniques: probabilistic and provable. Let $P$ be the integer we want to test for primality. Both categories are based on the subgroup structure of multiplicative groups modulo $P$ (i.e., mod$P$), and use exponentiation to gain information about this structure.

Probabilistic techniques are fairly simple, theoretically, but require many large modulo $P$ exponentiations to reduce the probability of $P$ being composite. Even then, there is no guarantee that the number generated is prime. For cryptographic uses, the threat is compounded by the need for a second prime: if a prime $P$ is needed for finite field cryptography, then a second large prime $r$ must divide $(P-1)$ [7][8] to prevent group order based attacks. If either prime ($P$ or $r$) is actually composite, the cryptographic algorithms may be compromised[11].

Provable prime generation[5], [10], [4], on the other hand, is theoretically more complex, but requires only a small number of exponentiations — generally only two. Based off Pocklington's theorem[1] (see appendix A, as well as [6] and [3]), provable primes are generated by bootstrapping from smaller known primes.

The purpose of a prime tree (figure 1) is to prove that its root is prime. The leaves of the tree are all small primes whose primality is known. Working up from the leaves, proving nodes along the way results in an efficient prime proving algorithm.

[1] Testing random numbers for primality can be done using AKS, but it is not computationally practical.

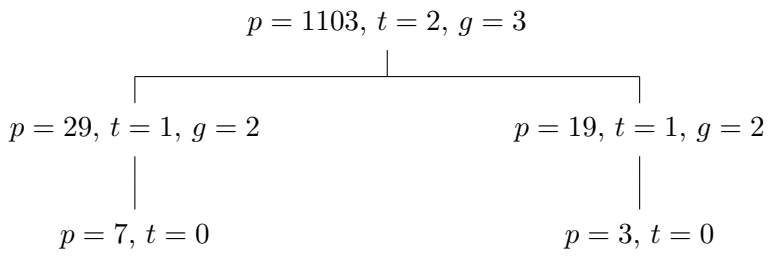


Figure 1: Simple, concrete prime proof tree with $p$ is the prime for the node, $t$ is the number of children, and $g$ is an integer modulo $p$ used to prove primality

In our algorithm, the prime proof tree is converted to a linear list. The algorithm works through this list of unproven primes (input list), converting them into a list of proven primes (output list). As the algorithm continues, primes from the output list are used (and removed) to prove primality on the input list integers, which are then added to the output list. The end result (assuming a properly formed prime proof tree) is an empty input list and a single remaining prime (the root) on the output list.

See examples (section B) for examples of how the proof progresses.

# 3 Prime Proof Algorithm

The prime proof protocol stores potential primes and the data for proving in a tree structure. To prove an integer $P$ is prime, one or more prime divisors of $(P-1)$ and an element (integer) of a passable order.

Two structure are used to organize the prime proof. For simplicity, a structures elements will be written as `a:b` where `a` is a structure variable or a pointer to the variable and `b` is the element being referenced. The shorthand MPI is used to indicate a multiple precision integer.

1. `PossiblePrime` is used to store the prime proof information;

2. `prime` is used to store a list of previously proved primes needed to proof future

primes.

Data Structure: PossiblePrime

**Purpose:** Linked list of primes and information which enables the prime proof protocol

**Data:**
- ▷ `p`: MPI, holds the prime being tested for primality;
- ▷ `numChild`: integer, holds count of the number of prime children required to prove primality;
- ▷ `gen`: MPI, holds the integer element which passes the prime test for `prime`;
- ▷ `next`: pointer to a PossiblePrime, holds the next prime element in the list.

Data Structure: prime

**Purpose:** Simple linked list of primes

**Data:**
- ▷ `q` : MPI, holding the known prime;
- ▷ `next`: pointer to prime, holding the next prime in the list.
- ▷ getNewPrime($p, list$): allocates and returns a new `prime` for the list, initializing it with a MPI (p) that is prime and a pointer (list) to the existing list

---

**Algorithm 3.1: Non-Recursive Prime Proof Protocol**

Input:
- `proof`: a pointer to the first `PossiblePrime` in the proof list;
- `smallP`: a list of small proven primes, generally pre-generated with the sieve of Eratosthenes.

Output:
a validation integer: `status` $= \begin{cases} -1 & \text{bad PossiblePrime tree} \\ 0 & \text{prime proof failed} \\ 1 & \text{prime has been validated} \end{cases}$

`prime` pointer: containing validated prime or nil (failure)

*I*:

| | | |
|---|---|---|
| `status` = 1 | : | Contains success/failure code of proof |
| `primes` = $\emptyset$ | : | `prime`, pointer to a list of proven primes; |
| `qDiv` = 1 | : | MPI, stores the product of prime divisors of current prime minus 1 |
| `qRem` = 1 | : | MPI, stores the remaining integer after dividing the current prime (minus 1) by `qDiv` |
| `q` = 1 | : | MPI, stores current known prime that is being used at the current stage of the proof; |
| `P` = 1 | : | MPI, stores integer currently being tested for primality; |
| `g` = 1 | : | MPI, stores base used to test current prime; |
| `num` = 1 | : | integer, stores number of primes used to prove `P` is prime. |

*Internal variables and initial settings:*

*II*: while `proof` $\neq \emptyset$ and `status` = 1:

*Continue while there are primes left to check and no errors occurred*

*A*:
$$\begin{cases} \texttt{P=proof:p} \\ \texttt{g=proof:gen} \\ \texttt{num=proof:numChild} \\ \texttt{proof=proof:next} \end{cases}$$

*Get the next prime, generator, and number of child primes from the proof list, and remove the first proof element.*

*B*: If `num` = 0

*Current possible prime should be in the small primes list*

*1*: if `P` is in `smallP`: `primes` = getNewPrime(`P`, `primes`)

*2*: else: `status` = 0

*failure – given prime is not acceptable*

*C*: else:

*1*:
$$\begin{cases} \texttt{qRem=P} - 1 \\ \texttt{qDiv=1} \end{cases}$$

*Initialize remainder of prime order and divisors*

*2*: if $\texttt{g}^{\texttt{qRem}} \not\equiv 1 \bmod \texttt{P}$: `status` = 0

*Not prime; fails Fermat test*

*3*: $j = 0$

*4*: while $j <$ `num` and `status` = 1

*continue until all the children needed to prove primality are used or failure*

*i*: if `primes` = 0: `status` = $-1$

*Failure: not enough proven primes for this stage of proof*

*ii*: else:

- ◇ $\begin{cases} \texttt{q=primes:q} \\ \texttt{primes=primes:next} \end{cases}$ *Get the next known prime off the list and update the known prime list*
- ◇ if $\texttt{P} \not\equiv 1 \bmod q$: $\texttt{status} = -1$ *Failure: child prime does not divide the parent minus one*
- ◇ else:
  - ◇ while $\texttt{qRem} \equiv 0 \bmod \texttt{q}$: $\begin{cases} \texttt{qRem=qRem/q} \\ \texttt{qDiv=qDiv} \cdot \texttt{q} \end{cases}$ *Divide out all copies of q from the remainder, multiply them to the divisors*
  - ◇ if $\gcd\left(\left(\texttt{g}^{(\texttt{P}-1)/\texttt{q}} - 1 \bmod \texttt{P}\right), \texttt{P}\right) \neq 1$: $\texttt{status} = 0$ *Failure: failed Pocklington's second test*

*iii*: $j = j + 1$

*5*: If Algorithm 3.2 with $\texttt{qRem}, \texttt{qDiv}$ returns false: $\texttt{status} = -1$ *The known prime divisors were too small or failed Pocklington's extended test*

*6*: else: $\texttt{primes} = \text{getNewPrime}(\texttt{P}, \texttt{primes})$ *Prime is proved; Add to known primes list.*

*III*: Return `status` and `Primes`.

End of Algorithm 3.1

---

**Algorithm 3.2: Pocklington's Extension Test**

Input:

| | |
|---|---|
| $h$ | Random portion of the prime |
| $R$ | Known portion of the prime |

Output: true if prime, false otherwise

Preliminary

*I*:

| | |
|---|---|
| Constants : | A set of small primes $\{q_j \mid 0 \leq j < 1000\}$, with $q_0 = 5, q_1 = 7$ an so on. |
| Simple size: | If $h < R$, return true |
| : | else if $h > R^2$, return false |
| Status : | `status` = false |

*set constants, check basic size constraints, and set status*

*II*: Set $b = (h \bmod R)^2$ — *b and a are multiple precision integers*

*III*: Set $a = 4 \cdot \lfloor \frac{h}{R} \rfloor$

*IV*: If $b < a$: $\texttt{neg}$= true, $\texttt{val}$=$a - b$ — *Computing b − a, keeping track of the negative sign*

*V*: else $\texttt{neg}$= false, $\texttt{val}$=$b - a$

*VI*: set $\texttt{j} = 0$

*VII*: While $\texttt{j} < 1000$ and $\texttt{status}$ = false

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

*B*: if $\texttt{neg}$ is true:

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

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

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

*D*: $\texttt{j} = \texttt{j} + 1$

End of Algorithm 3.2

# A Pocklington's Theorem

**Theorem A.1** (Pocklington, 1914)**:** Let $P, h, R$ be integers with

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

where $r_k$ are distinct prime integers. If there exists an integer $g_k$ for each $r_k$ with

$$g_k^{hR} \equiv 1 \bmod P \tag{2}$$

$$\gcd\left(\left(g_k^{\frac{hR}{r_k}} \bmod P\right) - 1, P\right) = 1 \qquad \text{for all } 1 \leq k \leq t \tag{3}$$

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

*Proof.* The proof to this can be found in [6],[10], and others. The following is a short sketch of the proof. Equations (2) and (3) imply that all factors of $P$ have the form $(Rh_j + 1)$, or that:

$$P = \prod_j \left(Rh_j + 1\right), \quad \text{for some } h_j \geq 1. \tag{4}$$

If $P$ is composite, it will have at least two factors and

$$\begin{aligned} P = (Rh + 1) &\geq (Rh_0 + 1)(Rh_1 + 1) \\ &\geq R^2 h_0 h_1 + R(h_0 + h_1) + 1 \\ h &\geq Rh_0 h_1 + (h_0 + h_1) \end{aligned}$$

If $h \leq R$, then $h_0, h_1$ can not be greater than or equal to one. This implies only one factor can exist, and $P$ is prime. □

Note that for simplicity, a single $g$ which works for all small factors, $r_k$, is generally chosen (i.e., $g = g_k$ for all $k$).

## A.1 Extending Pocklington's Bound

Pocklington based prime generation test (theorem A.1) 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 iteration. Not only does this extension improve the growth rate of primes, but it increases entropy and reduces computation costs.

The bound on Pocklington's prime text can be extended ([2], theorem 5 and [4], lemma 2) with a simple computation.

**Corollary A.2** (Extension to Pocklington's)**:** Let $P, R,\ g$ be defined as in theorem A.1, satisfying equations (2) and (3); define

$$\beta \equiv h \bmod R \tag{5}$$

$$\gamma = \left\lfloor \frac{h}{R} \right\rfloor \tag{6}$$

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

The extension (theorem A.2) changes the bound on $h$ from $h \leq R$ to $h \leq R^2$. The added test is simple to implement, and can be done using single precision arithmetic. Assuming the integer passes the exponentiation tests and $h < R^2$, then the only additional test is to check that $\left((h \bmod R)^2 - 4\left\lfloor \frac{h}{R} \right\rfloor\right)$ is not a perfect square.

While $h$, $R$ are generally both multiple precision integers, the test for not being a perfect square can be reduced to single precision arithmetic. 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 3.2 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.

# B Primality Proving and Proof Trees

## B.1 Prockington's prime Prove Example

The tree (figure 1) proves primality of 1103 in the following way:

- 7, and 3 are small primes found/proved using the sieve of Eratosthenes.
- 29 has only one child (7) and is proved using Pocklington's theorem:

1. $2^{28} \equiv 1 \bmod 29$;
2. $\gcd\big(2^{28/7} - 1 \bmod 29, 29\big) = 1$
3. $(28/7) \leq 7$

proving that 29 is prime;

- 19 has only one child (3) and is proved using Pocklington's theorem:

  1. $2^{18} \equiv 1 \bmod 19$;
  2. $\gcd\big(2^{18/3} - 1 \bmod 19, 19\big) = 1$;
  3. $18/3^2 \leq 3^2$

  proving that 19 is prime;

- Finally, 1103 has 2 children $(19, 29)$:

  1. $3^{1102} \equiv 1 \bmod 1103$;
  2. $\gcd\big(3^{1102/19} - 1 \bmod 1103, 1103\big) = 1$;
  3. $\gcd\big(3^{1102/29} - 1 \bmod 1103, 1103\big) = 1$;
  4. $1102/(19 \cdot 29) < 19 \cdot 29$

  proving that 1103 is prime.

## B.2 Pocklington's example with the Prime Proof Protocol

The tree in figure 1 converts to the following list of primes (p) and the number of child primes (n) needed to prove primality.

$$\left[\begin{array}{c} p = 7 \\ n = 0 \end{array}\right] \rightarrow \left[\begin{array}{c} p = 29 \\ n = 1 \end{array}\right] \rightarrow \left[\begin{array}{c} p = 3 \\ n = 0 \end{array}\right] \rightarrow \left[\begin{array}{c} p = 19 \\ n = 1 \end{array}\right] \rightarrow \left[\begin{array}{c} p = 1103 \\ n = 2 \end{array}\right]$$

Here's how the algorithm progresses, starting with $|in| = 5, |out| = 0$ nodes in each list.

1. The first node on the input list is a known prime[2] as is indicated by $n = 0$. After checking that this prime is in the small prime set, it is added to the output list. $|in| = 4,\ |out| = 1$

[2]The known prime list are small primes generated up to a certain bound with the sieve of Eratosthenes.

2. Next, $p = 29$ is read. It has one child, so one prime is read off (and removed) from the output list – $q = 7$. After proving it is prime, 29 is added to the output list. $|in| = 3,\ |out| = 1$

3. 3 is read in (zero children), and checked in the small prime set. After it is verified, it is added to the output list. $|in| = 2,\ |out| = 2$

4. 19 is read in, with one child. 3 read from the output list (note the first-in-last-out access), and used to prove 19 is prime. 19 is added to the output list. $|in| = 2,\ |out| = 2$

5. Finally, $p = 1103$ is read from the input list, with $n = 2$ children. 19 and 29 are read off the output list, used to prove that 1103 is prime, then $p = 1103$ is put onto the output list, and returning the output list. $|in| = 0, |out| = 1$

### B.3 A more complex example

Figure 2 shows a more complex prime proof tree. Leaf primes are small known primes, generally computed with the sieve of Eratosthenes up to some bound. From there, $p_0^{(4)}$ can be proved, then $p_0^{(3)}$. We can't prove $p_0^{(2)}$ is prime until $p_1^{(3)}$ is proved, so the next primes read in will be

1. $p_1^{(4)}$ (small prime),

2. $p_0^{(6)}, p_1^{(6)}$ (two more small primes),

3. $p_1^{(5)}$ (proved using the last two primes),

4. $p_2^{(4)}$ (proved using $p_1^{(5)}$),

5. $p_1^{(3)}$ (proved using $p_1^{(4)}, p_2^{(4)}$),

and finally $p_0^{(2)}$ is proved, using $p_0^{(3)}$ and $p_1^{(3)}$.

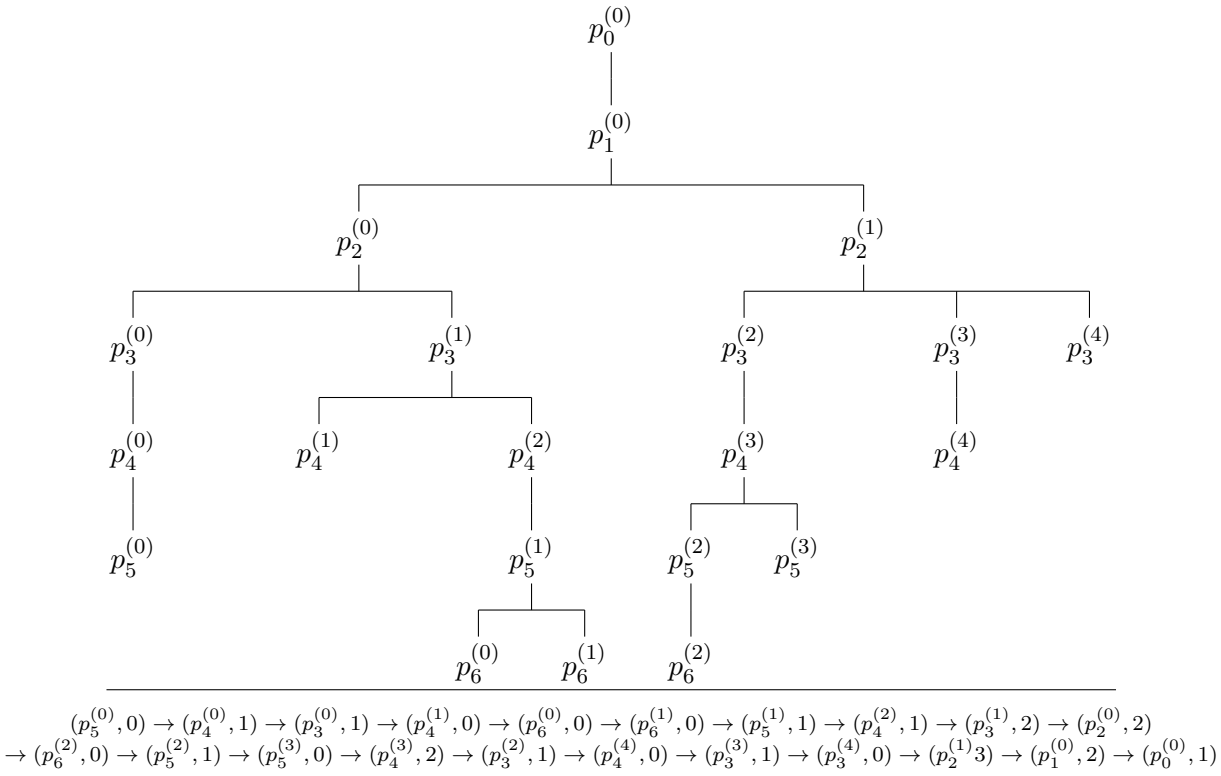


Figure 2: Sample prime proof tree, with the number of children given by the tree structure

## References


[1] M. Agrawal, N. Kayal, and N. Saxena, *Primes in p*, Tech. report, Indian Institute of Technology Kanpur, 2002, Updated document on web: *http://www.cse.iitk.ac.in/users/manindra/primality_v6.pdf*.

[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., ch. 4.1, p. 175, Springer-Verlag, 175 Fifth Avenue, New York, New York 10010, U.S.A., 2005.

[4] Anna M. Johnston, *Designer primes*, Cryptology ePrint Archive, Report 2020/1535, 2020, `https://eprint.iacr.org/2020/1535`.

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

[6] 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.

[7] 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.

[8] 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.

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

[10] M.O. Rabin, *Probabilistic algorithms for testing primality*, Journal of Number Theory **12** (1980), 128–138.

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

[12] Jake Massimo Steven D. Galbraith and Kenneth G. Paterson, *Safety in numbers: On the need for robust diffie-hellman parameter validation*, Public-Key Cryptography (PKC 2019) **11443** (2019), 379–407.