Appendices from FIPS 186-3: B.3, C.3, C.6, C.9, C.10 and F

B.3 IFC Key Pair Generation

B.3.1 Criteria for IFC Key Pairs

Key pairs for IFC consist of a public key (n, e), and a private key (n, d), where n is the modulus and is the product of two prime numbers p and q. The security of IFC depends on the quality and secrecy of these primes and the private prime factor d. The primes p and q **shall** be generated using one of the following methods:

A. Both p and q are randomly generated prime numbers (Random Primes), where p and q **shall** both be either :

   1. Provable primes (see Appendix B.3.2), or
   2. Probable primes (see Appendix B.3.3).

B. Both p and q are randomly generated prime numbers that satisfy the following additional conditions (Primes with Conditions):

   - (p-1) has a prime factor p_1
   - (p+1) has a prime factor p_2
   - (q-1) has a prime factor q_1
   - (q+1) has a prime factor q_2

   where p_1, p_2, q_1 and q_2 are called auxiliary primes of p and q.

   Using this method, one of the following cases **shall** apply:

   1. The primes p_1, p_2, q_1, q_2, p and q **shall** all be provable primes (see Appendix B.3.4),
   2. The primes p_1, p_2, q_1 and q_2 **shall** be provable primes, and the primes p and q **shall** be probable primes (see Appendix B.3.5), or
   3 The primes p_1, p_2, q_1, q_2, p and q **shall** all be probable primes (see Appendix B.3.6).

   The minimum lengths for each of the auxiliary primes p_1, p_2, q_1 and q_2 are dependent on *nlen*, where *nlen* is the length of the modulus n in bits (see Table B.1). Note that *nlen* is also called the key size. The maximum length is determined by *nlen*, the sum of the length of each auxiliary prime pair and whether the primes are probable primes or provable primes (e.g., for the auxiliary prime pair p_1 and p_2, **len**(p_1) + **len**(p_2) **shall** be less than a value determined by *nlen* and whether p_1 and p_2 are generated to be probable or provable primes)[1].

[1] For probable primes: $\mathbf{len}(p_1) + \mathbf{len}(p_2) < \mathbf{len}(p) - \log_2(\mathbf{len}(p)) - 6$; similarly for $\mathbf{len}(q_1) + \mathbf{len}(q_2)$. For provable primes: $\mathbf{len}(p_1) + \mathbf{len}(p_2) < \mathbf{len}(p)/2 - \log_2(\mathbf{len}(p)) - 7$; similarly for $\mathbf{len}(q_1) + \mathbf{len}(q_2)$.

Table B.1. Minimum and maximum lengths of p_1, p_2, q_1 and q_2

nlen	Min. length of p_1, p_2, q_1 and q_2	Max. length of len(p_1) + len(p_2) and len(q_1) + len(q_2)	
		Probable primes	**Provable primes**
1024	> 100 bits	< 496 bits	< 239 bits
2048	> 140 bits	< 1007 bits	< 494 bits
3072	> 170 bits	< 1518 bits	< 750 bits

For different values of *nlen* (i.e., key sizes), the methods allowed for the generation of *p* and *q* are specified in Table B.2.

Table B.2. Allowable Prime Generation Methods

nlen	Random Primes	Primes with Conditions
1024	No	Yes
2048	Yes	Yes
3072	Yes	Yes

In addition, all IFC keys **shall** meet the following criteria in order to conform to FIPS 186-3:

1. The public exponent *e* **shall** be selected with the following constraints:

   (a) The public verification exponent *e* **shall** be selected prior to generating the primes *p* and *q*, and the private signature exponent *d*.

   (b) The exponent *e* **shall** be an odd positive integer such that:

   $$2^{16} < e < 2^{256}.$$

   Note that the value of *e* may be any value that meets constraint 1(b), i.e., *e* may be either a fixed value or a random value.

2. The primes *p* and *q* **shall** be selected with the following constraints:

   (a) (p-1) and (q-1) **shall** be relatively prime to the public exponent *e*.

   (b) The private prime factor *p* **shall** be selected randomly from the primes that satisfy $(\sqrt{2})(2^{(nlen/2)-1}) \leq p \leq (2^{nlen/2} - 1)$, where *nlen* is the appropriate length for the desired *security_strength*.

   (c) The private prime factor *q* **shall** be selected randomly from the primes that satisfy $(\sqrt{2})(2^{(nlen/2)-1}) \leq q \leq (2^{nlen/2} - 1)$, where *nlen* is the appropriate length for the desired *security_strength*.

   (d) $|p - q| > 2^{(nlen/2) - 100}$.

3. The private signature exponent *d* **shall** be selected with the following constraints

after the generation of p and q:

(a) The exponent d **shall** be a positive integer value such that $d > 2^{nlen/2}$, and

(b) $d = e^{-1} \bmod (\text{LCM}((p\text{-}1), (q\text{-}1)))$.

That is, the inequality in (a) holds, and $1 \equiv ed\ (\text{LCM}((p\text{-}1), (q\text{-}1)))$.

In the extremely rare event that $d \le 2^{nlen/2}$, then new values for p, q and d **shall** be determined. A different value of e may be used, although this is not required.

B.3.2 Generation of Random Primes that are Provably Prime

An Approved method that satisfies the constraints of Appendix B.3.1 **shall** be used for the generation of IFC random primes p and q that are provably prime (see case A.1). One such method is provided in Appendix B.3.2.1 and B.3.2.2. For this method, a random seed is initially required (see Appendix B.3.2.1); the length of the seed is equal to twice the security strength associated with the modulus n. After the seed is obtained, the primes can be generated (see Appendix B.3.2.2).

B.3.2.1 Get the Seed

The following process or its equivalent **shall** be used to generate the seed for this method.

Input:

nlen	The intended bit length of the modulus n.

Output:

status	The status to be returned, where *status* is either **SUCCESS** or **FAILURE**.
seed	The seed. If *status* = **FAILURE**, a value of zero is returned as the *seed*.

Process:

1. If *nlen* is not valid (see Section 5.1), then Return (**FAILURE**, 0).
2. Let *security_strength* be the security strength associated with *nlen*, as specified in SP 800-57, Part 1.
2. Obtain a string *seed* of (2 × *security_strength*) bits from an **RBG** that supports the *security_strength*.
3. Return (**SUCCESS**, *seed*).

B.3.2.2 Construction of the Provable Primes *p* and *q*

The following process or its equivalent **shall** be used to construct the random primes p and q (to be used as factors of the RSA modulus n) that are provably prime:

Input:

nlen	The intended bit length of the modulus n.
e	The public verification exponent.
seed	The seed.

Output:

status	The status of the generation process, where *status* is either **SUCCESS** or **FAILURE**. When **FAILURE** is returned, zero values **shall** be returned as the other parameters.
p and *q*	The private prime factors of *n*.

Process:

1. If *nlen* is neither 2048 nor 3072, then return (**FAILURE**, 0, 0).
2. If $((e \le 2^{16})$ OR $(e \ge 2^{256}))$, then return (**FAILURE**, 0, 0).
3. Set the value of *security_strength* in accordance with the value of *nlen*, as specified in SP 800-57, Part 1.
4. If (**len**(*seed*) $\neq 2 \times$ *security_strength*), then return (**FAILURE**, 0, 0).
5. *working_seed* = *seed*.
6. Generate *p*:

   6.1 Using $L = nlen/2$, $N_1 = 1$, $N_2 = 1$, *first_seed* = *working_seed* and *e*, use the provable prime construction method in Appendix C.10 to obtain *p* and *pseed*. If **FAILURE** is returned, then return (**FAILURE**, 0, 0).

   6.2 *working_seed* = *pseed*.

7. Generate *q*:

   7.1 Using $L = nlen/2$, $N_1 = 1$, $N_2 = 1$, *first_seed* = *working_seed* and *e*, use the provable prime construction method in Appendix C.10 to obtain *q* and *qseed*. If **FAILURE** is returned, then return (**FAILURE**, 0, 0).

   7.2 *working_seed* = *qseed*.

8. If ($|p - q| \le 2^{nlen/2 - 100}$), then go to step 7.
9. Zeroize the internally generated seeds:

   9.1 *pseed* = 0;

   9.2 *qseed* = 0;

   9.3 *working_seed* = 0.

10. Return (**SUCCESS**, *p*, *q*).

B.3.3 Generation of Random Primes that are Probably Prime

An Approved method that satisfies the constraints of Appendix B.3.1 **shall** be used for the generation of IFC random primes *p* and *q* that are probably prime (see case A.2).

The following process or its equivalent **shall** be used to construct the random probable primes *p* and *q* (to be used as factors of the RSA modulus *n*):

Input:

nlen	The intended bit length of the modulus *n*.
e	The public verification exponent.

Output:

status	The status of the generation process, where status is either **SUCCESS** or **FAILURE**.
p and *q*	The private prime factors of *n*. When **FAILURE** is returned, zero values **shall** be returned as *p* and *q*.

Process:

1. If *nlen* is neither 2048 nor 3072, return (**FAILURE**, 0, 0).
2. If (($e \leq 2^{16}$) OR ($e \geq 2^{256}$)), then return (**FAILURE**, 0, 0).
3. Set the value of *security_strength* in accordance with the value of *nlen*, as specified in SP 800-57, Part 1.
4. Generate *p*:
   - 4.1 Obtain a string *p* of (*nlen*/2) bits from an **RBG** that supports the *security_strength*.
   - 4.2 If (*p* is not odd), then $p = p + 1$.
   - 4.3 If (($p < (\sqrt{2})(2^{(nlen/2)-1})$) OR (($p$-1) is not relatively prime to e)), then go to step 4.1.
   - 4.4 Test *p* for primality as specified in Appendix C.3 using an appropriate value from Table C-2 or C-3 in Appendix C.3 as the number of *iterations*. If *p* is **COMPOSITE**, go to step 4.1.
5. Generate *q*:
   - 5.1 Obtain a string *q* of (*nlen*/2) bits from an **RBG** that supports the *security_strength*
   - 5.2 If (*q* is not odd), then $q = q + 1$.
   - 5.3 If ($|p - q| \leq 2^{nlen/2 - 100}$), then go to step 5.1.
   - 5.4 If (($q < (\sqrt{2})(2^{(nlen/2)-1})$) OR (($q$-1) is not relatively prime to e)), then go to step 5.1.
   - 5.5 Test *q* for primality as specified in Appendix C.3 using an appropriate value from Table C.2 or C.3 as the number of *iterations*. If *q* is **COMPOSITE**, go to step 5.1.
6. Return (**SUCCESS**, *p*, *q*).

B.3.4 Generation of Provable Primes with Conditions Based on Auxiliary Provable Primes

This section specifies an Approved method for the generation of the IFC primes *p* and *q* with the additional conditions specified in Appendix B.3.1, case B.1, where p, p_1, p_2, q, q_1 and q_2 are all provable primes. For this method, a random seed is initially required (see Appendix B.3.2.1); the length of the seed is equal to twice the security strength associated with the modulus *n*. After the first seed is obtained, the primes can be generated.

Let *$bitlen_1$*, *$bitlen_2$*, *$bitlen_3$*, and *$bitlen_4$* be the bit lengths for p_1, p_2, q_1 and q_2, respectively, in accordance with Table B.1. The following process or its equivalent **shall** be used to generate the provable primes:

Input:

nlen	The intended bit length of the modulus *n*.
e	The public verification exponent.
seed	The seed.

Output:

status	The status of the generation process, where status is either **SUCCESS** or **FAILURE**. If **Failure** is returned then zeros **shall** be returned as the values for *p* and *q*.
p and *q*	The private prime factors of *n*.

Process:

1. If *nlen* is neither 1024, 2048, nor 3072, then return (**FAILURE**, 0, 0).
2. If (($e \leq 2^{16}$) OR ($e \geq 2^{256}$)), then return (**FAILURE**, 0, 0).
3. Set the value of *security_strength* in accordance with the value of *nlen*, as specified in SP 800-57, Part 1.
4. If (**len**(*seed*) $\neq 2 \times$ *security_strength*), then return (**FAILURE**, 0, 0).
5. *working_seed* = *seed*.
6. Generate *p*:

   6.1 Using $L = nlen/2$, $N_1 = bitlen_1$, $N_2 = bitlen_2$, *firstseed* = *working_seed* and *e*, use the provable prime construction method in Appendix C.10 to obtain *p*, p_1, p_2 and *pseed*. If **FAILURE** is returned, return (**FAILURE**, 0, 0).

   6.2 *working_seed* = *pseed.*
7. Generate *q*:

   7.1 Using $L = nlen/2\ L$, $N_1 = bitlen_3$, $N_2 = bitlen_4$ and *firstseed* = *working_seed* and *e*, use the provable prime construction method in Appendix C.10 to obtain *q*, q_1, q_2 and *qseed*. If **FAILURE** is returned, return (**FAILURE**, 0, 0).

   7.2 *working_seed* = *qseed.*
8. If ($|p - q| \leq 2^{nlen/2 - 100}$), then go to step 7.
9. Zeroize the internally generated seeds:

   9.1 *pseed* = 0.

   9.2 *q_seed* = 0.

9.3 *working_seed* = 0.

10. Return (**SUCCESS**, *p*, *q*).

B.3.5 Generation of Probable Primes with Conditions Based on Auxiliary Provable Primes

This section specifies an Approved method for the generation of the IFC primes *p* and *q* with the additional conditions specified in Appendix B.3.1, case B.2, where p_1, p_2, q_1 and q_2 are provably prime, and *p* and *q* are probably prime. For this method, a random seed is initially required (see Appendix B.3.2.1); the length of the seed is equal to twice the security strength associated with the modulus *n*. After the first seed is obtained, the primes can be generated.

Let $bitlen_1$, $bitlen_2$, $bitlen_3$, and $bitlen_4$ be the bit lengths for p_1, p_2, q_1 and q_2, respectively in accordance with Table B.1. The following process or its equivalent **shall** be used to construct *p* and *q*.

Input:

nlen	The intended bit length of the modulus *n*.
e	The public verification exponent.
seed	The seed.

Output:

status	The status of the generation process, where status is either **SUCCESS** or **FAILURE**. If **Failure** is returned then zeros **shall** be returned as the values for *p* and *q*.
p and *q*	The private prime factors of *n*.

Process:

1. If *nlen* is neither 1024, 2048, nor 3072, then return (**FAILURE**, 0, 0).
2. If (($e \leq 2^{16}$) OR ($e \geq 2^{256}$)), then return (**FAILURE**, 0, 0).
3. Set the value of *security_strength* in accordance with the value of *nlen*, as specified in SP 800-57, Part 1.
4. If (**len**(*seed*) $\neq 2 \times$ *security_strength*), then return (**FAILURE**, 0, 0).

Comment: Generate four primes p_1, p_2, q_1 and q_2 that are provably prime.

5. Generate *p*:

   5.1 Using $bitlen_1$ as the length, and *seed* as the *input_seed*, use the random prime generation routine in Appendix C.6 to obtain p_1 and *prime_seed*. If **FAILURE** is returned, the return (**FAILURE**, 0, 0).

   5.2 Using $bitlen_2$ as the length, and *prime_seed* as the *input_seed*, use the random prime generation routine in Appendix C.6 to obtain p_2 and a new value for *prime_seed*. If **FAILURE** is returned, the return (**FAILURE**, 0, 0).

5.3 Generate a prime p using the routine in Appendix C.9 with inputs of p_1, p_2, *nlen, e* and *security_strength*, also obtaining X_p. If **FAILURE** is returned, return (**FAILURE**, 0, 0).

6. Generate q:

   6.1. Using $bitlen_3$ as the length, and *prime_seed* as the *input_seed*, use the random prime generation routine in Appendix C.6 to obtain q_1 and a new value for *prime_seed*. If **FAILURE** is returned, the return (**FAILURE**, 0, 0).

   6.2 Using $bitlen_4$ as the length, and *prime_seed* as the *input_seed*, use the random prime generation routine in Appendix C.6 to obtain q_2 and a new value for *prime_seed*. If **FAILURE** is returned, the return (**FAILURE**, 0, 0).

   6.3 Generate a prime q using the routine in Appendix C.9 with inputs of q_1, q_2, *nlen, e* and *security_strength*, also obtaining X_q.. If **FAILURE** is returned, return (**FAILURE**, 0, 0).

7. If $((|p - q| \le 2^{nlen/2 - 100})$ OR $(|X_p - X_q| \le 2^{nlen/2 - 100}))$, then go to step 6.

8. Zeroize the random values used to generate p and q:

   8.1 $X_p = 0$.

   8.2 $X_q = 0$.

   8.3 *prime_seed* = 0.

9. Return (**SUCCESS**, p, q).

B.3.6 Generation of Probable Primes with Conditions Based on Auxiliary Probable Primes

An Approved method that satisfies the constraints of Appendix B.3.1 **shall** be used for the generation of IFC primes p and q that are probably prime and meet the additional constraints (see case B.3). For this case, the prime factors p_1, p_2, q_1 and q_2 are also probably prime.

Four random numbers X_{p1}, X_{p2}, X_{q1} and X_{q2} are generated, from which the prime factors p_1, p_2, q_1 and q_2 are determined. p_1 and p_2, and an additional random number X_p are then used to determine p, and q_1 and q_2 and a random number X_q are used to obtain q. Let $bitlen_1$, $bitlen_2$, $bitlen_3$, and $bitlen_4$ be the bit lengths for p_1, p_2, q_1 and q_2, respectively in accordance with Table B.1.

The following process or its equivalent **shall** be used to generate p and q:

Input:

nlen	The intended bit length of the modulus n.
e	The public verification exponent.

Output:

status	The status of the generation process, where status is either **SUCCESS** or **FAILURE**. If **Failure** is returned then zeros **shall** be returned as the values for *p* and *q*.
p and *q*	The private prime factors of *n*.

Process:

1. If *nlen* is neither 1024, 2048, nor 3072, then return (**FAILURE**, 0, 0).
2. If (($e \leq 2^{16}$) OR ($e \geq 2^{256}$)), then return (**FAILURE**, 0, 0).
3. Set the value of *security_strength* in accordance with the value of *nlen*, as specified in SP 800-57, Part 1.
4. Generate *p*:

   4.1 Generate an odd integer X_{p1} of length $bitlen_1$ bits, and a second odd integer X_{p2} of length $bitlen_2$ bits, using an Approved random number generator that supports the *security_strength*.

   4.2 Sequentially search successive odd integers, starting at X_{p1} until the first probable prime p_1 is found. Candidate integers **shall** be tested for primality as specified in Appendix C.3. Repeat the process to find p_2, starting at X_{p2}. The probable primes p_1 and p_2 **shall** be the first integers that pass the primality test.

   4.3 Generate a prime *p* using the routine in Appendix C.9 with inputs of p_1, p_2, *nlen*, *e* and *security_ strength*, also obtaining X_p. If **FAILURE** is returned, return (**FAILURE**, 0, 0).

5. Generate *q*:

   5.1 Generate an odd integer X_{q1} of length $bitlen_3$ bits, and a second odd integer X_{q2} of length $bitlen_4$ bits, using an Approved random number generator that supports the *security_strength*.

   5.2 Sequentially search successive odd integers, starting at X_{q1} until the first probable prime q_1 is found. Candidate integers **shall** be tested for primality as specified in Appendix C.3. Repeat the process to find q_2, starting at X_{q2}. The probable primes q_1 and q_2 **shall** be the first integers that pass the primality test.

   5.3 Generate a prime *q* using the routine in Appendix C.9 with inputs of q_1, q_2, *nlen, e* and *security_ strength*, also obtaining X_q. If **FAILURE** is returned, return (**FAILURE**, 0, 0).

6. If (($|X_p - X_q| \leq 2^{nlen/2 - 100}$) OR ($|p - q| \leq 2^{nlen/2 - 100)}$)), then go to step 5.
7. Zeroize the random values used to generate *p* and *q*:

   7.1 $X_p = 0$.

   7.2 $X_q = 0$.

8. Return (**SUCCESS**, *p*, *q*).

C.3 Probabilistic Primality Tests

A probabilistic primality test may be required during the generation and validation of prime numbers. An Approved robust probabilistic primality test **shall** be selected and used.

There are several probabilistic algorithms available. The Miller-Rabin probabilistic primality tests described in Appendices C.3.1 and C.3.2 are versions of a procedure due to M.O. Rabin, based in part on ideas of Gary L. Miller. For more information, see Knuth, The Art of Computer Programming, Vol. 2, 3rd Ed., Addison-Wesley, 1998, Algorithm P, page 395. For these tests, let **RBG** be an Approved random bit generator (see SP 800-90).

There are several Lucas probabilistic primality tests available; the version provided in Appendix C.3.3 was described in the Mathematics of Computation, V. 35 (1980), pages 1391 – 1417 by Baillie and Wagstaff.

This Standard allows two alternatives for testing primality: either using several iterations of only the Miller-Rabin test, or using the iterated Miller-Rabin test, followed by a single Lucas test. The value of *iterations* (as used in Appendices C.3.1 and C.3.2) depends on the algorithm being used, the security strength, the error probability used, the length (in bits) of the candidate prime and the type of tests to be performed. Tables C.1, C.2 and C.3 list the minimum number of *iterations* of the Miller-Rabin tests that **shall** be performed.

As stated in Appendix F.3, if the definition of the error probability that led to the values of the number of Miller-Rabin tests for p and q in Tables C.1, C.2 and C.3 is not conservative enough, the prescribed number of Miller-Rabin tests can be followed by a single Lucas test. The Lucas test is not necessary when testing the p_1, p_2, q_1 and q_2 values for primality when generating RSA primes. Since there are no known non-prime values that pass the two test combination (i.e., the indicated number of rounds of the Miller-Rabin test with randomly selected bases, followed by one round of the Lucas test), the two test combination may provide additional assurance of primality over the use of only the Miller-Rabin test. For DSA, the two-test combination may provide better performance. See Appendix F for further information.

Table C.1. Minimum number of Miller-Rabin iterations for DSA

Parameters	M-R Tests Only	M-R Tests when followed by One Lucas test
p: 1024 bits q: 160 bits Error probability = 2^{-80}	For p and q: 40	For p: 3 For q: 19
p: 2048 bits q: 224 bits Error probability = 2^{-112}	For p and q: 56	For p: 3 For q: 24

p: 2048 bits *q*: 256 bits Error probability = 2^{-112}	For *p* and *q*: 56	For *p*: 3 For *q*: 27
p: 3072 bits *q*: 256 bits Error probability = 2^{-128}	For *p* and *q*: 64	For *p*: 2 For *q*: 27

Table C.2. Minimum number of rounds of M-R testing when generating primes for use in RSA Digital Signatures

Parameters	M-R Tests Only
p_1, p_2, q_1 and q_2 > 100 bits *p* and *q*: 512 bits Error probability = 2^{-80}	For p_1, p_2, q_1 and q_2: 28 For *p* and *q*: 5
p_1, p_2, q_1 and q_2 > 140 bits *p* and *q*: 1024 bits Error probability = 2^{-112}	For p_1, p_2, q_1 and q_2: 38 For *p* and *q*: 5
p_1, p_2, q_1 and q_2 > 170 bits *p* and *q*: 1536 bits Error probability = 2^{-128}	For p_1, p_2, q_1 and, q_2: 41 For *p* and *q*: 4

Table C.3. Minimum number of rounds of M-R testing when generating primes for use in RSA Digital Signatures using an error probability of 2^{-100}

Parameters	M-R Tests Only
p_1, p_2, q_1 and q_2 > 100 bits *p* and *q*: 512	For p_1, p_2, q_1 and q_2: 38 For *p* and *q*: 7
p_1, p_2, q_1 and q_2 > 140 bits *p* and *q*: 1024 bits	For p_1, p_2, q_1 and q_2: 32 For *p* and *q*: 4
p_1, p_2, q_1 and q_2 > 170 bits *p* and *q*: 1536 bits	For p_1, p_2, q_1 and q_2: 27 For *p* and *q*: 3

C.3.1 Miller-Rabin Probabilistic Primality Test

The following process or its equivalent **shall** be used as the Miller-Rabin test. Let **RBG** be an Approved random bit generator (see SP 800-90).

Input:

1. *w*	The odd integer to be tested for primality. This will be either *p* or *q*, or one of the auxiliary primes p_1, p_2, q_1 or q_2.
2. *iterations*	The number of iterations of the test to be performed; the value **shall** be consistent with Table C.1, C.2 or C.3.

Output:

1. *status*	The status returned from the validation procedure, where status is either **PROBABLY PRIME** or **COMPOSITE**.

Process:

```
1. Let $a$ be the largest integer such that $2^a$ divides $w-1$.
2. $m = (w-1) / 2^{a.}$
3. $wlen$ = len ($w$).
4. For $i$ = 1 to $iterations$ do
    4.1 Obtain a string $b$ of $wlen$ bits from an RBG.
                                  Comment: Ensure that $1 < b < w-1$.
    4.2 If (($b \leq 1$) or ($b \geq w-1$)), then go to step 4.1.
    4.3 $z = b^m \bmod w$.
    4.4 If (($z = 1$) or ($z = w - 1$)), then go to step 4.7.
    4.5 For $j$ = 1 to $a - 1$ do.
        4.5.1 $z = z^2 \bmod w$.
        4.5.2 If ($z = w-1$), then go to step 4.7.
        4.5.3 If ($z = 1$), then go to step 4.6.
    4.6 Return COMPOSITE.
    4.7 Continue.                 Comment: Increment $i$ for the do-
                                  loop in step 4.
5. Return PROBABLY PRIME.
```

C.3.2 Enhanced Miller-Rabin Probabilistic Primality Test

The following process or its equivalent **shall** be used as the Enhanced Miller-Rabin test. This method provides additional information when an error is encountered that may be useful when generating or validating RSA moduli. Let **RBG** be an Approved random bit generator (see SP 800-90).

Input:

1.	*w*	The odd integer to be tested for primality. This will be either p or q , or one of the auxiliary primes p_1, p_2, q_1 or q_2..
2.	*iterations*	The number of iterations of the test to be performed; the value **shall** be consistent with Table C.1, C.2 or C.3.

Output:

1.	*status*	The status returned from the validation procedure, where status is either **PROBABLY PRIME**, **PROVABLY COMPOSITE WITH FACTOR** (returned with the factor), and **PROVABLY COMPOSITE AND NOT A POWER OF A PRIME**.

Process:

```
1. Let $a$ be the largest integer such that $2^a$ divides $w$-1.
2. $m = (w-1) / 2^{a.}$
3. $wlen$ = **len** ($w$).
4. For $i = 1$ to $iterations$ do
    4.1   Obtain a string $b$ of $wlen$ bits from an RBG.
                                        Comment: Ensure that $1 < b < w-1$.
    4.2   If (($b \le 1$) or ($b \ge w-1$)), then go to step 4.1.
    4.3   $g$ = **GCD** ($b$, $w$).
    4.4   If ($g > 1$), then return **PROVABLY COMPOSITE WITH FACTOR**
          and the value of $g$.
    4.5   $z = b^m \bmod w$.
    4.6   If (($z = 1$) or ($z = w - 1$)), then go to step 4.15.
    4.7   For $j = 1$ to $a - 1$ do.
        4.7.1   $x = z$.              Comment: $x \neq 1$ and $x \neq w-1$.
        4.7.2   $z = x^2 \bmod w$.
        4.7.3   If ($z = w-1$), then go to step 4.15.
        4.7.4   If ($z = 1$), then go to step 4.12.
    4.8   $x = z$.                    Comment: $x = b^{(w-1)/2} \bmod w$ and $x \neq w-1$.
    4.9   $z = x^2 \bmod w$.
    4.10  If ($z = 1$), then go to step 4.12.
    4.11  $x = z$.                    Comment: $x = b^{(w-1)} \bmod w$ and $x \neq 1$.
    4.12  $g$ = **GCD** ($x$-1, $w$).
    4.13  If ($g > 1$), then return **PROVABLY COMPOSITE WITH FACTOR**
          and the value of $g$.
    4.14  Return **PROVABLY COMPOSITE AND NOT A POWER OF A PRIME.**
    4.15  Continue.                 Comment: Increment i for the do-loop in step 4.
5. Return **PROBABLY PRIME.**
```

C.3.3 (General) Lucas Probabilistic Primality Test

The following process or its equivalent **shall** be used as the Lucas test.

Input:

C	The candidate odd integer to be tested for primality.

Output:

status	Where status is either **PROBABLY PRIME** or **COMPOSITE**.

Process:

```
1. Test whether $C$ is a perfect square (see Appendix C.4). If so, return (COMPOSITE).
2. Find the first $D$ in the sequence $\{5, -7, 9, -11, 13, -15, 17, \ldots\}$ for which the Jacobi symbol $\left(\frac{D}{C}\right) = -1$. See Appendix C.5 for an Approved method to compute the Jacobi Symbol. If $\left(\frac{D}{C}\right) = 0$ for any $D$ in the sequence, return (COMPOSITE).
3. $K = C+1$.
4. Let $K_r K_{r-1} \ldots K_0$ be the binary expansion of $K$, with $K_r = 1$.
5. Set $U_r = 1$ and $V_r = 1$.
6. For $i = r-1$ to 0, do
    6.1  $U_{temp} = U_{i+1} V_{i+1} \bmod C$.
    6.2  $V_{temp} = \frac{V_{i+1}{}^2 + DU_{i+1}{}^2}{2} \bmod C$.
    6.3  If ($K_i = 1$), then        Comment: If $K_i = 1$, then do steps 6.3.1 and 6.3.2; otherwise, do steps 6.3.3 and 6.3.4.
        6.3.1  $U_i = \frac{U_{temp} + V_{temp}}{2} \bmod C$.
        6.3.2  $V_i = \frac{V_{temp} + DU_{temp}}{2} \bmod C$.
    Else
        6.3.3  $U_i = U_{temp}$.
        6.3.4  $V_i = V_{temp}$.
```

7. If ($U_0 = 0$), then return (**PROBABLY PRIME**). Otherwise, return (**COMPOSITE**).

Steps 6.2, 6.3.1 and 6.3.2 contain expressions of the form $A/2 \bmod C$, where A is an integer, and C is an odd integer. If $A/2$ is not an integer (i.e., A is odd), then $A/2 \bmod C$ may be calculated as $(A+C)/2 \bmod C$. Alternatively, $A/2 \bmod C = A\cdot(C+1)/2 \bmod C$, for any integer A, without regard to A being odd or even.

C.6 Shawe-Taylor Random_Prime Routine

This routine is recursive and may be used to construct a provable prime number using a hash function.

Let **Hash ( )** be the selected hash function for the (L, N) pair, and let *outlen* be the bit length of the hash function output block. The following process or its equivalent **shall** be used to generate a prime number for this constructive method.

ST_Random_Prime ( ):

Input:

1.	*length*	The length of the prime to be generated.
2.	*input_seed*	The seed to be used for the generation of the requested prime.

Output:

1.	*status*	The status returned from the generation routine, where *status* is either **SUCCESS** or **FAILURE.** If Failure is returned, then zeros are returned as the other output values.
2.	*prime*	The requested prime.
3	*prime_seed*	A seed determined during generation.
4.	*prime_gen_counter*	(Optional) A counter determined during the generation of the prime.

Process:

```
1. If ($length < 2$), then return (FAILURE, 0, 0 {, 0}).
2. If ($length \geq 33$), then go to step 14.
3. $prime\_seed = input\_seed$.
4. $prime\_gen\_counter = 0$.
                                        Comment: Generate a pseudorandom
                                        integer c of length bits.
5. $c = \mathbf{Hash}(prime\_seed) \oplus \mathbf{Hash}(prime\_seed + 1)$.
6. $c = 2^{length - 1} + (c \bmod 2^{length - 1})$.
```

```
7.  $c = (2 * \lfloor c / 2 \rfloor) + 1$.        Comment: Set prime to the least odd integer greater than or equal to c.
8.  $prime\_gen\_counter = prime\_gen\_counter + 1$.
9.  $prime\_seed = prime\_seed + 2$.
10. Perform a deterministic primality test on $c$. For example, since $c$ is small, its
    primality can be tested by trial division. See Appendix C.7.
11. If ($c$ is a prime number), then
    11.1  $prime = c$.
    11.2  Return (**SUCCESS**, $prime$, $prime\_seed$ {, $prime\_gen\_counter$}).
12. If ($prime\_gen\_counter > (4 * length)$), then return **FAILURE**
13. Go to step 5.
14. ($status$, $c_0$, $prime\_seed$, $prime\_gen\_counter$) = (**ST_Random_Prime** (($\lceil length / 2 \rceil + 1$), $input\_seed$).
15. If **FAILURE** is returned, return (**FAILURE**, 0, 0 {, 0}).
16. $iterations = \lceil length / outlen \rceil - 1$.
17. $old\_counter = prime\_gen\_counter$.
                                                   Comment: Generate a pseudorandom integer x in the interval $[2^{length - 1}, 2^{length}]$.
18. $x = 0$.
19. For $i = 0$ to $iterations$ do
        $x = x + (\textbf{Hash}(prime\_seed + i) * 2^{i \times outlen})$.
20. $prime\_seed = prime\_seed + iterations + 1$.
21. $x = 2^{length - 1} + (x \bmod 2^{length - 1})$.
                                                   Comment: Generate a candidate prime c in the interval $[2^{length - 1}, 2^{length}]$.
22. $t = \lceil x / (2c_0) \rceil$.
23. If ($2tc_0 + 1 > 2^{length}$), then $t = \lceil 2^{length - 1} / (2c_0) \rceil$.
24. $c = 2tc_0 + 1$.
25. $prime\_gen\_counter = prime\_gen\_counter + 1$.
                                                   Comment: Test the candidate prime c for primality; first pick an integer a between 2 and c - 2.
```

```
26. $a = 0$.
27. For $i = 0$ to $iterations$ do
      $a = a + (\textbf{Hash}(prime\_seed + i) * 2^{i \times outlen})$.
28. $prime\_seed = prime\_seed + iterations + 1$.
29. $a = 2 + (a \bmod (c - 3))$.
30. $z = a^{2t} \bmod c$.
31. If $((1 = \textbf{GCD}(z - 1, c))$ and $(1 = z^{c_0} \bmod c))$, then
      31.1  $prime = c$.
      31.2  Return (**SUCCESS**, $prime$, $prime\_seed$ {, $prime\_gen\_counter$}).
32. If $(prime\_gen\_counter \geq ((4 * length) + old\_counter))$, then return
    (**FAILURE**, 0, 0 {, 0}).
33. $t = t + 1$.
34. Go to step 23.
```

C.9 Compute a Probable Prime Factor Based on Auxiliary Primes

This routine constructs a probable prime (a candidate for p or q) using two auxiliary prime numbers and the Chinese Remainder Theorem (CRT).

Input:

r_1 and r_2	Two odd prime numbers satisfying $\log_2(r_1 r_2) \leq (nlen/2) - \log_2(nlen/2) - 6$.
nlen	The desired length of n, the RSA modulus.
e	The public verification exponent.
security_strength	The minimum security strength required for random number generation.

Output:

status	The status returned from the generation procedure, where *status* is either **SUCCESS** or **FAILURE**. If **FAILURE** is returned, then zeros are returned as the other output values.
private_prime_factor	The prime factor of n.
X	The random number used during the generation of the *private_prime_factor*.

Process:

```
1.  $R = ((r_2^{-1} \bmod 2r_1) \times r_2) - (((2r_1)^{-1} \bmod r_2) \times 2r_1)$.
                                   Comment: Apply the CRT, so that $R \equiv 1$
                                   mod $2r_1$ and $R \equiv -1 \bmod r_2$.
```

```
2. Generate a random number $X$ using an Approved random number generator
   that supports the security_ strength, such that
   $(\sqrt{2})(2^{nlen/2-1}) \le X \le (2^{nlen/2} - 1)$.
3. $Y = X + ((R - X) \bmod 2r_1r_2)$.
       Comment: $Y$ is the first integer $\ge X$, such that $r_1$ is a large
       prime factor of $Y$-1, and $r_2$ is a large prime factor of $Y$+1.
       Comment: Determine the requested prime number by constructing
       candidates from a sequence and performing primality tests.
4. $i = 0$.
5. If ($Y \ge 2^{nlen/2}$), then go to step 2.
6. If (GCD ($Y$-1, $e$) = 1), then
   6.1 Check the primality of $Y$ as specified in Appendix C.3. If COMPOSITE
       is returned, go to step 7.
   6.2 private_prime_factor = $Y$.
   6.3 Return (SUCCESS, private_prime_factor, $X$).
7. $i = i + 1$.
8. If ($i \ge 5(nlen/2)$), then return (FAILURE, 0, 0).
9. $Y = Y + (2r_1r_2)$.
10. Go to step 5.
```

C.10 Construct a Provable Prime (possibly with Conditions), Based on Contemporaneously Constructed Auxiliary Provable Primes

The following process (or its equivalent) **shall** be used to generate an L-bit provable prime p (a candidate for one of the prime factors of an RSA modulus). Note that the use of p in this specification is used generically; both RSA prime factors p and q may be generated using this method.

If a so-called "strong prime" is required, this process can generate primes p_1 and p_2 (of specified bit-lengths N_1 and N_2) dividing $p-1$ and $p+1$, respectively. The resulting prime p will satisfy the conditions traditionally required of a strong prime, provided that the requested bit-lengths for p_1 and p_2 have appropriate sizes.

Regardless of the bit-lengths selected for p_1 and p_2, the quantity $p-1$ will have a prime divisor p_0 whose bit-length is slightly more than half that of p. In addition, the quantity p_0-1 will have a prime divisor whose bit-length is slightly more than half that of p_0.

This algorithm requires that $N_1 + N_2 \le L - \lceil L/2 \rceil - 4$. Values for N_1 and N_2 **should** be chosen such that $N_1 + N_2 \le (L/2) - \log_2(L) - 7$, to ensure that the algorithm can generate as many as $5L$ distinct candidates for p.

Let **Hash** be the selected hash function to be used, and let *outlen* be the bit length of the hash function output block.

Provable_Prime_Construction():

Input:

1. L	A positive integer equal to the requested bit-length for p. Note that acceptable values for L= nlen/2 are computed as specified in Appendix B.3.1, criteria 2(b) and (c), with *nlen* assuming a value specified in Table B.1.
2. N_1	A positive integer equal to the requested bit-length for p_1. If $N_1 \geq 2$, then p_1 is an odd prime of N_1 bits; otherwise, $p_1 = 1$. Acceptable values for $N_1 \geq 2$ are provided in Table B.1
3. N_2	A positive integer equal to the requested bit-length for p_2. If $N_2 \geq 2$, then p_2 is an odd prime of N_2 bits; otherwise, $p_2 = 1$. Acceptable values for $N_2 \geq 2$ are provided in Table B.1
4. *firstseed*	A bit string equal to the first seed to be used.
5. e	The public verification exponent.

Output:

1. *status*	The status returned from the generation procedure, where *status* is either **SUCCESS** or **FAILURE.** If **FAILURE** is returned, then zeros are returned as the other output values.
2. p, p_1, p_2	The required prime p, along with p_1 and p_2 having the property that p_1 divides $p–1$ and p_2 divides $p+1$.
3. *pseed*	A seed determined during generation.

Process:

1. If L, N_1, and N_2 are not acceptable, then, return (**FAILURE**, 0, 0, 0, 0).

   Comment: Generate p_1 and p_2, as well as the prime p_0.

2. If $N_1 = 1$, then

   2.1 $p_1 = 1$.

   2.2 $p_2seed = firstseed$.

3. If $N_1 \geq 2$, then

   3.1 Using N_1 as the length and *firstseed* as the *input_seed*, use the random prime generation routine in Appendix C.6 to obtain p_1 and p_2seed.

   3.2 If **FAILURE** is returned, then return (**FAILURE**, 0, 0, 0, 0).

4. If $N_2 = 1$, then

   4.1 $p_2 = 1$.

   4.2 $p_0seed = p_2seed$.

```
5. If $N_2 \geq 2$, then
    5.1 Using $N_2$ as the length and $p_2seed$ as the *input_seed*, use the random prime generation routine in Appendix C.6 to obtain $p_2$ and $p_0seed$.
    5.2 If **FAILURE** is returned, then return (**FAILURE**, 0, 0, 0, 0).
6. Using $\lceil L / 2 \rceil + 1$ as the length and $p_0seed$ as the *input_seed*, use the random prime generation routine in Appendix C.6 to obtain $p_0$ and *pseed*. If **FAILURE** is returned, then return (**FAILURE**, 0, 0, 0, 0).
                                        Comment: Generate a (strong) prime p in
                                        the interval $[(\sqrt{2})(2^{L-1}), 2^L - 1]$.
7. $iterations = \lceil L / outlen \rceil - 1$.
8. $pgen\_counter = 0$.
                                        Comment: Generate pseudo-random x in the
                                        interval $[(\sqrt{2})(2^{L-1}) - 1, 2^L - 1]$.
9. $x = 0$.
10. For $i = 0$ to *iterations* do
        $x = x + (\mathbf{Hash}(pseed + i)) * 2^{i \times outlen}$.
11. $pseed = pseed + iterations + 1$.
12. $x = \lfloor(\sqrt{2})(2^{L-1})\rfloor + (x \bmod (2^L - \lfloor(\sqrt{2})(2^{L-1})\rfloor))$.
                                        Comment: Generate a candidate for the
                                        prime p.
13. Compute $y$ in the interval $[1, p_2]$ such that $1 = (y\, p_0\, p_1) \bmod p_2$.
14. $t = \lceil((2 y\, p_0\, p_1) + x)/(2\, p_0\, p_1\, p_2)\rceil$.
15. If $((2(t\, p_2 - y)\, p_0\, p_1 + 1) > 2^L)$, then
        $t = \lceil( (2 y\, p_0\, p_1) + \lfloor(\sqrt{2})(2^{L-1})\rfloor ) / (2\, p_0\, p_1\, p_2)\rceil$.
                                        Comment: p satisfies
                                            $0 = (p - 1) \bmod (2 p_0\, p_1)$ and
                                            $0 = (p + 1) \bmod p_2$.
16. $p = 2(t\, p_2 - y)\, p_0\, p_1 + 1$.
17. $pgen\_counter = pgen\_counter + 1$.
18. If $(\mathbf{GCD}(p-1, e) = 1)$, then
                                        Comment: Choose an integer a in the
                                        interval $[2, p - 2]$.
    18.1 $a = 0$
    18.2 For $i = 0$ to *iterations* do
```

```
            $a = a + (\mathbf{Hash}(pseed + i)) * 2^{i \times outlen}$.
    18.3  $pseed = pseed + iterations + 1$.
    18.4  $a = 2 + (a \bmod (p-3))$.
                                        Comment: Test $p$ for primality:
    18.5  $z = a^{2(t p_2 - y) p_1} \bmod p$.
    18.6  If $((1 = \mathbf{GCD}(z-1, p))$ and $(1 = (z^{p_0} \bmod p))$, then return (**SUCCESS**, $p$,
          $p_1$, $p_2$, $pseed$).
19. If ($pgen\_counter \geq 5L$), then return (**FAILURE**, 0, 0, 0, 0).
20. $t = t + 1$.
21. Go to step 15.
```

Appendix F: Calculating the Required Number of Rounds of Testing Using the Miller-Rabin Probabilistic Primality Test (Informative)

F.1 The Required Number of Rounds of the Miller-Rabin Primality Tests

The ideas of paper [1] were applied to estimate $p_{k,t}$, the probability that an odd *k*-bit integer that passes *t* rounds of Miller-Rabin (M-R) testing is actually composite. The probability $p_{k,t}$ is understood as the ratio of the number of odd composite numbers of a binary length *k* that pass *t* rounds of M-R testing (with randomly generated bases) to the total number of odd integers of binary length k. This is equivalent to assuming that candidates selected for testing will be chosen uniformly at random from the entire set of odd *k*-bit integers. From the perspective of a party charged with the responsibility of generating a *k*-bit prime, the objective is to determine a value of *t* such that $p_{k,t}$ is no greater than an acceptably small target value p_{target}.

Using [1], it is possible to compute an upper bound for $p_{k,t}$ as a function of *k* and *t*. From this, an upper bound can be computed for *t* as a function of *k* and p_{target}, the maximum allowed probability of accidentally generating a composite number. The following is an algorithm for computing *t*:

```
For $t$ = 1, 2 … (up to some stopping value needed to be able to stop when no number
        of rounds of M-R would suffice)
    For $M$ = 3, 4 … $\lfloor 2\sqrt{k-1} - 1 \rfloor$
        Compute $p_{k,t}$ as in (2).
        If $p_{k,t} \le p_{target}$
            Accept $t$.                                        (1)
            Stop.
        Endif
    Endfor
Endfor
```

In (1), *k* is the bit length of the candidate primes and (2) is as follows:

$$p_{k,t} = 2.00743 \cdot \ln(2) \cdot k \cdot 2^{-k} \left[2^{k-2-Mt} + \frac{8(\pi^2 - 6)}{3} 2^{k-2} \sum_{m=3}^{M} 2^{m-(m-1)t} \sum_{j=2}^{m} \frac{1}{2^{j+{(k-1)}/{j}}} \right] . \qquad (2)$$

Using this expression for *t*, the following methodologies are used for testing the DSA and the RSA candidate primes.

F.2 Generating DSA Primes

For DSA, the maximum possible care must be taken when generating the primes p and q that are used for the domain parameters. The same primes p and q are used by many parties. This means that any weakness that these numbers may possess would affect multiple users. It also means that the primes are not generated very often; typically, an entire system uses the same set of domain parameters. Therefore, in this case, some additional care is called for.

With this in mind, it may be too optimistic to assume that conditions allow a simple computation of t according to (1) and (2). It may be necessary to be more cautious and either include some additional testing (beyond the M-R tests) or use a more conservative estimate of the error probabilities associated with the M-R tests. This approach leads to the following strategies: either (A) use the number of M-R tests as calculated above and follow them with a single Lucas test (as recommended in ANS X9.31), or (B) base the choice of t on a different formulation of the probability of an error occurring in the M-R testing, leading to a more conservative course of action.

One approach for strategy (B) would be to adopt the viewpoint of the majority of system users, who have no part in generating the (supposed) prime, but who must rely upon its primality for their security. Such parties may be concerned that the candidates for M-R testing have been selected in a fashion that deviates significantly from the uniform distribution – which was assumed when determining t according to (1) and (2). In cases where the selection process could be unusually biased in some way, it is important to minimize the probability that a composite number will survive testing. It can be shown that for any k-bit odd composite number (regardless of how it was selected), the probability that it will pass t rounds of M-R testing with randomly chosen bases is less than 4^{-t} (although this is not a particularly tight bound). Selecting t such that $4^{-t} \le p_{target}$ is equivalent to choosing $t \ge -\log_2(p_{target})/2$. To ensure that a composite number has a probability no greater than p_{target} of surviving the M-R tests, the number of rounds can be set at $t = \lceil -\log_2(p_{target})/2 \rceil$. Even if the method of selecting candidates were so biased that it offered nothing but composite numbers for testing, it is reasonable to expect that it would take at least $1/p_{target}$ attempts (which is greater than 4^t) before a composite number would slip through the t-round M-R testing process.

WARNING: As the discussion above illustrates, care must be taken when using the phrase "error probability" in connection with the recommended number of rounds of M-R testing. The probability that a composite number survives t rounds of Miller-Rabin testing is <u>not</u> the same as $p_{k,t}$, which is the probability that a number surviving t rounds of Miller-Rabin testing is composite. Ordinarily, the latter probability is the one that should be of most interest to a party responsible for generating primes, while the former may be more important to a party responsible for validating the primality of a number generated by someone else. However, for sufficiently large k (e.g., $k \ge 51$), it can be shown that $p_{k,t} \le 4^{-t}$ under the same assumptions concerning the selection of candidates as those made to obtain formula (2). (See [1].) In such cases, $t = \lceil -\log_2(p_{target})/2 \rceil$ rounds of Miller-Rabin testing can be used both in generating and validating primes, with p_{target} serving as an upper bound on both the probability that the generation process yields a composite number and the probability that a composite number would survive an attempt to validate its primality.

Table C.1 in Appendix C.3 identifies the minimum values for t when generating the primes p and q forDSA using either strategy (A) or (B) above. To obtain the t values shown in the column titled “M-R Tests Only”, the conservative strategy (B) was followed; those t values are sufficient to validate the primality of p and q. The t values shown in the column titled “M-R Tests when followed by One Lucas Test” result from following strategy (A) using computations (1) and (2).

F.3 Generating Primes for RSA Signatures

When generating primes for the RSA signature algorithm, it is still very important to reduce the probability of errors in the M-R testing procedure. However, since the (probable) primes are used to generate a user’s key pair, if a composite number survives the testing process, the consequences of the error may be less dramatic than in the case of generating DSA domain parameters; only one user’s transactions are affected, rather than a domain of users. Furthermore, if the p or q value generated for some user is composite, the problem will not go undiscovered for long, since it is almost certain that signatures generated by that user will not be verifiable.

Therefore, when generating the RSA primes p and q, it is sufficient to use the number of rounds derived from (1) and (2) as the minimum number of M-R tests to be performed. However, if the definition of $p_{k,\,t}$ is not considered to be sufficiently conservative when testing p and q, it is recommended that the t rounds of Miller-Rabin tests be followed by a single Lucas test.

The lengths for p and q that are recommended for use in RSA signature algorithms are 512, 1024 and 1536 bits; recall that $n = pq$, so the corresponding lengths for n are 1024, 2048 and 3072 bits, respectively. As currently specified in SP 800-57, Part 1, these lengths correspond to security strengths of 80, 112 and 128 bits, respectively. Hence, it makes sense to match the number of rounds of Miller-Rabin testing to the target error probability values of 2^{-80}, 2^{-112}, and 2^{-128}. A probability of 2^{-100} is included for all prime lengths, since this probability has often been used in the past and may be acceptable for many applications.

When generating the RSA primes p and q with conditions, it is sufficient to use the value t derived from (1) and (2) as the minimum number of M-R tests to be performed when generating the auxiliary primes p_1, p_2, q_1 and q_2. It is not necessary to use an additional Lucas test on these numbers. In the extremely unlikely event that one of the numbers p_1, p_2, q_1 or q_2 is composite, there is still a high probability that the corresponding RSA prime (p or q) will satisfy the requisite conditions.

The sizes of p_1, p_2, q_1, and q_2 were chosen to ensure that, for an adversary with significant but not overwhelming resources, Lenstra’s elliptic curve factoring method [2] (against which there is no protection beyond choosing large p and q) is a more effective factoring algorithm than either the Pollard P-1 [2] method, the Williams P+1 method [3] or various cycling methods [2]. For an adversary with overwhelming resources, the best all-purpose factoring algorithm is assumed to be the General Number Field Sieve [2].

Tables C.2 and C.3 in Appendix C.3 specify the minimum number of rounds of M-R testing when generating primes to be used in the construction of RSA signature key pairs.

References

[1] I. Damgard, P. Landrock, and C. Pomerance, C. "Average Case Error Estimates for the Strong Provable Prime Test," Mathematics of Computation, v. 61, No, 203, pp. 177-194, 1993.

[2] A.J Menezes, P.C. Oorschot, and S.A. Vanstone. Handbook of Applied Cryptography. CRC Press, 1996.

[3] H.C. Williams. "A p+1 Method of factoring". *Math. Comp*. 39, 225-234, 1982.

[4] D.E. Knuth, The Art of Computer Programming, Vol. 2, 3rd Ed., Addison-Wesley, 1998, Algorithm P, page 395.

[5] R. Baillie and S.S. Wagstaff Jr.,Mathematics of Computation, V. 35 (1980), pages 1391 – 1417.

Definitions to be included in FIPS 186-3:

Probable prime	An integer that is believed to be prime, based on a probabilistic primality test. There should be no more than a negligible probability that the so-called probable prime is actually composite.
Provable prime	An integer that is either constructed to be prime or is calculated to be prime using a primality-proving algorithm.