American National Standard for Financial Services

X9.80–2005

Prime Number Generation, Primality Testing, and Primality Certificates

Accredited Standards Committee X9, Incorporated
Financial Industry Standards

Date Approved: August 15, 2005
American National Standards Institute

Foreword

Approval of an American National Standard requires verification by ANSI that the requirements for due process, consensus, and other criteria for approval have been met by the standards developer.

Consensus is established when, in the judgment of the ANSI Board of Standards Review, substantial agreement has been reached by directly and materially affected interests. Substantial agreement means much more than a simple majority, but not necessarily unanimity. Consensus requires that all views and objections be considered, and that a concerted effort be made toward their resolution.

The use of American National Standards is completely voluntary; their existence does not in any respect preclude anyone, whether he has approved the standards or not from manufacturing, marketing, purchasing, or using products, processes, or procedures not conforming to the standards.

The American National Standards Institute does not develop standards and will in no circumstances give an interpretation of any American National Standard. Moreover, no person shall have the right or authority to issue an interpretation of an American National Standard in the name of the American National Standards Institute. Requests for interpretations should be addressed to the secretariat or sponsor whose name appears on the title page of this standard.

CAUTION NOTICE: This American National Standard may be revised or withdrawn at any time. The procedures of the American National Standards Institute require that action be taken to reaffirm, revise, or withdraw this standard no later than five years from the date of approval.

Published by

Accredited Standards Committee X9, Incorporated
Financial Industry Standards
P.O. Box 4035
Annapolis, MD 21403 USA
X9 Online http://www.x9.org

Contents

Tables

Introduction

NOTE The user's attention is called to the possibility that compliance with this standard may require use of an invention covered by patent rights.

By publication of this standard, no position is taken with respect to the validity of this claim or of any patent rights in connection therewith. The patent holder has, however, filed a statement of willingness to grant a license under these rights on reasonable and nondiscriminatory terms and conditions to applicants desiring to obtain such a license. Details may be obtained from the standards developer.

Suggestions for the improvement or revision of this Standard are welcome. They should be sent to the X9 Committee Secretariat, Accredited Standards Committee X9, Inc., Financial Industry Standards, P.O. Box 4035, Annapolis, MD 21403 USA.

This Standard was processed and approved for submittal to ANSI by the Accredited Standards Committee on Financial Services, X9. Committee approval of the Standard does not necessarily imply that all the committee members voted for its approval.

The X9 committee had the following members:
Gene Kathol, X9 Chairman
Vincent DeSantis, X9 Vice-Chairman
Cynthia Fuller, Executive Director
Isabel Bailey, Managing Director

Organization Represented	**Representative**
ACI Worldwide	Jim Shaffer
American Bankers Association	C. Diane Poole
American Express Company	Mike Jones
American Financial Services Association	Mark Zalewski
Bank of America	Daniel Welch
Capital One	Scott Sykes
Certicom Corporation	Daniel Brown
Citigroup, Inc.	Daniel Schutzer
Deluxe Corporation	John Fitzpatrick
Diebold, Inc.	Bruce Chapa
Discover Financial Services	Jennifer Schroeder
Federal Reserve Bank	Dexter Holt
First Data Corporation	Gene Kathol
Fiserv	Bud Beattie
Hewlett Packard	Larry Hines
Hypercom	Scott Spiker
IBM Corporation	Todd Arnold
Ingenico	John Sheets
Intuit, Inc.	Jana Hocker
J.P. Morgan Chase & Co	Jacqueline Pagan
KPMG LLP	Alfred F. Van Ranst

MagTek, Inc.	Carlos Morales
MasterCard International	William Poletti
Ntl. Association of Convenience Stores	Teri Richman
National Security Agency	Sheila Brand
NCR Corporation	David Norris
SWIFT/Pan Americas	Malene McMahon
The Clearing House	Vincent DeSantis
Unisys Corporation	David J. Concannon
University Bank	Stephen Ranzini
VeriFone, Inc.	Brad McGuinness
VECTORsgi	Ron Schultz
VISA	Richard Sweeney
Wachovia Bank	Ray Gatland
Wells Fargo Bank	Ruven Schwartz

The X9F subcommittee on Data and Information Security had the following members:

Richard J. Sweeney, Chairman

Organization Represented	**Representative**
3PEA Technologies, Inc.	Mark Newcomer
ACI Worldwide	Jim Shaffer
American Bankers Association	C. Diane Poole
American Express Company	Mike Jones
American Financial Services Association	Mark Zalewski
Bank of America	Mack Hicks
Capital One	Scott Sykes
Certicom Corporation	Daniel Brown
Citigroup, Inc.	Paul Gubiotti
Deluxe Corporation	John Fitzpatrick
DeLap, White, Caldwell and Croy, LLP	Darlene Kargel
Diebold, Inc.	Bruce Chapa
Entrust, Inc.	Robert Zuccherato
Federal Reserve Bank	Neil Hersch
Ferris and Associates, Inc.	J. Martin Ferris
Fidelity Investments	Michael Versace
First Data Corporation	Gene Kathol
First National Bank of Nebraska, Inc.	Lisa Curry
Fiserv	Bud Beattie
Futurex	Jason Anderson
Hewlett Packard	Larry Hines
Hypercom	Scott Spiker
IBM Corporation	Todd Arnold
Identrus	Brandon Brown
InfoGard Laboratories	Tom Caddy
Ingenico	John Sheets
J.P. Morgan Chase & Co	Edward Koslow
KPMG LLP	Alfred F. Van Ranst

MagTek, Inc.	Terry Benson
MasterCard International	Ron Karlin
Microsoft Corp	Niels Ferguson
Ntl. Association of Convenience Stores	Teri Richman
National Inst. of Stds and Technology	Elaine Barker
National Security Agency	Sheila Brand
NCR Corporation	David Norris
NTRU Cryptosystems, Inc.	William Whyte
Orion Security Solutions	Miles Smid
Pi R Squared Consulting LLP	Ralph Poore
Pitney Bowes, Inc.	Leon Pintsov
Proofspace	Paul F. Doyle
RSA Security, Inc.	James Randall
Surety, Inc.	Dimitrios Andivahis
TECSEC Incorporated	Ed Scheidt
Thales e-Security, Inc.	James Torjussen
Triton Systems of Delaware, Inc.	Daryll Cordeiro
University Bank	Stephen Ranzini
VeriFone, Inc.	Dave Faoro
VECTORsgi	Ron Schultz
VISA	Richard Sweeney
Wachovia Bank	Ray Gatland
Wells Fargo Bank	Ruven Schwartz

Under ASC X9 procedures, a working group may be established to address specific segments of work under the ASC X9 Committee or one of its subcommittees. A working group exists only to develop standard(s) or guideline(s) in a specific area and is then disbanded. The individual experts are listed with their affiliated organizations. However, this does not imply that the organization has approved the content of the standard or guideline. (Note: Per X9 policy, company names of non-member participants are listed only if, at time of publication, the X9 Secretariat received an original signed release permitting such company names to appear in print.)

The X9F1 Cryptographic Tool Standards and Guidelines group that developed this standard had the following members:

Miles Smid, Chairman
James Randall, Project Editor

Organization Represented	**Representative**
Certicom Corporation	Dan Brown Scott Vanstone Simon Blake-Wilson
Communications Security Establishment of Canada	Mike Chawrun
Entrust	Don Johnson Robert Zuccherato
HP	Susan Langford
IBM Corporation	Alan Roginsky
Microsoft	Niels Furguson

National Institute of Standards and Technology	Morris Dworkin Elaine Barker John Kelsey
National Security Agency	Paul Timmel Michael Boyle William Whyte
NTRU	Miles Smid
ORION	Ralph Poore
Pi R Squared	Matt Compagna
Pitney Bowes, Inc	James Randall
RSA Security	Burt Kaliski Steve Schmalz

Business practice has changed with the introduction of computer-based technologies. The substitution of electronic transactions for their paper-based predecessors has reduced costs and improved efficiency. Trillions of dollars in funds and securities are transferred daily by telephone, wire services, and other electronic communication mechanisms. The high value or sheer volume of such transactions within an open environment exposes the financial community and its customers to potentially severe risks from accidental or deliberate alteration, substitution, or destruction of data. This risk is compounded by interconnected networks, and the increased number and sophistication of malicious adversaries.

Some of the conventional “due care” controls used with paper-based transactions are unavailable in electronic transactions. Examples of such controls are safety paper, which protects integrity, and hand-written signatures or embossed seals, which indicate the intent of the originator to be legally bound. In an electronic-based environment, controls must be in place that provide the same degree of assurance and certainty as in a paper environment. The financial community is responding to these needs.

The Accredited Standards Committee on Financial Services (ANSI X9) has developed several sets of standards based on public key cryptography to protect financial information:

- X9.30-1996, *Public Key Cryptography Using Irreversible Algorithms for the Financial Services Industry* contains

  Part 1: *The Digital Signature Algorithm (DSA)* and

  Part 2: *The Secure Hash Algorithm -1 (SHA-1)*.

- X9.31-1998, *Digital Signatures Using Reversible Public Key Cryptography for the Financial Services Industry (rDSA)*

- X9.42-2000, *Public Key Cryptography for the Financial Services Industry – Agreement of Symmetric Keys Using Discrete Logarithm Cryptography*

- X9.62-1998, *Public Key Cryptography for the Financial Services Industry - The Elliptic Curve Digital Signature Algorithm (ECDSA)*

- X9.63-2002, *Public Key Cryptography for the Financial Services Industry – Key Agreement and Transport Using Elliptic Curve Cryptography*

This Standard, *Prime Number Generation, Primality Testing, and Primality Certificates*, defines techniques for generating prime numbers that are needed as parameters in public key algorithms.

The use of this Standard, together with appropriate controls, may have considerable legal effect with respect to the apportionment of liability for erroneous or fraudulent transactions and the satisfaction of requirements for transaction

enforceability. The legal implications associated with the use of this Standard may have their origin in both case law and legislation, including the Uniform Commercial Code Article 4A on Funds Transfers (Article 4A).

The details of Article 4A address (in part) the implementation of commercially reasonable security procedures and the effect of using such procedures on the apportionment of liability between a customer and a bank. A security procedure is used by Article 4A-201 "for the purpose of (i) verifying that a payment order is that of the customer, or (ii) detecting error in the transmission or the content of the payment order or communication." The commercial reasonableness of a security procedure is determined by the criteria established in Article 4A-201.

Prime Number Generation, Primality Testing, and Primality Certificates

1 Scope

In the current state of the art in public key cryptography, all methods require, in one way or another, the use of prime numbers as parameters to the various algorithms. This document presents a set of accepted techniques for generating primes.

It is intended that ASC X9 standards that require the use of primes will refer to this document, rather than trying to define these techniques on a case-by-case basis. Standards, as they exist today, may differ in the methods they use for parameter generation from those specified in this document. It is anticipated that as each existing ASC X9 standard comes up for its 5-year review, it will be modified to reference this document instead of specifying its own techniques for generating primes.

This standard defines methods for generating large prime numbers as needed by public key cryptographic algorithms. It also provides testing methods for testing candidate primes presented by a third party.

This standard allows primes to be generated either deterministically or probabilistically, where:

— A number shall be accepted as prime when a probabilistic algorithm that declares it to be prime is in error with probability less than 2^{-100}.

— A deterministic prime shall be generated using a method that guarantees that it is prime.

In addition to algorithms for generating primes, this standard also presents primality certificates for some of the algorithms where it is feasible to do so. The syntax for such certificates is beyond the scope of this document. Primality certificates are never required by this standard. Primality certificates are not needed when a prime is generated and kept in a secure environment that is managed by the party that generated the prime.

A requirement placed upon the use of this standard, but out of scope, is as follows:

— When a random or pseudo-random number generator is used to generate prime numbers, an ANSI approved random number (or bit) generator (i.e., one that is specified in an ANSI X9 standard) shall be used. This requirement is necessary to ensure security.

NOTE—The 2^{-100} failure probability is selected to be sufficiently small that errors are extremely unlikely ever to occur in normal practice. Moreover, even if an error were to occur when one party tests a prime, subsequent tests by the same or other parties would detect the error with overwhelming probability. Furthermore, the 2^{-100} probability is an upper bound on the worst-case probability that a test declares *any* non-prime candidate to be prime; not all non-primes may reach this bound, and the probability that a non-prime generated at random passes such a test is much lower. Accordingly, the 2^{-100} bound is considered appropriate independent of the size of the prime being generated and the intended security level of the cryptosystem in which the prime is to be employed. For high-assurance applications, however, the deterministic methods may nevertheless be preferable.

2 Normative references

The following referenced documents are indispensable for the application of this document. For dated references, only the edition cited applies. Nevertheless, parties to agreements based on this document are encouraged to consider applying the most recent edition of the referenced documents indicated below. For undated references, the latest edition of the referenced document (including any amendments) applies.

X9.62-1998, *Public Key Cryptography for the Financial Services Industry – The Elliptic Curve Digital Signature Algorithm (ECDSA)*©.

X9.63-2002, *Public Key Cryptography for the Financial Services Industry – Key Agreement and Transport Using Elliptic Curve Cryptography*

3 Terms and definitions

For the purposes of this document, the following terms and definitions apply.

3.1
Composite
An integer that has at least two prime factors (two or more may be identical).

3.2
ECPP
Acronym for Elliptic Curve Primality Proving algorithm.

3.3
***m*-bit number**
A positive integer whose binary representation consists of *m* bits, where the high order bit, by definition, is always a "1". In the case of an *m*-bit prime number, the low order bit is also a "1" except for the 2-bit prime number decimal 2, which has the binary value b'10'.

For example, the two-byte hexadecimal prime number x'01FD' (decimal 509) is the 9-bit prime number b'111111101'.

3.4
Lehmer-Kraitchik Algorithm
A deterministic algorithm that may be used to rigorously prove the primality of an integer.

3.5
Lucas Probable Prime[1]
An integer that has been declared to be a prime number after applying the Lucas test. This number may, however, actually be composite with low probability.

3.6
Miller-Rabin Probable Prime
An integer that has been declared to be a prime number after applying the Miller-Rabin (M-R) test. This number may, however, actually be composite with low probability.

3.7
Monic Polynomial
A polynomial whose lead coefficient is 1.

---

[1] Formerly called "Lucas-Lehmer Probable Prime" in ANS X9.80–2001.

3.8
Polynomial Discriminant
The squared product of the pairwise difference of roots of the polynomial.

3.9
Prime Certificate
A certificate containing a small set of numbers associated with an integer that can be used to rigorously prove that the integer is prime. Certificates can be independently checked by other software.

3.10
Prime Number
An integer that is greater than 1, and divisible only by 1 and itself.

3.11
Primitive Root
An integer *a* is a primitive root with respect to a prime *p* if the smallest integer value of *x,* such that $a^x \equiv 1 \pmod{p}$, is $x = p–1$.

3.12
Private Key
In an asymmetric (public) key cryptosystem, that key of an entity's key pair which is known only by that entity.

3.13
Probable Prime
An integer that has been declared prime via a probabilistic method.

3.14
Pseudo-Random Number
A single number taken as a sample from a sequence of numbers produced by a pseudo-random number generator.

3.15
Public Key
In an asymmetric key system, that key of an entity’s key pair which may be publicly known.

3.16
Public-Key Cryptography
An asymmetric cryptographic algorithm that uses two related keys, a public key and a private key; the two keys have the property that, given the public key, it is computationally infeasible to derive the private key.

3.17
Sharp Estimate
An estimate for a variable or mathematical quantity is said to be sharp when the estimate is close to the true value of the quantity being estimated.

3.18
Sieve
A computation technique that identifies integers in an arithmetic progression that have small prime factors.

3.19
Trial Division
A procedure whereby an integer is tested to see if it is divisible by small primes by performing the actual division and seeing if the remainder is zero.

3.20
Uniform Random Variable
A real number *x*, lying within a pre-specified interval [*a, b*] such that the probability that *x* lies in a sub-interval [*c, d*] with $c \geq a$ and $d \leq b$ is equal to $(d–c)/(b–a)$.

4 Symbols and abbreviated terms

4.1
$\lfloor x \rfloor$
Floor function of *x*; the largest integer ≤ x. For example, $\lfloor 5.3 \rfloor = 5$, and $\lfloor -1.5 \rfloor = -2$.

4.2
$\lceil x \rceil$
Ceiling function of *x*; the smallest integer ≥ x. For example, $\lceil 5.3 \rceil = 6$ and $\lceil -1.5 \rceil = -1$.

4.3
$\sqrt{x}$
Square root of x.

4.4
$\left(\frac{a}{n}\right)$
Jacobi symbol of *a* with respect to *n*. See Annex A.2.

4.5
a*|*b
Evenly divides; e.g. *a* divides *b* evenly (with no remainder).

4.6
+
Addition.

4.7
≡
Congruence. $A \equiv B \pmod{C}$ means that $(A–B)$ is divisible by *C*.

4.8
a^b
Exponentiation. *a* raised to the b^{th} power.

4.9
.
Multiplication.

4.10
$|x|$
Absolute value of *x*; $|x|$ is $–x$ if $x < 0$; otherwise, it is simply *x*.

4.11
[*a,b*]
The set of real numbers from *a* to *b* inclusive.

4.12
b'01'
An example of binary notation used to represent one or more bits.

4.13
CRT
Chinese Remainder Theorem.

4.14
GCD (*a*, *b*)
Greatest common divisor of integers *a* and *b*.

4.15
LCM (*a*, *b*)
Least common multiple of integers *a* and *b*.

4.16
mod
Modulo.

4.17
modulo *n*
Arithmetic modulo *n*.

4.18
m, n, a
Any positive integer that may or may not be prime.

4.19
p, q
Prime numbers.

NOTES

1. All integers (and all strings of bits or bytes) are written with the most significant digit (or bit or byte) in the left position.

2. Numbers represented by powers of 2, e.g. 2^x, are $x + 1$ bit numbers.

5 Prime Generation Methods

5.1 General Discussion

This section discusses prime generation by a first party; i.e. generation of primes by the user of said primes.

Prime numbers may be generated by a variety of methods. The methods are categorized into two groups. The first group generates candidate integers at random and then tests them for primality using either a probabilistic or deterministic algorithm. These methods are discussed in Section 5.2. The second set of methods actually constructs integers from smaller integers in such a way that the constructed integer is guaranteed to be prime. These methods are discussed in Section 5.3.

Section 5.4 discusses how to generate primes subject to side conditions. A typical side condition might be that a prime *p* must have *p*–1 divisible by a large prime factor. ANSI X9.31 requires, for example, that *p*–1 must have a prime factor of at least 100 bits.

The algorithms approved by this standard include the following:

1. Testing of Randomly Chosen Integers

    1A. Probabilistic Methods for Testing Integers

        — Miller-Rabin, Section 5.2.3.1

        — Lucas, Section 5.2.3.2

        — Frobenius-Grantham, Section 5.2.3.3

    1B. Deterministic Methods for Testing Integers

        — Trial division (for sufficiently small primes, e.g. up to 10 decimal digits), Section 5.2.4.1.1

        — The Selfridge extensions to PPL (Proth, Pocklington, & Lehmer), Section 5.2.4.1.2

        — Kraitchik-Lehmer, Section 5.2.4.1.3

        — ECPP (Elliptic Curve Primality Proof), Section 5.2.4.2

2. Methods for Direct Construction of Prime Numbers

        — Maurer's Algorithm, Section 5.3.1

        — Shawe-Taylor's Algorithm, Section 5.3.2

A method for generating random primes over an interval [*a*, *b*] that gives an equal probability of selecting each prime, is simply to generate a random integer in [*a*, *b*], then test it for primality. If it is not prime, then continue generating random integers and testing them until a prime is found. This method is much too slow to be of practical use, but it is allowed by this standard. Instead, a method is given in Section 5.2.1 that selects primes almost uniformly at random, but is significantly faster than the method suggested above.

Candidate primes can be tested for primality using either a probabilistic or a deterministic algorithm. The difference between these two is that a deterministic algorithm takes a candidate integer and proves whether the candidate is prime, whereas a probabilistic algorithm only declares that the candidate is probably prime. For each invocation of the algorithm, there is a small probability that it will erroneously declare a composite integer to be prime. For this reason, probabilistic algorithms are run more than once in order to ensure that the probability of error is sufficiently small. This standard defines that probability to be 2^{-100}.

This standard allows three different probabilistic procedures to be used. The first is to use at least 50 rounds of the Miller-Rabin algorithm (see Section 5.2.3.1). The second is using multiple rounds of the Miller-Rabin algorithm followed by a single round of the Lucas algorithm (see Section 5.2.3.2). Composite integers that fool multiple rounds of Miller-Rabin are known to exist. However, there is no known composite integer that passes even a single Miller-Rabin test to the base 2 followed by a single Lucas test. This combination was suggested by Pomerance, Selfridge and Wagstaff [11]. While heuristics suggest that such integers do exist, they must be exceedingly rare, and no example is known. Indeed, a prize offered in [11] for such a composite remains unclaimed. In previous X9 standards, only the use of multiple rounds of Miller-Rabin was specified as an acceptable method. However, including a single follow-up round of Lucas lowers the probability of error dramatically. The number of rounds required for Miller-Rabin used in conjunction with Lucas is given in Table 2 (Section 7). This second method is expected to have better performance than the first, but is more complex and takes more code.

The third probabilistic procedure is due to Grantham [6]. It is slightly more complicated to code than Miller-Rabin and Lucas. Each iteration takes about 3 times as long, but it has the advantage that the probability of failure for each iteration of Frobenius-Grantham (see Section 5.2.3.3) is considerably lower than a single iteration of Miller-Rabin. The number of rounds required for Frobenius-Grantham is given in Table 3 (Section 7). At the time of publication of this standard, unlike for Miller-Rabin, there is no known mathematical method for constructing integers that achieve the worst-case probability of error for this test.

Damgaard, Landrock, and Pomerance [5] provide upper bounds on the probability of a failure of multiple rounds of Miller-Rabin based upon the size of the number being tested, assuming that the candidate was randomly chosen. These results are given in Table 2. Similar results have yet to be computed for the Frobenius-Grantham method. However, based upon the probability of a single iteration failing (which can be derived from [6]), along with the upper bound per iteration probability of failure, a crude upper bound table for the number of required rounds of Frobenius-Grantham can be constructed. This is provided in Table 3. It is not a sharp estimate in the sense that the actual probabilities of failure are much lower than given by the table. When the results of [5] are extended to the Frobenius-Grantham method, Table 3 can be revised.

5.2 Generation of Primes Using Random Integers

5.2.1 Generation of Random Primes with Sequential Search

To generate a prime in the interval [*a,b*], first generate an integer *x* uniformly at random in this interval. Then, apply a *sieve procedure* (Annex A.3) to the sequence of integers *x*, *x* + 1, *x* + 2, *x* + 3, *x* + 4, … *x* + *k* for $k = \lfloor 5 \cdot \log(b) \rfloor$ where log is the natural log. The probability that any given integer in this sequence is prime is approximately 1/log(*b*), so selecting an interval whose length is 5 · log(*b*) should be sufficient to find a prime. If no prime is found, then increase the value of *k*. In the very rare event that x+k is outside the desired range, discard x and start over.

The sieve procedure removes all integers in the sequence that are divisible by small primes and hence cannot be prime. Then, remaining integers are incrementally tested with a prime- testing algorithm.

Because this is an incremental search procedure and because the gaps between prime numbers are not uniformly distributed, this method does not select all primes in the interval [*a,b*] with equal probability. For example, there exist many *prime pairs*, integers *p* and *p* + 2, that are both prime. Searching incrementally from the random starting point *x* as outlined above is more likely to select the prime *p* than the prime *p* + 2 (Annex B.2). As stated in 5.1, an alternative method is to randomly choose from among the integers that survive the sieve as candidates for testing

The following algorithm accomplishes the sequential search technique described above. In step 2 two different interval options are provided. ($[(\sqrt{2})(2^{d-1}), (2^d - 1)]$ is included to ensure support for legacy systems)

1. Determine the size *d*, in bits, of the prime to be generated.

2. Select an integer *x* at random from the interval $[2^{d-1}, (2^d - 1)]$. or from the interval $[(\sqrt{2})(2^{d-1}), (2^d - 1)]$.

3. Apply the sieve procedure from Annex A.3 with $Y_0 = x$, and $J = 5 \cdot \log x$.

4. For each successive integer starting at *x* that survives the sieve, or for a randomly chosen integer that survives the sieve, test it for primality by:

   (4a) Applying at least 50 rounds of the Miller-Rabin test.

   or

   (4b) Applying an appropriate number of rounds of the Miller-Rabin algorithm as specified by Table 2, followed by a single Lucas test;

   or

(4c) Applying an appropriate number of rounds of the Frobenius-Grantham algorithm as specified by Table 3;

or

(4d) Applying the methods of Section 5.2.4.1 (if the number is small enough; typically at most 150 bits);

or

(4e) Applying the ECPP method as described in Section 5.2.4.2.

If the number is declared prime, STOP. Otherwise repeat Step 4. If no primes have been found, lengthen the sieve interval by increasing *J* and then repeat Step 3. *J* should be increased by at least log *x* and at most 5 log *x* each time. The exact amount of the increase is not critical. In the very rare event that candidate prime is outside the desired range, start over

Note: The choice of the actual testing method used is left to the implementer. Only one testing method is required

Example. Take $d = 32$. x is chosen randomly as 3439601168. Sieving with primes less than 30 between x and $x + 40$ leaves the following survivors: $x + 15$, $x + 21$, $x + 29$, $x + 33$, and $x + 35$. All other numbers are divisible by a prime less than 30. Testing the survivors with Miller-Rabin reveals that $x + 29 = 3439601197$ is probably prime. This is the first prime greater than x. This can also be confirmed by trial division with all primes up to 58649 (see 5.2.4.1.1).

5.2.2 Generation of Random Primes with Uniform Distribution

As an alternative to the incremental search procedure given above, one may also do the following:

1. Determine the size *d*, in bits, of the prime to be generated.

2. Select an odd integer *x* at random from the interval $[2^{d-1}, (2^d - 1)]$ or from the interval $[(\sqrt{2})(2^{d-1}), (2^d - 1)]$.

3. Test *x* for primality using any method approved by this standard. If *x* is not prime, then go to Step 2. If *x* is prime, then STOP.

NOTE The probability that a randomly chosen integer near x is prime is about 1/log *x*. Therefore, Step 2 will select about (log *x*)/2 candidates on average, before a prime is found.

5.2.3 Testing Using Probabilistic Methods

5.2.3.1 Miller-Rabin[2]

The Miller-Rabin algorithm to test a candidate *p is* as follows:

1. Select an integer *a* uniformly at random from [2, $p - 2$] such that GCD(*a, p*) = 1. The successive small primes 2, 3, 5, 7, 11, …, may also be used for *a* for successive invocations of this test. This gives a slight speed advantage. However, please refer to Section 6 for instances when using 2, 3, 5, 7, 11, …, is not appropriate.

2. Compute the largest integer *t* and *s* such that $p - 1 = 2^t s$. It follows that *s* will be odd.

3. Compute $y = a^s \bmod p$ using the method of Annex A.1.

---

[2] This specification is a technical improvement over the specification in ANS X9.80–2001. The 2001 specification will work in almost all cases, although this specification is preferred.

4. If $y = 1$ or $y = p - 1$, then declare 'p is probably prime' and exit.

5. For $i = 1$ to $t - 1$

   i) Let $y = y^2 \bmod p$.

   ii) If $y = p - 1$, then declare 'p is probably prime' and exit.

   iii) If $y = 1$, then declare 'p is composite' and exit.

6. Return 'p is composite'

Example. Let $p = 3439601197$. Then $p - 1 = 2^2 \cdot 859900299$, so $t = 2$ and $s = 859900299$. The successive primes 2, 3, 5, 7, … are chosen as the bases for the tests, but the computations are shown only for base 2. The modular exponentiation procedure given in Annex A.1 gives $y = 2^s \equiv 1379737433 \pmod{p}$ for Step 3. Since y is not 1 or $p - 1$, proceed to Step 5. Since $t - 1 = 1$, perform one iteration of this loop. Then, $y^2 \equiv 3439601196 \equiv -1 \pmod{p}$, so declare '$p$ is probably prime', then proceed to similar computations for base 3, 5, 7, 11….. Note that for base 3, $3^s \equiv 1 \pmod{p}$ in Step 4, so it is not necessary to execute Step 5. Similarly for base 7, $7^s \equiv -1 \pmod{p}$ in Step 4, so there is no need to execute Step 5.

5.2.3.2 Lucas[3]

The Lucas algorithm to test a candidate p is as follows:

1. Test if p is an exact square. If so, return 'p is composite' and exit.

2. Find the first D in the sequence {5, –7, 9, –11, 13, –15, 17, …} for which the Jacobi symbol $\left(\frac{D}{p}\right) = -1$. See Annex A.2 for a method to compute the Jacobi symbol. The probability that any given value of D will yield –1 for the Jacobi symbol is about ½. If for any D in the sequence the Jacobi symbol is 0, declare 'p is composite' and exit.

3. Compute $U_{p+1} = \text{Lucas}(D, p + 1, p)$. Annex A.5 defines the subroutine Lucas and shows how to compute Lucas(D, $p + 1$, p).

4. If $U_{p+1} \equiv 0 \pmod{p}$, then p is a Lucas probable prime.

Example. Select $p = 3439601197$. A computation of the Jacobi symbol reveals that $\left(\frac{D}{p}\right) = -1$ for $D = 5$. The computation of Lucas(D, $p + 1$, p) returns 0. Since the final value of $U_{p+1} \bmod p$ is 0, declare p to be a Lucas probable prime. Annex A.5 shows some of the internal calculations of the Lucas sequence.

NOTES

1—The motivation for Step 1 is that if p is an exact square, the Jacobi symbol in Step 2 will always be +1 or 0, never –1. Step 1 may equivalently be applied after some number of values of D have been tried unsuccessfully in Step 2. When performing this Lucas test as part of 5.2.1 Step (4b), it is highly unlikely that an exact square in Step 1 or a value of D with Jacobi symbol 0 in Step 2 will ever be encountered.

2—There are several "Lucas tests" in the literature. This version is described by Baillie and Wagstaff [2].

---

[3] Formerly called "Lucas-Lehmer" in ANS X9.80–2001. This specification is a technical improvement over the specification in ANS X9.80–2001. The 2001 specification will work in almost all cases, although this specification is preferred.

5.2.3.3 Frobenius-Grantham

Algorithms for performing the polynomial arithmetic required by this method are given in Annex A.4.

Let $f(x)$ be a low-degree monic polynomial. The selection of $f(x)$ is discussed below. Let d be the degree of $f(x)$, and let D be its discriminant. p is the integer being tested. To compute the discriminant, see below. Perform the following steps:

1. Confirm that GCD(p, $D \cdot f(0)$) = 1. If not, return 'p is composite' and exit.

2. Set $f_0(x) = f(x) \bmod p$.

3. For i = 1 to d

   i) Let $F_i(x) = \text{GCD}(x^{p^{\wedge}i} - x, f_{i-1}(x))$

   ii) Let $f_i(x) = f_{i-1}(x)/F_i(x)$

   iii) If i does not divide the degree of $F_i(x)$, declare 'p is composite' and exit.

   iv) If $f_d(x)$ does not equal 1, declare p is composite and exit.

4. For i = 2 to d

   Compute $F_i(x^p) \bmod F_i(x)$. If the result is non-zero for any i, declare 'p is composite' and exit.

5. Compute $S = \sum \frac{\text{degree}(F_i(x))}{i}$ for $i \le d$ and i even.

6. If S does not equal $\frac{1}{2}\left(\left(\frac{D}{p}\right) - 1\right) \bmod 2$, then declare '$p$ is composite' and exit.

7. Declare 'p is probably prime' and exit.

It is suggested that $f(x)$ should be a quadratic polynomial $f(x) = (x - a)(x - b)$ or $f(x) = x^2 - cx - e$. It is true that in the first case, if p is a Frobenius-Grantham probable prime, then it is also an ordinary probable prime to the bases a and b. If $f(x) = x^2 - cx - e$, then p will also be a Lucas probable prime. The Frobenius-Grantham test subsumes the ordinary probable prime test, the Lucas, and several other tests as well (which are beyond the scope of this document). Higher degree polynomials are also permitted, but are beyond the scope of this document. It is assumed that any implementer who wishes to use them will have sufficient mathematical background to know how to compute their discriminants.

The discriminant of a linear polynomial is 1. The discriminant of $x^2 - cx - e$ is $c^2 + 4e$. The discriminant of $(x - a)(x - b)$ is $(a + b)^2 - 4ab$.

Multiple rounds of this algorithm are executed by selecting different polynomials $f(x)$. A polynomial may be randomly selected by first fixing its degree, then choosing its coefficients as random integers taken modulo p.

Example. Take p = 3439601197. Select $f(x) = x^2 + x + 1$. Then, d = 2 and the discriminant is D = –3. The Jacobi symbol $\left(\frac{D}{p}\right)$ is 1.

1. Confirm that GCD(p, $-3 \cdot f(0)$) = 1.

2. $f_0(x) = f(x) = x^2 + x + 1$.

3. $i = 1$, $F_1(x) = \text{GCD}(x^p - x, f_0(x)) = x^2 + x + 1$ by the methods in Annex A.4.

   Then $f_1(x) = (x^2 + x + 1)/ (x^2 + x + 1) = 1$.

   $i = 1$ divides $d = 2$, so continue.

   $i = 2$, $F_2(x) = \text{GCD}(x^{p\wedge 2} - x, 1) = 1$. The methods of Annex A.4 need not be applied here, because $f_1(x) = 1$. This condition is a fairly common occurrence and is worth coding.

   Then $f_2(x) = 1$. The degree of f_2 (a constant polynomial) is 0.

   $i = 2$ divides 0, so continue.

4. For $i = 2$, $F_2(x^p) \bmod F_2(x) = 1 \bmod 1 = 0$, so continue.

5. For $i = 2$, $S = \text{degree}(F_2)/2 = 0/2 = 0$.

6. Now, $\frac{1}{2}\left(\left(\frac{D}{p}\right) - 1\right) \bmod 2 = 1/2\ (1 - 1) \bmod 2 = 0$. Since this equals S, declare p to be a Frobenius-Grantham probable prime and exit.

5.2.4 Testing Using Deterministic Methods

5.2.4.1 Classical Methods

This section discusses techniques for rigorously proving primality. They are very efficient for small primes (say around 100 bits) and sometimes work well for larger primes. They can also work well when there are known large prime factors of $p - 1$ and/or $p + 1$ such as the case when constructing strong primes. (See Section 5.4)

These methods usually succeed within milliseconds for primes up to 100 bits on a typical fast PC. When these methods succeed, they also give a small certificate of primality.

It is important that there is a high degree of certainty that the number being tested is prime before trying any of these methods. If a candidate really is not prime, then both PPLS (Section 5.2.4.1.2) and KL (Section 5.2.4.1.3) may not terminate.

5.2.4.1.1 Trial Division

Trial division is the oldest of prime testing methods. An integer is proved prime by showing that it has no prime factors less than or equal to its square root. This procedure is not recommended for testing any integers longer than 10 digits.

To prove p is prime:

1. Prepare a table of primes less than or equal to $\sqrt{p}$. This can be done by applying the sieve procedure in Annex A.3.

2. Divide p by every prime in the table. If p is divisible by one of the primes, then declare that 'p is composite' and exit. It is also allowed to divide p by composite numbers, if it is convenient to do so. For example, rather than preparing a table of primes, it might be more convenient to divide by all integers except those divisible by 2, 3 or 5 (as well as 2, 3, or 5 themselves, of course).

3. Otherwise, declare that 'p is prime' and exit.

Example. For $p = 3439601197$, then $\sqrt{p} \approx 58648.11$. p is not divisible by any prime less than or equal to 58648, so p is prime.

5.2.4.1.2 Proth-Pocklington-Lehmer-Selfridge (PPLS)

The second method is an extension of a method known as PPL (Proth, Pocklington, and Lehmer). The extension is due to John Selfridge. See Section B3 in [2]. Verify that p passes at least one iteration of Miller-Rabin, so that there is some confidence that p is prime before applying this test.

Suppose p is the prime to be verified. Suppose there exists a partial factorization of $p - 1$ and $p + 1$ as follows:

$$p - 1 = F_1 \cdot r$$

$$p + 1 = F_2 \cdot s.$$

F_1 is the product of some known prime factors of p–1; F_2 is the product of some known prime factors of $p + 1$; and r and s have no prime factors less than B_1 and B_2 respectively, where B_1 and B_2 are integers chosen by the implementer, and F_1 and F_2 are completely factored. The choice of values for B_1 and B_2 is left to the implementer. See the note below for reasonable choices. If B_1 and B_2 are chosen as large as $p^{1/3}$, then this method will always work, but it will be very slow for p greater than about 60 bits. If B_1 and B_2 are chosen smaller than $p^{1/3}$, then this method will not always work; the inequality in Step 1 may not be satisfied.

1. Let

   $$G = \max(B_1 F_1, B_2 F_2 - 1).$$

   If $p < G B_1 B_2 F_1 F_2 /2$, and the conditions in Step 2 and Step 3 hold, then p is prime.

2. p does not divide $U_h = \text{Lucas}(D, h, p)$, where $h = (p + 1)/q$ for each prime q dividing F_2. D is an element in the sequence $\{5, -7, 9, -11, 13, -15, 17, \ldots\}$ for which the Jacobi symbol $\left(\frac{D}{p}\right) = -1$. A different value of D may be chosen for each prime q when computing U_h. It is sufficient to find just one value of D that works for each prime. If p does divide U_h for some choice of D, continue trying other values of D.

3. $\text{GCD}(a^{(p-1)/q} - 1, p) = 1$ for all primes q dividing F_1 for some value of a that is co-prime to p and that can be different for each value of q.

If any of the conditions cannot be met, then the method fails. If this happens, either increase B_1 and B_2 and repeat the calculations or use some other method. Failure of the method does not imply that p is composite.

This test works any time that the product of all the known prime factors for $p - 1$ and $p + 1$ taken together exceeds $p^{1/3}$. This test works very quickly for numbers up to about 35 decimal digits because it is easy to find enough factors of $p \pm 1$. For larger numbers, it will sometimes work quickly, depending on whether enough small factors of $p \pm 1$ can be found. Finding enough factors of $p \pm 1$ is what makes the algorithm slow for large values of p.

A certificate for this primality proof consists of the following information:

B_1, B_2, G, all of the prime factors of F_1 and F_2 and the values of a and D corresponding to the primes dividing F_1 and F_2.

Example. For $p = 3439601197$, take $B_1 = B_2 = 30000$ and then:

$$p - 1 = 2^2 \cdot 3 \cdot 286633433$$

$$p + 1 = 2 \cdot 11 \cdot 19 \cdot 127 \cdot 64793$$

with F_1 = 12, F_2 = 53086, r = 286633433, and s = 64793. Then G = max(360000,1592579999) = 1592579999. Also, $p < G\ B_1\ B_2\ F_1\ F_2/2$ and r and s have no factors less than 30000.

Now compute U_h for each of $h = (p + 1)/2$, $h = (p + 1)/11$, $h = (p + 1)/19$, and $h = (p + 1)/127$.

For q = 2, it is determined that D = 5 has $\left(\frac{D}{p}\right) = -1$, but $U_h \equiv 0 \pmod{p}$. This does not satisfy the requirement of Step 2, so continue trying other values of D. For D = –19, $\left(\frac{D}{p}\right) = -1$, and p does not divide U_h = 1430646294.

For q = 11, and for D = 5, p does not divide U_h = 1913300486.

For q = 19, and for D = 5, p does not divide U_h = 2036784421.

For q = 127, and for D = 5, p does not divide U_h = 2603656754.

Now, try to satisfy the requirement of Step 3. Determine that

For q = 2, $2^{(p-1)/2} \equiv -1 \pmod{p}$, and GCD(–2, p) = 1.

For q = 3, $2^{(p-1)/3} \equiv$ 1346083919 and GCD(1346083918, p) = 1.

All of the requirements in Step 1, Step 2, and Step 3 have been satisfied, so p has been *proved* prime. A certificate consists of (30000, 30000, 1592579999, [(2, –19), (11,5), (19, 5), (127,5)], [(2, 2), (3,2)]). The pairs of numbers in [] are the primes dividing F_2 with the corresponding values of D, and followed by the primes dividing F_1 with the corresponding values of a.

NOTES

1. A good value to choose for B_1 and B_2 is about $p^{1/6}$. If the method does not work, these values can be increased.

2. This algorithm can be used recursively. If a factorization of $p + 1$ or $p - 1$ reveals that it is the product of some small primes times a large probable prime, then this algorithm can be applied recursively to that large probable prime cofactor. This would then give a full factorization of either $p + 1$ or $p - 1$, which in turn would allow the proof to be completed.

5.2.4.1.3 Kraitchik-Lehmer

Another deterministic testing technique is due to Kraitchik and Lehmer. Suppose all of the prime factors of $p - 1$ are known. If there exists an integer a for each prime q dividing $p - 1$ (the value of a can be different for each q) such that

$$a^{(p-1)/q} \not\equiv 1 \pmod{p}$$

and

$$a^{p-1} \equiv 1 \pmod{p},$$

then p is prime.

This method requires that all the prime factors of $p - 1$ be known. This makes it less flexible and less effective than PPLS. PPLS will succeed much more often. The method is, however, easier to code. There is at least a 50% probability that any randomly chosen a will work, so in practice checking only a few values of a usually suffices. The probability of not succeeding after trying k different values of a is less than 2^{-k}. Choosing the values of a consecutively from the sequence of integers starting with 2, but skipping perfect n^{th} powers ($n > 1$), will be the

most efficient for computation: {2, 3, 5, 6, 7, 10, 11, 12, 13, 14, 15, 17, 18, 19, 20, 21, 22, 23, 24, 26, 28, 29, 30, 31, 33, 34, 35, 37, …}.

1. Set a threshold k for stopping.

   Let n be the number of factors of $p - 1$; i.e., denote the factors as $\{q_1, q_2, \ldots, q_n\}$.

   Let $j = 0$ (to keep track of how many q_i have been satisfied).

   Let $a = 2$.

2. If $a^{p-1} \equiv 1 \pmod{p}$ then

   for each i for which an a has not been found for factor q_i,

   if $a^{(p-1)/q_i} \not\equiv 1 \pmod{p}$ then,

   increment j, and

   print (or store) $\{q_i, a\}$.

3. If $a < k$ and $j < n$ then

   increment a by 1,

   go to Step 2.

4. If $j = n$, then p is prime.

A certificate for this primality proof consists of a list of the values of the factors of p–1 and the corresponding values of a.

Example. For $p = 3439601197$, $p - 1 = 2^2 \cdot 3 \cdot 286633433$. Then:

$2^{p-1} \equiv 1 \pmod{p}$ and

For $q = 2$, $2^{(p-1)/2} \equiv -1 \pmod{p}$.

For $q = 3$, $2^{(p-1)/3} \equiv 1346083919 \pmod{p}$.

For $q = 286633433$, $2^{(p-1)/286633433} \equiv 4096 \pmod{p}$.

Thus, p is *proved* prime.

A certificate consists of the list [(2,2), (3,2), (286633433, 2)].

NOTE In this case, a single value of a is sufficient for all factors of (p–1). This need not be the case in general. Further, the proof depended upon knowing that 286633433 is prime. This may be proven by any of the methods described in this document. In particular, trial division would work well. Alternatively, one could apply the methods of this section to 286633433. This means that this algorithm can, if desired, be used recursively.

5.2.4.2 Prime Testing with Elliptic Curves

The Elliptic Curve Primality Proving (ECPP) algorithm is a truly *general-purpose* algorithm that can *prove* primality of arbitrary large primes. The algorithm is also very efficient. Typically, a proof for a 200 decimal-digit number will take less than half an hour on a fast workstation. Unfortunately, the ECPP algorithm is exceedingly complex to specify and code. However, the proof it constructs is fairly simple to verify independently. The algorithm

constructs a primality certificate containing sufficient information to verify the proof, and the computations necessary for validating the certificate are necessary components of any public-key Elliptic Curve software.

This standard defines the structure and components of ECPP certificates, and specifies how they are verified. However, it is beyond the scope of this standard to define and specify the ECPP algorithm for constructing these certificates, due to its extreme complexity. Successful verification of an ECPP certificate, however it is created, is recognized by this standard as sufficient to accept a number as prime. Use of the ECPP algorithm itself is limited by this standard to the construction of ECPP certificates. This means that the existence, or assertion of existence, of an ECPP certificate does not by itself constitute sufficient grounds for primality under this standard; it is the independent verification of the certificate that establishes primality. Therefore, anyone using the ECPP method is *required* by this standard to save the resulting certificate. Such a certificate must be securely stored if the prime is part of a private key, just like all other information associated with a private key. Refer to Annex B.1 for further discussion.

ECPP is a recursive descent algorithm. At each level of descent, the problem is reduced to proving that a smaller number than the previous level is prime, until finally the number is small enough to be proved prime by the methods of Section 5.2. This is done by constructing an elliptic curve modulo the candidate prime and exhibiting a point on this curve with sufficiently large order. If the order of the exhibited point is prime, then the candidate is also prime. The order of the exhibited point becomes the candidate prime for the next level of descent.

ECPP Primality Certificate

The structure of a section of the certificate is as follows: The sections are repeated for every descent level of the algorithm, starting at 0. The overall certificate is an ordered union of the individual sections, ordered by descent level.

j : The descent level.

p : The prime being proved at this level.

D : The value of the discriminant of the complex quadratic field. If $D = 0$, then p was proved prime with the methods of Section 5.2. The data below is then omitted and replaced with data specific to the method used, including identification of the method.

a,b : The coefficients of the elliptic curve $y^2 = x^3 + ax + b$.

m : The number of points on this curve modulo p.

$f_1, f_2, f_3, \ldots$: All of the prime factors of m.

$S = (x, y)$: A point S on the curve having coordinates (x, y); i.e., x and y satisfy $y^2 \equiv x^3 + ax + b \pmod{p}$.

q : the order modulo p of the point (x, y).

ECPP Certificate Verification

The certificate is verified a level at a time, beginning with level 0. For each level, perform the following steps:

1. Verify that the correct prime is being proved. For level 0, it should be the target number for the certificate. Otherwise, it should be the order q of the point exhibited in the previous level.

2. Verify that $4a^3 + 27b^2 \not\equiv 0 \pmod{p}$; i.e., that a and b define a nonsingular elliptic curve.

3. Verify that the point (x, y) is a valid point on the curve (i.e., $y^2 \equiv x^3 + ax + b \pmod{p}$). If the point is represented by three coordinates instead of two, then a check shall be made to verify that the point is not the elliptic curve identity O.

4. Verify that $qS = O$. This is scalar multiplication of the elliptic curve point $S = (x, y)$ by q. Refer to ANSI X9.62 (particularly Annexes B.3 and D.3.2) for the definition and techniques of this multiplication. If $qS = O$, and if q is prime (verified by succeeding descent levels), then q is the order of the point (x, y).

5. Verify that $q > (p^{1/4} + 1)^2$. Then p is prime if q is prime; i.e., q becomes the candidate prime to be proved by the next level.

NOTE Verification does not use D, m, or the factors of m. Nevertheless, for a number of reasons they appear in certificates produced by many implementations of the ECPP algorithm. This standard includes them in the certificate definition in order to accommodate the use of such certificates.

The method is illustrated in Table 1.

Table 1: An ECPP certificate for p = 377681287.

j	p	D	(a, b)	m	factors of m	S	q
0	377681287	–3	(0, 343)	377660188	{2, 19, 97, 51229}	(328938831, 132650121)	51229
1	51229	–3	(0, 4)	50817	{3, 13, 1303}	(18245, 46776)	1303
2	1303	–3	(0, 1)	1236	{2, 3, 103}	(1152, 1203)	103
3	103	–3	(0, 22)	124	{2, 31}	(31, 47)	31
4	31	0	31 is a prime by trial division.				
NOTE This example could easily have been terminated at an earlier stage by appeal to trial division, but was continued as far as possible for the purpose of illustration.							

5.3 Constructive Methods

5.3.1 Shawe-Taylor's Algorithm[4]

The following algorithm generates guaranteed primes that need not be tested by any other algorithms. It is therefore intended for first party prime generation. This version of the algorithm cannot be used to test a prime given by a second party. This algorithm is recursive. To generate a prime of b bits let the routine ST_Random_Prime(b) return a prime of b bits:

1. If $b < 33$, then let p be an integer of b bits generated uniformly at random from the set of b-bit integers. Test p for primality by trial division. That is, the primality of p is proven by trying all possible prime divisors up to its square root. The methods of Section 5.2 may also be used in place of trial division for improved speed. If p is prime then return(p). Otherwise, repeat this step.

---

[4] This specification is a technical improvement over the specification in ANS X9.80–2001. The 2001 specification will work in almost all cases, although this specification is preferred.

2. If b is odd, let q = ST_Random_Prime((b + 3)/2). If b is even, let q = ST_Random_Prime((b + 2)/2).

3. Select a uniformly chosen random integer x in $[2^{b-1}, 2^b - 1]$ or from the interval $[(\sqrt{2})(2^{b-1}), (2^b - 1)]$.

4. Let t be the least integer greater than $x/(2q)$.

5. If $2tq + 1 \geq 2^b$, then let t be the least integer greater than $2^{b-1}/(2q)$.

6. Let $y = 2tq + 1$.

7. Select, uniformly at random, an integer a in $[2, y-2]$.

8. Let $x = a^{2t} \bmod y$.

   a. If x does not equal 1 and GCD(x–1,y) = 1 and $x^q \equiv 1 \pmod{y}$, then return(y).
   Otherwise, let $t = t + 1$.

   b. Go to Step 5.

Example. Generate a 40-bit prime via ST_Random_Prime(40).

1. $b \geq 33$, so proceed to Step 2.

2. Set q = ST_Random_Prime(21). This invokes this procedure recursively:

   1. Since $b < 33$, randomly select 21-bit integers until finding one that is proven prime via trial division. This procedure gives (say) p = 1832371, and thus p is returned.

   Now, q = 1832371 so proceed to Step 3.

3. Choose x = 966903465047 as a random integer in $[2^{39}, 2^{40} - 1]$.

4. Then, t = 263840.

5. Since $2tq + 1 < 2^{40}$, proceed to Step 6.

6. Then, $y = 2tq + 1$ = 966905529281.

7. Select (say) a = 41767655812.

8. Compute x = 34463255264 = $a^{2t} \bmod y$. However, GCD(x – 1,y) = 1553, so set $t = t + 1$ = 263841. Then, $2tq < 2^{40}$, so return to Step 5.

   After 40 iterations of repeating Steps 5 through 8, it is determined that t = 263879 and y = 967048454219 satisfy the conditions in Step 8 with a = 18999231450, so return the prime p = 967048454219 as the constructed prime.

5.3.2 Maurer's Algorithm

This is another recursive algorithm that returns guaranteed primes. It is therefore intended for first party prime generation. It cannot be used to test a prime given by a second party. M_Random_Prime(b) will return a random prime of b bits:

1. Let c = 1.05, $rmax$ = 20.

2. If $b < 20$
    a. Set n = a random b-bit integer.
    b. Determine, by trial division or by the methods of Section 5.2, that n is prime. If n is not prime, go to 2a.
    c. Return n.
3. Set $B = c \cdot b^2$.
4. If $b > 2 \cdot rmax$, then
    a. Select s uniformly at random in [0,1].
    b. Let $r = 2^{s-1}$. It suffices, in practice, to compute r to single precision.
    c. If $rmax > (b - r \cdot b)$, go to 4a.

    Else set $r = 0.5$.
5. Set q = M_Random_Prime($\lfloor r \cdot b \rfloor + 1$).
6. Set $I = \lfloor 2^{b-1}/(2q) \rfloor$.
7. Set success = False.
8. While (success is False)
    a. Set R = a uniformly chosen random integer in $[I + 1, 2I]$.
    b. Set $n = 2Rq + 1$.
    c. Determine, by trial division, if n is divisible by a prime number $< B$ (defined in Step 3). If it is not, then
        i. Set a = a uniform random integer in $[2, n-2]$.
        ii. Compute $e = a^{n-1} \bmod n$.
        iii. If $e = 1$

            then set $e = a^{2R} \bmod n$ and $d = \text{GCD}(e-1, n)$.

            If $d = 1$, then set success = True.
9. Return n.

NOTE A speed improvement can be obtained using the methods of Section 5.2 to prove that n is prime in Step 8, rather than using the method given above.

Example. Generate a 40-bit prime via M_Random_Prime(40).

1. Set $c = 1.05$ and $rmax = 20$.
2. $b > 20$, so proceed to Step 3.
3. Set $B = 1680$.
4. $b \leq 2 \cdot rmax$, so set $r = .5$ and proceed to Step 5.
5. Set q = M_Random_Prime(21). This calls the procedure recursively.
    1. Set $c = 1.05$ and $rmax = 20$.

2. $b > 20$, so proceed to Step 3.

3. Set $B = 463.01$

4. $b \leq 2 \cdot rmax$, so set $r = 0.5$ and proceed to Step 5.

5. Set q = M_Random_Prime(10). This again calls the procedure recursively.

    1. Set $c = 1.05$ and $rmax = 20$.

    2. $b < 20$, so randomly search for a 10-bit number that is proven prime by trial division. This returns $q = 571$ (say).

6. Set $I = \lfloor 2^{20}/(2 \cdot 571) \rfloor = 918$.

    7. Set success = False.

    8. Set $R = 1170$ (say), a uniform random number in [919, 1836]. Set $n = 1336141 = 2 \cdot R \cdot q + 1$. This happens to be prime, so return $n = 1336141$.

6. Now, $q = 1336141$. Set $I = \lfloor 2^{39}/(2 \cdot q) \rfloor = 205725$.

7. Set success = False.

8. Select R repeatedly, uniformly at random, until finding that $R = 300713$ (say) gives $n = 2 \cdot R \cdot q + 1 = 803589937067$. A value of $a = 7117234671$ chosen randomly satisfies $GCD(a^{2R}–1, n) = 1$. Therefore, set success = True and proceed to Step 9.

9. Return $n = 803589937067$ as the constructed prime.

5.4 Side Conditions for Generating Primes using Random Integers

It is sometimes desirable to generate random primes with certain side conditions. For example, it might be desired to generate a prime p such that $p \equiv 1 \pmod{k}$ for some k. Or it might be desired that $p \equiv 1 \pmod{k}$ *and* $p \equiv -1 \pmod{s}$ for some k and s. These are the conditions required when generating so-called *strong* primes. A strong prime p is one for which both $p – 1$ and $p + 1$ have a large prime factor. Such primes are generated as follows.

Suppose it is desired that $p \equiv 1 \pmod{p_1}$, $p \equiv -1 \pmod{p_2}$, and p has k bits. First, generate p_1 and p_2 as primes of the desired size. Then

1. Compute $R = (p_2^{-1} \bmod p_1)\, p_2 – (p_1^{-1} \bmod p_2)\, p_1$.

2. If $R < 0$, replace R with $R + p_1p_2$.

3. Select a uniformly chosen random integer X in the desired interval.

4. Let $Y = X + (R – X \bmod p_1p_2)$.

5. Sieve the sequence Y, $Y + p_1p_2$, $Y + 2p_1p_2$, … using the sieve procedure in Annex A.3.

Test candidates that survive the sieve for primality until a prime is found or until the candidate is outside the desired interval. In the latter case select a new X and start again. The methods of Section 5.2.4 are applicable, since $p – 1$ and $p + 1$ already have a large known prime factor. The computation of R is an application of the

Chinese Remainder Theorem (CRT). Additional modular side conditions can be added, if desired, by using the CRT. See reference [9] for a full description of the CRT.

If it is desired to construct primes with only one modular side condition, then do the following. Suppose it is desired that $p \equiv a \pmod{p_1}$ for some a.

1. Select a uniformly chosen random integer X in the desired interval.

2. Let $R = X \bmod p_1$

3. Let $Y = X + (a - R)$.

4. Sieve the sequence $Y,\ Y + p_1,\ Y + 2p_1,\ \ldots..$ using the sieve procedure in Annex A.3.

5. Test candidates that survive the sieve for primality until a prime is found or until the candidate is outside the desired interval. In the latter case select a new X and start again..

Example 1. Find a 32-bit prime such that $p \equiv 1 \pmod{1009}$.

Choose (say) X = 2579700989. Then compute $R = 779 = X \bmod 1009$ and $Y = X + R = 2579701768$. Y is the first number greater than X that is 1 modulo 1009. Applying the sieve procedure from Annex A.3 with primes less than 20 yields that among the first 40 positions, positions 1, 15, 19, 25, 27, 31, 37, and 39 are left untouched by the sieve. These correspond to $Y + 1 \cdot 1009$, $Y + 15 \cdot 1009$, …etc. Testing each one for primality by trial division reveals that $Y + 27 \cdot 1009$ is prime.

Example 2. Find a 32-bit prime p such $p \equiv 1 \pmod{1009}$ and $p \equiv -1 \pmod{827}$.

Again choose X = 2579700989. Compute $R = (827^{-1} \bmod 1009) \cdot 827 - (1009^{-1} \bmod 827) \cdot 1009 = -311780$. Since $R < 0$, replace it with $R + 1009 \cdot 827 = 522663$. Then, $Y = 2580620419$. This is the first integer greater than X congruent to 1 modulo 1009 and –1 modulo 827. Applying the sieve procedure from Annex A.3 with primes less than 20 yields that among the first 50 positions, positions 14, 20, 24, 30, 38, 44, and 48 are untouched by the sieve. These correspond to $Y + 14 \cdot 1009 \cdot 827$, $Y + 20 \cdot 1009 \cdot 827$, … etc. Testing each one for primality reveals that $Y + 48 \cdot 1009 \cdot 827$ is prime.

6 Candidate Prime Testing Methods

This section discusses methods for second party prime testing; that is, the testing of primes which are given by another party.

Theoretically, it is possible to generate a composite integer which will pass one round of the Miller-Rabin (M-R) test with probability = ¼. However, it is not possible to construct a composite that passes M-R with probability = ¼ and also passes a Lucas test with high probability. Thus, while one normally needs to be extra cautious in using a prime generated by someone else, the methods of this standard do not permit another party to construct composites in such a way that the user of this standard can be fooled into believing they are prime.

Therefore, with one exception, the methods of Section 5 are adequate for testing primes that have been generated by an outside party. The exception is that the use of successive primes 2,3,5,7,11, … as the choice of base for a Miller-Rabin test is disallowed. The bases must be randomly selected and their choice must be unknown to the outside party. Similarly, the choice of polynomials chosen for Frobenius-Grantham must be kept secret from the outside party as well.

If absolute certainty is required when testing a candidate prime given by an outside party, then one of the deterministic methods must be applied.

7 Tables of Parameters

7.1 Rounds Required for Miller-Rabin if Followed by Lucas

Table 2 gives, as a function of the size of the prime number being tested, the number of rounds of the Miller-Rabin algorithm needed to achieve a failure probability of less than 2^{-100}. This table assumes that the M-R tests will be followed by a Lucas test. Otherwise, 50 rounds of Miller-Rabin are required.

Table 2: Rounds Required for Miller-Rabin

Size of Prime (in bits)	Number of Rounds to Achieve 2^{-100}
Less than 100	50
100 - 255	27
256 - 511	15
512 - 767	8
768 - 1023	4
1024 or more	2
NOTE Taken from reference [5]. See the Bibliography. The number of rounds stated above for each range of primes is the maximum required for that range. The ranges are included for convenience of implementation and interoperability.	

7.2 Rounds Required for Frobenius-Grantham

Table 3 presents the number of rounds of the Frobenius-Grantham algorithm that are required to achieve a failure probability of less than 2^{-100}. This table is derived by combining the results of reference [7] with the worst case, per-iteration failure probability of 1/7710 for this method, assuming that quadratic polynomials are used. While similar probabilities certainly exist for cubic and higher degree polynomials, nothing has been proved. The analytical estimates given in reference [5], from which Table 2 is taken, have not yet been worked out by mathematicians for the Frobenius-Grantham method. The numbers below are a crude upper bound on the number of rounds needed.

Table 3: Rounds Required for Frobenius-Grantham

Size of Prime (in bits)	Number of Rounds to Achieve 2^{-100}
Less than 256	8
256 – 511	7
512 - 767	3
768 or more	2
NOTE Derived from references [6] and [7]. See the Bibliography. The number of rounds stated above for each range of primes is the maximum required for that range. The ranges are included for convenience of implementation and interoperability.	

Annex A
(normative)

This Annex presents algorithms that can be used to effect the computations from Sections 5 and 6. Alternative methods that give identical results are allowed.

A.1 Modular Exponentiation

To compute a^k mod n:

1. Write k in binary as $b_n b_{n-1} \ldots b_0$ where $b_i = 0$ or 1 and b_n is 1.

2. Put $x = a$.

3. For $i = n - 1$ down to 0 do

   Replace x with x^2 mod n.

   If $b_i = 1$, also replace x with $(a \cdot x)$ mod n.

4. Return x.

Equivalent methods are allowed.

A.2 Jacobi Symbol[5]

The Jacobi symbol is derived from the Legendre symbol. Let p be an odd prime, and let a be a positive integer. The Legendre symbol of the integer a with respect to the prime p may be computed by Euler's Theorem:

$$\left(\frac{a}{p}\right) = a^{(p-1)/2} \bmod p$$

The Legendre symbol of multiples of p with respect to the prime p is zero. When the integer a is not a multiple of a prime p, then the Legendre symbol of a with respect to p is valued to either +1 or –1, depending on whether a is or is not a square modulo p.

Let n be an odd positive integer, and let a be a positive integer. The Jacobi symbol of a with respect to n is the product of the Legendre symbols of a with respect to the prime factors of n (counting multiplicity of the factors).

Therefore, if $n = pq$, then $\left(\frac{a}{n}\right) = \left(\frac{a}{p}\right)\left(\frac{a}{q}\right)$.

By definition, the Jacobi symbol of any integer with respect to 1 is 1.

---

[5] This specification is a technical correction to the specification in the ANS X9.80–2001. The 2001 specification **shall not** be used as it will sometimes produce erroneous results.

The Jacobi symbol of any integer *a* with respect to any odd positive integer *n* may be efficiently computed without knowing the prime factors of *n* using the Law of Quadratic Reciprocity. See [9], Algorithm 2.149.

To compute $\left(\frac{a}{n}\right)$, first reduce *a* modulo *n* so that $0 \le a < n$, then perform the following:

Jacobi (*a, n*) // $0 \le a < n$, *n* odd and positive

1. If $n = 1$ or $a = 1$, then return 1.
2. If $a = 0$, then return 0.
3. Find *e* such that $a = 2^e a_1$, where a_1 is odd.
4. If *e* is even, then set $s = 1$.
   Else if $n \equiv 1 \pmod 8$ or $n \equiv 7 \pmod 8$, then set $s = 1$.
   Else if $n \equiv 3 \pmod 8$ or $n \equiv 5 \pmod 8$, then set $s = -1$.
5. If $n \equiv 3 \pmod 4$ and $a_1 \equiv 3 \pmod 4$, then set $s = -s$.
6. Set $n_1 = n \bmod a_1$.
7. Return ($s \cdot$ Jacobi (n_1, a_1))

Example. Compute the Jacobi symbol for $a = 5$ and $n = 3439601197$.

1. *n* is not 1 and *a* is not 1, so proceed to Step 2.
2. *a* is not 0, so proceed to Step 3.
3. $5 = 2^0 \cdot 5$, so $e = 0$ and $a_1 = 5$.
4. *e* is even, so set $s = 1$.
5. a_1 is not congruent to 3 modulo 4, so do not change *s*.
6. Set $n_1 = 2 = n \bmod 5$.
7. Compute and return (1 · Jacobi (2, 5)). This calls Jacobi recursively. Jacobi (2, 5) = –1, so –1 is returned.
   1. *n* is not 1 and *a* is not 1, so proceed to Step 2.
   2. *a* is not 0, so proceed to Step 3.
   3. $2 = 2^1 \cdot 1$, so put $e = 1$ and $a_1 = 1$.
   4. *e* is odd, and $n \equiv 5 \pmod 8$, so set $s = -1$.
   5. *n* is not 3 modulo 4, so proceed to Step 6.
   6. Set $n_1 = 0 = n \bmod 1$.
   7. Return (– 1 · Jacobi (0, 1) = – 1). This calls Jacobi recursively.

1. *n* is 1, so return 1.

Thus, the value of the Jacobi symbol is –1.

A.3 Sieve Procedure

A *sieve procedure* is described as follows: Given a sequence of integers Y_0, $Y_0 + 1$, … , $Y_0 + J$, a sieve will identify which integers in the sequence are divisible by primes up to some selected limit.

Start by selecting a *factor base* of all the primes p_j , from 2 up to some selected limit L. The value of L is arbitrary and may be determined by computer limitations. A good, typical value of L would be anywhere from 10^3 to 10^5.

1. Compute $S_j = Y_0 \bmod p_j$ for all p_j in the factor base.

2. Initialize an array of length $J + 1$ to zero.

3. Then starting at $Y_0 - S_j + p_j$, let every p_j^{th} element of the array be set to 1. Do this for the entire length of the array and for every j.

4. When finished, every location in the array that has the value 1 is divisible by some small prime, and is therefore composite.

The array can be either a bit array for compactness when memory is small, or a byte array for speed when memory is readily available. There is no need to sieve the entire sieve interval at once. The array can be partitioned into suitably small pieces, sieve each piece, then go on to the next piece. When finished, every location with the value 0 is a candidate for prime testing.

The amount of work for this procedure is approximately M log log L, where M is the length of the sieve interval; this is a very efficient procedure for removing composite candidates for primality testing. If $L = 10^5$, the sieve will remove about 96% of all composites.

In some cases, rather than having a set of consecutive integers to sieve, the set of integers to be tested consists of integers lying in an arithmetic progression Y_0, $Y_0 + h$, $Y_0 + 2h$, …, $Y_0 + Jh$, where h is large and not divisible by any primes in the factor base. As before,

1. Start by selecting a factor base and initializing an array of length $J + 1$ to 0.

2. Again, compute $S_j = Y_0 \bmod p_j$ for all p_j in the factor base.

3. Also compute $T_j = h \bmod p_j$ and $r = -S_j T_j^{-1} \bmod p_j$.

4. Then starting at $Y_0 + r$, let every p_j^{th} element of the array be set to 1. Do this for the entire length of the array and for every j. Note that the position $Y_0 + r$ in the array actually denotes the number $Y_0 + rh$.

5. As before, when finished, every location in the array that has the value 1 is divisible by some small prime and is therefore composite.

NOTE The prime ‘2’ takes the longest amount of time (M/2) to sieve since it touches the most locations in the sieve array. An easy optimization is to combine the initialization of the sieve array with the sieving of the prime ‘2’. With a little clever coding, the sieving of the prime ‘3’ can be included during initialization. These optimizations can save about 1/3 of the total sieve time.

A.4 Algorithms for Polynomial Arithmetic

The following notation is used throughout this section. Let $f(x) = f_n x^n + f_{n-1} x^{n-1} + \ldots + f_0$ and $g(x) = g_d x^d + g_{d-1} x^{n-1} + \ldots + g_0$ be polynomials whose coefficients f_i and g_i are integers and f_n and g_d are not zero. Let p be a prime. The degree of $f(x)$ is n and the degree of $g(x)$ is d. Without loss of generality, it is assumed throughout this annex that $n > d$. This implies $g_i = 0$ for $i > d$. Let s = maximum(degree($f(x)$), degree($g(x)$)), thus $s = n$.

This section presents the algorithms needed to execute the Frobenius-Grantham algorithm.

I. Polynomial Reduction mod *p*.

To compute $f(x)$ mod p, replace $f(x)$ with

$$(f_n \bmod p)\, x^n + (f_{n-1} \bmod p)\, x^{n-1} + \ldots + (f_0 \bmod p).$$

That is to say, replace each coefficient with its value modulo p.

II. Polynomial Addition.

$$f(x) + g(x) = (f_s + g_s)\, x^s + (f_{s-1} + g_{s-1})\, x^{s-1} + \ldots + (f_0 + g_0)$$

The sum of two polynomials has degree s. Its coefficients are the sum of the respective coefficients.

III. Polynomial Multiplication.

The degree of the product of two polynomials is the sum of the degrees of the individual polynomials. Let $m = n + d$. Then

$$f(x)\, g(x) = r_m x^m + r_{m-1} x^{m-1} + \ldots + r_0$$

where $r_h = f_0 g_h + f_1 g_{h-1} + f_2 g_{h-2} + \ldots + f_h g_0$.

Example.

$f(x) = 2x^2 + 3x + 1$

$g(x) = 3x^3 + x^2 + 2x + 2$

$f(x)\, g(x) = r_5 x^5 + r_4 x^4 + r_3 x^3 + r_2 x^2 + r_1 x + r_0$

where $r_5 = 2 \cdot 3$

$r_4 = 3 \cdot 3 + 1 \cdot 2$

$r_3 = 1 \cdot 3 + 3 \cdot 1 + 2 \cdot 2$

$r_2 = 1 \cdot 2 + 3 \cdot 2 + 2 \cdot 2$

$r_1 = 1 \cdot 2 + 3 \cdot 2$

$r_0 = 1 \cdot 2$

IV. Polynomial Division.

Polynomial division represents $f(x)/g(x)$ as $f(x) = q(x)g(x) + r(x)$, where $q(x)$ is the quotient polynomial, and $r(x)$ is the remainder. Unless $g(x)$ is monic or there are certain other very restrictive conditions, the coefficients of $q(x)$ and $r(x)$ will not be integers. They will be rational numbers. The degree of the quotient $q(x)$ will be $n - d$, and the degree of the remainder will be less than d.

1. Set $h(x) = f(x)$ and $i = n - d$
2. While ($i \geq 0$)
   a. Compute coefficient: $q_i = h_{d+i} / g_d$
   b. Compute partial product: $s(x) = q_i x^i g(x)$
   c. Subtract: Replace $h(x)$ with $h(x) - s(x)$, and set $i = i - 1$
3. Set $r(x) = h(x)$

Example.

$f(x) = 3x^3 + x^2 + 2x + 2$

$g(x) = x^2 + 3x + 1$

Set $h(x) = 3x^3 + x^2 + 2x + 2$

Set $i = 3 - 2 = 1$

$q_1 = 3/1 = 3$

$s(x) = 3\ x\ g(x) = 3x^3 + 9x^2 + 3x$

$h(x) = 3x^3 + x^2 + 2x + 2 - (3x^3 + 9x^2 + 3x) = -8x^2 - x + 2$

Set $i = i - 1 = 0$

$q_0 = -8/1 = -8$

$s(x) = -8\ g(x) = -8x^2 - 24x - 8$

$h(x) = -8x^2 - x - (-8x^2 - 24x - 8) = 23\ x + 10$

Since $i = 0$, the algorithm is complete. $q(x) = 3x - 8$ and $r(x) = 23x + 10$

NOTE Some of the algorithms herein require the computation of $f(x)$ mod $g(x)$. This is the same as $r(x)$, i.e. the remainder when $f(x)$ is divided by $g(x)$.

V. Polynomial Division mod *p*.

Polynomial division mod p is identical to polynomial division except that the algorithm starts by replacing $g(x)$ with $((g_d^{-1} \bmod p)\ g(x) \bmod p)$. That is to say, compute the inverse modulo p of the lead coefficient of $g(x)$, multiply the entire polynomial by that inverse, then reduce modulo p. Perform the division, then reduce $q(x)$ and $r(x)$ modulo p.

NOTE Various methods including the extended Euclidean algorithm can be used to compute the inverse of a non-zero integer.

VI. Polynomial Exponentiation.

To compute $f(x)^e$

1. Write e in binary as $b_n b_{n-1} \ldots b_0$, where $b_i = 0$ or 1 and b_n is 1.

2. Set $h(x) = f(x)$.

3. For $i = n - 1$ down to 0 do

   Replace $h(x)$ with $h(x) \cdot h(x)$. If computing $f(x)^e \bmod p$, reduce mod p.

   If $b_i = 1$, also replace $h(x)$ with $f(x)h(x)$. Again, reduce mod p if computing $f(x)^e \bmod p$

Step 4. Return $h(x)$.

VII. Polynomial GCD.

To compute GCD($f(x)$, $g(x)$)

1. Set $\max(x) = f(x)$ and $\min(x) = g(x)$

2. Compute the remainder for $\max(x)/\min(x)$., i.e. $max(x) \bmod min(x)$.

   Reduce the remainder modulo p if computing GCD($f(x)$, $g(x)$) mod p.

3. If the remainder is 0, return($\min(x)$). If the remainder is a non-zero constant, return the (constant) polynomial 1.

4. Replace $\max(x)$ with $\min(x)$, and $\min(x)$ with the remainder from Step 2.

A.5 Lucas Sequence[6]

This section shows how to compute Lucas(D, K, p), the K^{th} element of a Lucas sequence modulo p with pre-chosen discriminant D.

Lucas(D, K, p)

Initialize: Set $P = 1$, $U = 1$, and $V = 1$

1. Let $K_r K_{r-1} \ldots K_0$ be the binary expansion of K where K_r is 1. The bit length of K is then $r + 1$.

2. For $i = r - 1$ to 0

   Set $(U, V) = (UV \bmod p, \frac{V^2 + DU^2}{2} \bmod p)$

   If $K_i = 1$, then also set

   $$(U, V) = (\frac{PU + V}{2} \bmod p, \frac{PV + DU}{2} \bmod p)$$

3. Return ($U_K = U$).

NOTE Step 2 contains expressions of the form $A/2 \bmod p$, where A is an integer and p is assumed to be an odd integer. If $A/2$ is not an integer, in other words, if A is odd, then $A/2 \bmod p$ may be calculated as $(A+p)/2 \bmod p$. (Alternatively, $A/2 \bmod p = A \cdot (p+1)/2 \bmod p$, for any integer A, odd or even.)

---

[6] Formerly called “Lucas-Lehmer Sequence” in ANS X9.80–2001.

Example. Select p = 3439601197. A computation of the Jacobi symbol reveals that $\left(\frac{D}{N}\right) = -1$ for $D = 5$. We show here the calculations for Lucas(D, $p + 1$, p).

Set r = 31, P = 1, U = 1 and V = 1. The values of (U,V) mod p at the end of each iteration in Step 2 for i = 30,25,20,15,10,5, and 0 are:

I	(U, V) mod p
30	(2, 4)
25	(661792756, 730279310)
20	(–605926906, –914766781)
15	(437522511, –1716871053)
10	(–858309982, 204659912)
5	(–1043840240, 1506919048)
0	(0,–2)

Return 0 as the value for U_{p+1}.

NOTE The (U, V) values in the example are represented modulo p as signed integers in the range (–p/2, p/2), but alternative representations that produce equivalent results, such as the usual [0, p–1] range, are also allowed.

Annex B
(informative)

B.1 Discussion of General Prime Proving Methods

The methods discussed in Section 5 are all either special purpose methods, i.e. they generate or test primes of some specific form, or they are random methods; they only guarantee primality with a certain (very high) probability. There do exist two *general-purpose* algorithms that can *prove* primality of arbitrary large primes. The first is known by several names: Adleman-Pomerance-Rumely, Cohen-Lenstra-Bosma, and the Cyclotomy Method. The second is known as the Elliptic Curve Primality Proof (ECPP). Both algorithms can readily prove the primality of numbers up to several hundred decimal digits in only a few minutes on fast workstations.

This standard does not describe these methods, as a detailed description of each one would easily run to 100 pages of text apiece. The code for each would be similarly complicated. This standard does not allow the use of the Cyclotomy method. The reason for this is that the method is so complicated that the probability of an undetected bug or programming error far exceeds the probability that a probable prime, generated by the methods of Section 5.1 is *not* prime. Furthermore, the method does not provide a certificate of primality that might be checked by an independent program. If an error is present in the code, it would be very difficult to detect. One would only detect an error in the code if a number that was declared prime turned out to be composite, and even then, the fact that the number is composite would need to be determined by some other means.

This standard allows the use of ECPP. While ECPP is also complicated to code, this method provides a *certificate* of primality. This certificate consists of numbers that can be easily checked by an independent ECPP program. Thus, the proof can be independently verified. The Cyclotomy method provides no such certificate and is hence disallowed. The certificate provides assurance that the primality proof is correct, despite the complex calculations.

Furthermore, ECPP certificate verification can be coded from components of any public-key Elliptic Curve software, as noted in Section 5.2.4.2. Details of these components can be found in ANSI X9.62. Further documentation of the method, including the algorithm for constructing the certificates, is found in references [10] and [1].

B.2 Discussion of the Distribution of Randomly Chosen Primes

When generating random primes, this standard recommends selecting a random starting point, and then incrementally searching and testing successive (odd) integers until one is found that is prime. In addition, because testing individual integers for primality is expensive, this standard also suggests applying a sieve procedure to eliminate most of the composites as candidates to be tested.

Since primes are not uniformly distributed over a given interval, the above method does not select all primes with equal probability. For example, it is believed that there exist infinitely many prime *pairs,* integers n and $n + 2$, that are both prime. On the other hand, the average gap between primes is, by the Prime Number Theorem, about $\log n$. Therefore, if the initial integer is randomly selected, and subsequent integers are incrementally searched for primes, it is unlikely that the random starting point will be $n + 1$, when n and $n + 2$ are both prime. An incremental search method is unlikely to select the prime $n + 2$. However, primes that are preceded by a small gap are extremely rare. According to Brandt and Damgard [3] the gaps between consecutive primes loosely speaking follows an exponential distribution. Thus, the set of primes that can likely be found by an incremental search procedure is nearly the entire set of primes. There is no security risk from using this procedure in the sense that one might believe that it reduces the space of primes from which selection is made. The procedure selects primes *very nearly,* but not exactly, uniformly at random.

Annex C
Summary of Changes from ANS X9.80–2001
(informative)

C.1 Introduction

This annex summarizes the differences between this version of ANS X9.80 and the previous version, ANS X9.80–2001.

Technical changes are covered in C.2, followed by editorial changes in C.3. Minor editorial changes have also been made to ANS X9.80 (e.g., capitalization, punctuation, bibliography style, italics, symbol font, reference updates), but are not itemized here.

C.2 Technical changes

C.2.1 Search Range for primes

In Clause 5.2.1 the interval $[(\sqrt{2})(2^{d-1}), (2^d - 1)]$ was added as an option. The previous specification of this Standard did not support the specification of a range of output for the prime. This omission has been corrected in this Standard.

The impact on existing implementations is potentially significant as the needed interface did not exist for some applications of this Standard, such as the RSA Standards X9.31 or X9.44. If the previous Standard was somehow used for RSA applications (even though the interface was incorrect), then the intended security may not have been achieved.

C.2.2 Errors in Jacobi symbol algorithm

The following changes have been made in the algorithm for computing the Jacobi symbol (Clause A.2):

- Step 1 must return 1 if $n = 1$ or $a = 1$
- Step 2 must return 0 if $a = 0$

With these changes, the special case for $a = 2$ is no longer necessary. Without these changes, the algorithm does not compute the correct value in many cases.

Also, for clarity, the following conditions have been noted:

- the Jacobi symbol of any integer with respect to 1 is 1
- the integer n must be odd and positive
- within the algorithm itself, the integer a must be in the range $[0, n-1]$ (the integer a should be reduced to this range before the algorithm is called)

The example has been updated accordingly.

(An early version of the 2001 standard may have had an error in the first part of Step 4, but this was corrected in the version dated August 15, 2001, on which these changes are based. It is mentioned here for completeness. The first part of Step 4 should test $n \equiv 1$ or 7 mod 8. In an early version it tested $n \equiv 1$ or 3 mod 8.)

The impact on existing implementations is **potentially significant**, to the extent that the implementations follow the incorrect algorithm in the 2001 standard, rather than computing the correct Jacobi symbol. Under the 2001 incorrect algorithm it was possible to output a prime that was not a prime and vice versa.

C.2.3 Range of bases in Miller-Rabin test

The range of bases for the Miller-Rabin test (Clause 5.2.3.1) has been updated to [2, p–2]. The rationale is that a composite will always pass the test for p–1. (The range currently excludes 1 for this reason.) Other algorithms in the standard already exclude p–1.

The impact on existing implementations is negligible since p–1 is unlikely to be selected at random, nor chosen intentionally since smaller bases are more efficient.

C.2.4 Perfect squares in Lucas test

A new Step 1 has been inserted to the Lucas test (Clause 5.2.3.2) to exclude perfect squares, as otherwise the test will not terminate.

The impact on existing implementations is negligible since the condition is highly unlikely to occur in the way the test is employed in the standard (following several Miller-Rabin tests).

C.2.5 Discriminants with Jacobi symbol 0 in Lucas test

The renumbered Step 2 of the Lucas test (Clause 5.2.3.2) has been updated to return ‘composite’ if the discriminant D has Jacobi symbol 0. The statement about the probability that the Jacobi symbol is –1 has been updated to say “about ½”. The rationale is that the test would be more efficient and more consistent with the literature if it excluded discriminants with Jacobi symbol 0. Also, the probability that the Jacobi symbol is –1 is not exactly ½ in every case, but is about ½. A note has been added referencing the literature [2].

The impact on existing implementations is negligible since the condition is highly unlikely to occur in the way the test is employed in the standard (following several Miller-Rabin tests).

C.2.6 Boundary conditions in Shawe-Taylor’s algorithm

The range checking in Shawe-Taylor’s algorithm (Clause 5.3.1) has been updated to check that the initial candidate prime has the requested size. This range check has been moved to the top of the loop, at Step 5. Also, the high end of range for x in Step 3 has been corrected to $2^b - 1$, not 2^b. The rationale is that although unlikely in practice, it is possible that the previous algorithm would output a prime longer than the requested size.

The example has been updated accordingly. (Three typos have also been corrected in the example: in Step 1, “<” has been changed to “≤”; in Step 2, “p_0” has been changed to “q”; and the number of iterations has been changed to 40.)

The impact on existing implementations is negligible since the integer x is unlikely to be chosen so close to the high end of the range.

C.3 Editorial issues

C.3.1 Random bit generators

The phrase “(or bit)” has been added after “random number” in the last bullet of Clause 1, as ANSI X9.82 is a (draft) standard for random *bit* generation.

C.3.2 Failure probability

A note has been added to Clause 1 giving rationale why the 2^{-100} failure probability is considered appropriate independent of the intended security level of the cryptosystem in which the prime is to be employed. (This addresses the question of whether the failure probability needs to be revised to accommodate security levels greater than 80 bits, which was the minimum security level in ASC X9 standards at the time ANSI X9.80 was originally developed.)

C.3.3 Lucas-Lehmer vs. Lucas

The name “Lucas-Lehmer” has been changed to “Lucas” throughout the document, as is standard in the literature. (“Lucas-Lehmer” is a different test for a special form of prime numbers, and is not used in the standard.)

C.3.4 Reference for combining Miller-Rabin and Lucas tests

The 1980 article by Pomerance, Selfridge and Wagstaff [11] is cited in Clause 5.1 in connection with the suggestion of combining a Miller-Rabin test to the base 2 and a Lucas test, and has been been added to the bibliography. This is in addition to the Baillie-Wagstaff reference already given [2].

C.3.5 Versions of Shawe-Taylor

The statement “It cannot be used to test a prime given by a second party” in 5.3.1 has been changed to “This version of the algorithm cannot be used …”, as a different version with additional properties is being considered in another X9 standard.

C.3.6 Binary expansions

Although the definition of “*m*-bit number” in Clause 3.3 resolves any ambiguity by requiring that the leftmost bit is 1, additional text has been added to the binary expansions in Clauses A.1, A.4 and A.5 to clarify this point.

C.3.7 Modulo *p* division in Lucas sequence algorithm

A note has been added to Clause A.5 to clarify the notation “$A/2 \bmod p$”. Division by 2 modulo p is implied, but the notation could be misinterpreted as real-number division, followed by a modulo p operation.

C.3.8 Negative numbers in Lucas sequence example

A note has been added to Clause A.5 explaining that the values in the table are represented modulo p in the range $(-p/2, p/2)$. A “signed” representation is convenient for this algorithm since the discriminant D may be negative.

C.3.9 Added Interval

The interval $[(\sqrt{2})(2^{d-1}),\ (2^{d}-1)]$ has been added to provide support for legacy systems in the generation of random primes with uniform distribution or using sequential search

Bibliography

[1] Atkin, A.O.L. & Morain, F., “Elliptic Curves and Primality Proving”, *Mathematics of Computation*, V.61 (1993), 29-69

[2] Baillie, R. & Wagstaff Jr., S.S., “Lucas Pseudoprimes”, *Mathematics of Computation*, V.35 (1980), 1391-1417

[3] Brandt, J & Damgard, I. On Generation of Probable Primes by Incremental Search E. F. Brikell, editor, *Advances in Cryptology – Crypto ’92,* pp. 358 – 370. Springer Verlag, 1993.

[4] Brillhart, J, Lehmer, D.H., Selfridge, J., Tuckerman, B., & Wagstaff Jr., S. “Factorizations of b^n +/- 1 for b = 2,3,5,6,7,10,11,12 up to high powers”, *Contemporary Mathematics*, Vol. 22, Amer. Math. Soc.

[5] Damgaard, I., Landrock, P., & Pomerance, C., “Average Case Error Estimates for the Strong Probable Prime Test”, *Mathematics of Computation,* V.61 (1993), 177-194

[6] Grantham, J., “A Probable Prime Test with High Confidence”**,** *Journal of Number Theory* V.72 (1998), 32-47

[7] Kim, S.H. & Pomerance, C., “The Probability that a Random Probable Prime is Composite”, *Mathematics of Computation,* V.53 (1989), 721-741

[8] Maurer, U., “Fast Generation of Prime Numbers and Secure Public-Key Cryptographic Parameters”, *Advances in Cryptology: EUROCRYPT ’89*, Springer-Verlag

[9] Menezes, A., van Oorschot, P., & Vanstone, S., *Handbook of Applied Cryptography,* CRC Press, 1997

[10] Morain, F., “Implementation of the Goldwasser-Kilian-Atkin Primality Testing Algorithm”, Ph.D. dissertation, U. Limoges, 1989

[11] Pomerance, C., Selfridge, J.L., & Wagstaff Jr., S.S., “The Pseudoprimes to $25 \cdot 10^9$”, *Mathematics of Computation*, V.35 (1980), 1003-1026

[12] Shawe-Taylor, J., “Generating Strong Primes”*, Electronics Letters*, V.22, (1986), 875-877