Masaryk University
Faculty of Informatics

The properties of RSA key generation process in software libraries

Master's Thesis

Bc. Matúš Nemec

Brno, Spring 2016

Declaration

Hereby I declare that this paper is my original authorial work, which I have worked out on my own. All sources, references, and literature used or excerpted during elaboration of this work are properly cited and listed in complete reference to the due source.

Bc. Matúš Nemec

Advisor: RNDr. Petr Švenda, Ph.D.

Acknowledgement

Many thanks to Petr Švenda for his guidance and valuable advice.

Access to computing and storage facilities owned by parties and projects contributing to the National Grid Infrastructure MetaCentrum, provided under the programme "Projects of Large Research, Development, and Innovations Infrastructures" (CESNET LM2015042), is greatly appreciated.

Abstract

This thesis surveys cryptographic libraries and compares different approaches to RSA key generation, selection of a prime and primality testing. Attacks on RSA keys are examined and properties of weak keys are discussed. The algorithms used in practice are compared to recommendations and requirements defined in international standards. Distributions of RSA keys and primes produced by different implementations are matched to corresponding algorithms. Additional properties of the keys and the key generation process are examined in order to establish criteria which make it possible to guess an algorithm based on the output of a black-box key generator.

Keywords

RSA, cryptographic library, prime number generation, key generation, factorization, strong primes

Contents

1 Introduction

Key pairs generated for the RSA cryptosystem are often used for long periods of time. When RSA is chosen for TLS or secure email, a compromise of the private key leads to a loss of confidentiality of all past and future communication captured by an attacker. The correct choice of the RSA parameters is of utmost importance for the overall security of the application.

Many standards and publications concerning RSA exist; however, few give a precise algorithm for key generation. Across different publications, the requirements placed on the RSA key pairs are often contradictory. In this work we surveyed widely used cryptographic libraries and described the algorithms used in practice. While most of them adhere to at least some international standard (mostly due to weak requirements set by some standards), several implementations add their own peculiarities to the process. Additionally, some published techniques are not used in any reviewed open-source library. Fortunately, despite wide variety of algorithms, none of them break security of RSA, yet they are characterized by a lot of interesting statistical properties.

1.1 Previous works

Loebenberger and Nüsken [1] compared methods for generating RSA integers defined in several standards and a few cryptographic libraries. Based on substantial differences in the standards they concluded, that the exact method for generating RSA integers does not affect the security of the keys. They computed entropy of the output distribution of different algorithms and deemed it sufficiently high.

Some conditions on RSA primes often required by standards were examined by Silverman [2], who deemed them unnecessary. Rivest and Silverman reviewed the arguments in favour of strong primes again in [3], where they argue that random primes resist contemporary special-purpose factorization methods. Even after many years of gradual improvements in factoring large numbers, the arguments are still valid, as indicated by current factorization record of 768-bit RSA key [4], which used the Number Field Sieve, a general-factorization method.

The argument, that some properties of RSA keys need not be actively avoided, is further supported by large scale practical attacks on RSA keys, where none of the properties were used in factoring large amounts of keys. Lenstra et al. [5] performed an analysis of large sets of RSA keys, refuting the assumption that different random choices are made each time keys are generated in practice. The devastating effect of malfunctioning random number generators was demonstrated by Heninger et al. [6], where they show that surprisingly large amount of keys retrieved by internet-wide scanning share factors, making them easily factorable. Even more evidence of bad generators being responsible for breakable keys was given by Bernstein et al. [7].

1.2 The RSA cryptosystem

The RSA public-key cryptosystem has earned wide acceptance since its initial publication by Rivest, Shamir and Adleman in 1977 [8]. It is used to ensure confidentiality (encryption) or authenticity (digital signatures). Its security is based on the computational infeasibility of integer factorization. Years of being subjected to intense scrutiny by cryptographers have led to many practical attacks [9] and their mitigations. Currently, the best general attack against RSA keys is factorization of the modulus, even though the intractability of factorization and its equivalence to breaking RSA remain open problems.

1.2.1 The RSA key pair

Two key types are needed: an RSA public key and an RSA private key. Together, they form an RSA key pair.

The RSA public key consists of the RSA modulus n and the RSA public exponent e. The private key in the simplest form consists of the RSA modulus n and the RSA private exponent d. All of the parameters are positive integers. The key generation process involves several steps:

1. Generate two large, distinct primes p and q, having approximately same bit length.
2. Compute the modulus $n = p \cdot q$ and the Euler's totient function $\phi(n) = (p-1) \cdot (q-1)$.
3. (Randomly) choose an odd public exponent $2 < e < \phi(n)$, coprime to $\phi(n)$.
4. Compute the private exponent $d = e^{-1} \bmod \phi(n)$ using the extended Euclidean algorithm.

The encryption exponent e is typically chosen as a fixed value with low Hamming weight (e.g., 3, 17, 65537) to speed up encryption. In such case, new primes are generated until they are coprime to the exponent.

The *universal private exponent* can be computed as $d = e^{-1} \bmod \lambda(p-1, q-1)$, where λ is the least common multiple of $p-1$ and $q-1$. Since $\lambda(n)$ is a proper divisor of $\phi(n)$, the private exponent will be smaller. In fact, any exponent $d + k \cdot \lambda(n)$, $k \in \{0, 1, \cdots\}$ will work for RSA decryption.

An alternative representation of the private key is a quintuple $(p, q, dP, dQ, qInv)$. It is used when the Chinese remainder theorem (CRT) is applied to RSA. The benefits of CRT for faster decryption were discovered by [10]. The parameters p and q are the factors of the modulus, dP and dQ are the CRT exponents of the first and the second factor, respectively. The CRT coefficient is denoted $qInv$. Conversion between the two representations is simple. No additional requirements are placed when generating CRT keys, hence any key can be used with or without CRT.

Some definitions [11, 12] allow for more factors of the modulus, however the multi-prime variant of RSA is rarely used in practice.

1.2.2 The RSA cryptographic primitives

The RSA scheme is built on the following cryptographic primitives (mathematical operations):

- The *RSA encryption* takes a message m represented as a positive integer between 0 and $n - 1$. The ciphertext c is $c = m^e \bmod n$.
- The *RSA decryption* converts the ciphertext to the message $m = c^d \bmod n$, assuming that the RSA private key is valid.
- The *RSA signature* takes a message m (typically a message digest) represented as a positive integer between 0 and $n - 1$. The signature s is computed as $s = m^d \bmod n$.
- The *RSA signature verification* converts the signature to the message (message digest) $m = s^e \bmod n$, assuming that the RSA public key is valid.

In practice, the CRT representation of the private key is used to benefit from faster decryption. The primitives, as described, suffer from several attacks (e.g., the Hastad's broadcast attack and the Franklin-Reiter related message attack [9]), therefore padding schemes are necessary for real implementations [11, 12].

2 The RSA integers

An RSA integer is an integer of the form $n = p \cdot q$, p and q primes, suitable as a modulus for the RSA cryptosystem.

For the purpose of counting RSA integers, the authors of [13] use the definition $n = p \cdot q$ with p and q primes such that $p < q < rp$ for fixed $r > 1$. To demonstrate the notions of RSA integers used by [1], we cite the *number theoretically inspired notion of RSA integers with tolerance r*, a subset of the positive quadrant $\mathbb{R}^2_{>1}$, where for every $n \in \mathbb{R}_{>1}$:

$$\mathcal{A}^{DM(r)} := \left\langle \left\{ (p,q) \in \mathbb{R}^2 \mid \frac{p}{r} < q < p \cdot r \wedge \frac{n}{r} < p \cdot q \leq n \right\} \right\rangle_{n \in \mathbb{R}_{>1}}.$$

The notion gives uniform distribution of the moduli (RSA integers), however no algorithm is provided to generate such keys.

Real implementations require concrete algorithms and are rarely concerned with the distribution of the moduli. In fact, for majority of the libraries, only two metrics are necessary to define RSA integers: the range of the possible moduli and the intervals from which the primes are chosen. It is typically required that the moduli generated by an algorithm always have the exact same bit length and the bit length of the primes is equal to half of the modulus length.

Guided by [1], we will approximate count of RSA integers based on the area in the (p,q)-plane bounded by these intervals, however we will diverge from their methodology there. Especially, we will not use logarithmic scale for illustrations, since it distorts the proportions when considering the distributions of individual primes (see figure 2.1).

2.1 Distribution of RSA primes

The number of primes in the interval $[1,n]$ is computed by the *prime counting function* $\pi(n)$. The prime number theorem approximates the function $\pi(n) \sim \frac{n}{ln(n)}$. Primes become rarer with increasing size.

The *Dirichlet's theorem* states that prime numbers are roughly uniformly distributed among the $\phi(n)$ congruence classes in the multiplicative group $\mathbb{Z}^*_n$, for any value of n [14].

Since the prime numbers are quite uniformly distributed, a good prime generator should also select them from uniform distribution. The prime number generator proposed in [15] is accompanied with the computation of its output entropy. Similarly, the authors of [1] evaluate the entropy of several generators, which is maximal, when the primes are chosen uniformly. As identified by these authors, bias is often introduced when choices for one of the prime factors (say, q) depend on the previous selection of the first prime factor (say, p). The result is naturally following the fact, that the conditional entropy $H(q|p)$ is less or equal to the individual entropy $H(q)$.

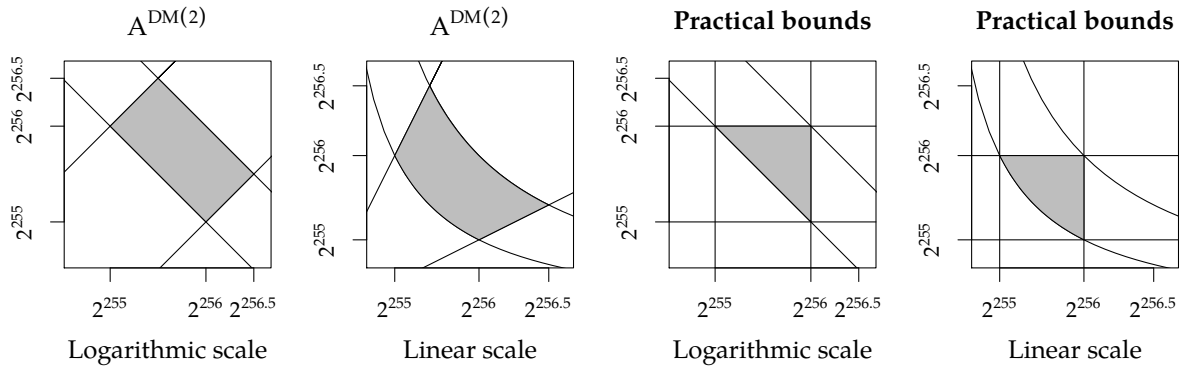

Figure 2.1: *Notions for RSA integers: illustration of the relation of primes p and q for 512-bit RSA keys. Left: notion for RSA integers inspired by number theoretical definition with tolerance* $r = 2$ *(the requirements are* $\frac{p}{r} < q < p \cdot r \wedge \frac{n}{r} < p \cdot q \leq n$*); right: the region bounded by precise length of modulus and precise length of primes (half of the modulus length). The region shapes are more intuitive in logarithmic scale, however the distribution of primes is better characterized using linear scale.*

Even when the prime number generator selects values from uniform distribution, the pairs of primes still may be produced with different probability. One of the design goals of a prime pair generator should then be to eliminate conditional probabilities.

Unfortunately, we are not aware of any work concerning the output entropy when generating primes with conditions (e.g., strong primes discussed in section 2.2.6).

Knuth and Trabb Pardo [16] approximated the distribution of k^{th} largest prime factor of a random number. Assuming the distribution holds for $p - 1$, where p is a random prime, the approximation will be useful to estimate portion of primes that are "weak" against special-purpose factorization attacks (see later in section 2.4).

2.2 Generating RSA primes

The choice of an algorithm for generating prime numbers depends on many conditions. One must find balance of time complexity, bias in the distribution, level of confidence in the primality, ease of implementation, memory requirements and other measures.

Practical methods for generating large primes involve selecting a candidate and determining whether it is composite or prime. For large integers, classical methods, such as factorization or trial division by primes up to the square root of the number, are impractical due to tremendous computational complexity. Instead, primality is decided using properties of primes known from number theory. We use the term *tests of compositeness* for probabilistic algorithms, and the corresponding primes are said to be *probable*.

Tests of primality which provide proof of primality are concerned with *provable* primes. True tests of primality exist for primes of special form or require some additional knowledge, such as a partial factorization of $p-1$. Since the RSA primes have general form and factorization is impractical, the tests are only used in specialized constructions (see section 2.2.5).

2.2.1 Probabilistic tests of compositeness

Methods described in this section are Monte Carlo algorithms with one-sided error. The decision *composite* is always correct and the algorithm may provide a *witness* (a proof) of compositeness. The decision *prime* is uncertain. Parameters that cause the algorithm to output *prime* for composite numbers are called *liars*. The uncertainty is decreased using amplification – repeated runs of the algorithms. For an efficient test, the witnesses must be abundant among the liars (i.e., more than half of all parameters are witnesses). Since the algorithms never output *composite* for prime numbers, they do not affect the distribution of a prime generator in any way.

Fermat test. A simple test is based on the *Fermat's little theorem*. It follows that for every prime n and an integer a, $1 \leq a < n$ the equality $a^{n-1} \equiv 1 \pmod{n}$ holds.

If n is composite and $a^{n-1} \not\equiv 1 \pmod{n}$ then a is a *Fermat witness* to compositeness of n.

If n is composite and $a^{n-1} \equiv 1 \pmod{n}$, then a is a *Fermat liar* to primality of n. The composite integer n is a *Fermat pseudoprime to the base a*.

Unfortunately, Fermat test may not be reliable even for large number of iterations. Some rare composite numbers are pseudoprime to all bases. *Carmichael numbers* are pseudoprime to all bases a, which satisfy $GCD(a,n) = 1$. If all prime factors of a Carmichael number n are large, then the probability that $GCD(a,n) \neq 1$ is small.

Fermat test may be combined with a stronger primality test, which will assert its result. The test will rule out most large composite numbers and it requires only one modular exponentiation. A fixed base, such as 2, may be used.

Solovay-Strassen test. The *Solovay-Strassen test* [17] is based on the Euler's criterion. It requires computation of the Jacobi symbol $\left(\frac{a}{n}\right)$, which is equal to the Legendre symbol for prime n.

Theorem 1 (Euler's criterion)**.** *Let $n \geq 3$ be a prime. Then for all integers a, $GCD(a,n) = 1$, the equivalence $a^{\frac{n-1}{2}} \equiv \left(\frac{a}{n}\right) \pmod{n}$ holds.*

If n is composite and either $GCD(a,n) \neq 1$ or $a^{\frac{n-1}{2}} \not\equiv \left(\frac{a}{n}\right) \pmod{n}$, then a is an *Euler witness* to compositeness of n.

If n is composite, $GCD(a,n) = 1$ and $a^{\frac{n-1}{2}} \equiv \left(\frac{a}{n}\right) \pmod{n}$, then a is an *Euler liar* to primality of n. The composite integer n is an *Euler pseudoprime to the base a*.

A composite number n has at most $\frac{\phi(n)}{2}$ Euler liars a in the interval $1 \leq a \leq n-1$. Every Euler liar is also a Fermat liar. A composite number is declared prime after i iterations of the test with probability lesser than $(\frac{1}{2})^i$ [14].

Miller-Rabin test. The *Miller-Rabin test* [18, 19] relies on the fact, that for each prime number n, $n = 2^s r + 1$, r is odd, at least one of the following equalities holds: $a^r \equiv 1 \pmod{n}$ or $a^{2^j r} \equiv -1 \pmod{n}$ for some j, $0 \leq j < s$.

If n is composite and $a^r \not\equiv 1 \pmod{n}$ and $a^{2^j r} \not\equiv -1 \pmod{n}$ for all j, $0 \leq j < s$, then a is a *strong witness* to compositeness of n.

If n is composite and $a^r \equiv 1 \pmod{n}$ or $a^{2^j r} \equiv -1 \pmod{n}$ for some j, $0 \leq j < s$, then a is a *strong liar* to primality of n. The composite integer n is a *strong pseudoprime to the base a*.

For composite n, at most $\frac{1}{4}$ of all bases a, $1 \leq a < n$ are strong liars for n. Every strong liar is also an Euler liar. Miller-Rabin test is more efficient than the Solovay-Strassen test and at least as correct, hence it is usually preferred. The result of the test can be efficiently amplified by several iterations with different bases a. The bases can be chosen randomly or the first few small primes are used [14].

In practice, the error probability of a single Miller-Rabin test becomes extremely small with increasing size of the number tested for primality. As a result, standards and other recommendations often reflect this and mandate only small number of iterations for large numbers.

Lucas test. The *Lucas probabilistic primality test* is based on Lucas sequences [20].

Let P, Q, D be integers such that $P > 0$ and $D = P^2 - 4Q \neq 0$. Let $U_0 = 0$, $U_1 = 1$, $V_0 = 2$, $V_1 = P$. We define the Lucas sequences U_k and V_k recursively for $k \geq 2$ as

$$U_k = PU_{k-1} - QU_{k-2}, \quad V_k = PV_{k-1} - QV_{k-2}.$$

The roots of $x^2 - Px + Q = 0$ are $\alpha = \frac{1}{2}(P + \sqrt{D})$ and $\beta = \frac{1}{2}(P - \sqrt{D})$. Then for $k \geq 0$ we have

$$U_k = \frac{\alpha^k - \beta^k}{\alpha - \beta}, \quad V_k = \alpha^k + \beta^k.$$

Let $\delta(n) = n - \left(\frac{D}{n}\right)$, where $\left(\frac{D}{n}\right)$ is the Jacobi symbol. If n is prime and $GCD(n, Q) = 1$, then

$$U_{\delta(n)} \equiv 0 \pmod{n}. \tag{2.1}$$

If n is composite, but the equality (2.1) holds, then n is a *Lucas pseudoprime with parameters P and Q*.

The test is used in the following algorithm:

1. Test if n is a perfect square (i.e., a square of a prime, a composite number).

2. Find the first D in the sequence $\{5, -7, 9, -11, 13, -15, 17, \dots\}$, for which the Jacobi symbol $\left(\frac{D}{n}\right) = -1$. If $\left(\frac{D}{n}\right) = 0$ for any D, the number is composite.

3. Set $P = 1$ and $Q = (1 - D)/4$.

4. Perform the Lucas test on n using parameters P, Q, D.

FIPS 186-4 recommends to use several iterations of the Miller-Rabin test followed by a single Lucas test, since there are no known non-prime values that pass the combination of the tests [21]. The Baillie–PSW primality test combines a Miller-Rabin test to base 2 with a Lucas test.

2.2.2 Selection of a candidate for primality test

A typical pseudorandom number generator (PRNG) generates a sequence of bits uniformly. In this section we discuss two methods for generating primes from intervals bounded by powers of two. As a result, the lowest and the highest bit of a candidate number can be set to one, to make the candidate odd and to ensure its correct bit length. It is not necessary to generate a new random number until the corresponding bits are equal to one.

Random sampling. Random search for a prime is an unbiased method for generating a prime candidate. PRNG is queried for a new large random number until a prime is found. The distribution of prime numbers is approximated by the prime number theorem (section 2.1). Since a random k-bit number is prime with probability $\frac{1}{ln(2^k)}$, approximately $\frac{ln(2^k)}{2}$ random numbers will be tested.

Incremental search. As opposed to the previous method, incremental search requires only one query to the PRNG to produce a prime number. It generates a random (odd) starting point of required length (by fixing the upper bit to one) and increments the candidate until the smallest prime is found. The idea was introduced in [22], where the authors show that there is no significant loss of security (entropy), when using this method. Small bias comes from the fact, that the "gaps" between consecutive primes have different length, therefore a prime preceded by a large gap is more likely to be chosen.

The distance from a random point to a prime follows similar distribution as random sampling, therefore the same number of primality tests will be performed. Methods that decrease the number of necessary tests are discussed in section 2.2.4.

2.2.3 Candidates from uneven intervals

In case that the primes are not chosen from intervals bounded by powers of two, the output of the PRNG must be transformed into a different interval. The task is to generate a random integer from the interval $[r, s]$. These techniques can be used for all algorithms that test random candidates for primality. Especially, they can be used to select starting point for incremental search, since it is unlikely that incrementing a large candidate would exceed the interval before a prime is found.

Rejection sampling. A naïve method generates integers from an interval bounded by powers of two, which encloses the required region. New sample is generated, until it lies in the correct region. The minimal number of random bits is used for every sample when generating from the interval $[0, 2^l]$, where l is the smallest integer such that $2^l \geq r - s$. The lower bound r is then added to the random number. This method will generate an unbiased distribution. Rejection sampling should be performed before the candidate is tested for primality.

Efficient method. The specification for RSA-OAEP [11] gives a method for generating a candidate c from the interval $[r, s]$ almost uniformly. Let M be the bit length of s and let m be a suitably chosen number (e.g., $m = 32$). Then the algorithm will generate random numbers, where the most probable integers (at the beginning of the interval) are chosen with probability at most $(1 + 2^{-m})$-times the probability of any of the least probable integers (from the end of the interval). The algorithm generates a random $(M + m)$-bit integer x uniformly from the interval $[0, 2^{M+m} - 1]$ and outputs

$$c = (x \bmod (s - r + 1)) + r.$$

Only one call to the PRNG is needed and the bias can be made arbitrary small by increasing the parameter m.

2.2.4 Efficiency improvements

Tests that determine primality of random integers perform complex operations (e.g., modular exponentiation). The probability that a large composite number is divisible by some small prime is high and such division can be computed efficiently. Random candidates are therefore subjected to some simple tests that rule out many composite numbers, before more expensive tests are used. The optimal number of small primes used in these improvements will depend on the specific implementation and the size of the large prime in question. The improvements do not affect the distribution of primes.

Trial division. Division of a candidate by a sequence of small primes is called trial division. The compositeness of the candidate is discovered as soon as any small integer

divides it. More expensive tests are performed only when the candidate is coprime to all predefined small primes.

Greatest common divisor. Small factors can be also detected by computing the greatest common divisor of the candidate and a product of small primes, using the Euclidean algorithm. This and the previous method can be used in combination with any method of selecting candidates.

Sieving. The sieve of Eratosthenes works especially well with incremental search. The algorithm computes remainders modulo a list of primes for the starting point and then marks the corresponding increments as composite (e.g., if $c \equiv 3 \pmod 5$, then $c + (5 - 3) + k \cdot 5$ is composite for $k \in \mathbb{N}$). The candidate for primality is chosen as the smallest number not yet marked as composite, until a prime is found.

Table based method. An algorithm that performs a simple transformation on candidates that fail primality test (e.g., incremental search with random increments) can benefit from a table based method. The algorithm computes a table of remainders modulo a predefined list of small primes. Every time the candidate is transformed, it is sufficient to recompute the table using operations with small integers (e.g., if the candidate was incremented by 2, all the remainders are also increased by 2, modulo the corresponding prime). The candidate is subjected to the complex primality test once all the remainders are non-zero.

Algebraic method. A candidate can be constructed to be coprime to a product of small primes what eliminates the need for trial division. The idea was proposed by [23], where they discuss the application for RSA primes (and primes with conditions). The algorithms were not used in any cryptographic library, therefore we omit the description.

2.2.5 Constructing provable primes

In this class of algorithms, the primes are constructed in such way, that their primality can be reasoned mathematically. An algorithm with almost uniform distribution was proposed by Maurer [24]. The method is based on Pocklington's theorem.

Theorem 2 (Pocklington's theorem)**.** *(Citation from [14]). Let $n \geq 3$ and let an integer F divide $n - 1$. The prime factorization of F is $F = \prod_{j=1}^{t} q_j^{e_j}$. If there exist an integer a satisfying:*

- $a^{n-1} \equiv 1 \bmod n$*; and*
- $GCD(a^{(n-1)/q_j} - 1, n) = 1$ *for each j, $i \leq j \leq t$,*

then every prime divisor p of n is congruent to 1 modulo F. If $F > \sqrt{n} - 1$, then n is prime.

Maurer's algorithm recursively constructs a provable prime p from a prime q. It selects a random R, $R < q$, until $n = 2Rq + 1$ can be proven prime using Pocklington's theorem. For prime n, a random base a satisfies the Pocklington's criterion with high probability.

A stronger theorem with an additional condition[1] only requires $F \geq \sqrt[3]{n}$.

2.2.6 Strong primes

The recommendation to use "strong" primes appears as early as in the first publication of the RSA cryptosystem [8]. The requirement to use strong primes is usually supported by citing special-purpose factoring and cycling attacks, as we will discuss in section 2.4. Rivest (one of the authors of the original paper) and Silverman later argued [3], that the use of strong primes is unnecessary and random primes are not "weak" in comparison. They also give a history of the recommendations appearing in literature. Despite many arguments against, strong primes are required for 1024-bit keys by FIPS 186-4 and are used by several cryptographic libraries that follow the standard.

Since the definition of strong primes differs in some sources, we adopt the terminology of [3], which covers all the cases. A prime p is considered to be a *strong prime*, if is satisfies the following conditions:

1. p is a large prime (e.g., $|p| \geq 512$ bits).
2. The largest prime factor of $p - 1$, denoted p^-, is large (e.g., $|p^-| \geq 100$ bits).
3. The largest prime factor of $p^- - 1$, denoted p^{--}, is large (e.g., $|p^{--}| \geq 100$ bits).
4. The largest prime factor of $p + 1$, denoted p^+, is large (e.g., $|p^+| \geq 100$ bits).

If a prime p is $p = 2 \cdot p^- + 1$, such that p^- is also prime, then p is a *safe prime* and p^- is a *Sophie Germain prime*. We say that a prime is:

- p^-*-strong* if p^- is a large prime.
- p^{--}*-strong* if p^{--} is a large prime.
- p^+*-strong* if p^+ is a large prime.
- (p^-, p^+)*-strong* if it is both p^-*-strong* and p^+*-strong*.
- (p^-, p^{--}, p^+)*-strong*, or just *strong*, if it is p^-*-strong*, p^{--}*-strong* and p^+*-strong*.

Additionally, one may use a safe prime p, then the prime p is p^-*-superstrong*. If p^- itself is a strong prime, then p is p^{--}*-superstrong*, etc. The requirement $p + 1 = 2 \cdot p^+$ means that p is p^+*-superstrong*. We did not encounter such primes in practice.

1. See Fact 4.59 and Note 4.63 in [14].

Generating RSA primes with conditions. Algorithms to generate (p^-, p^+)-*strong* primes are specified in the FIPS 186-4 standard. The principle of the algorithm in Appendix C.9 is as follows:

Given auxiliary primes p^- and p^+, construct a prime p of length $|p|$, such that $p-1 = a \cdot p^-$ and $p+1 = b \cdot p^+$ for some a and b. The algorithm generates primes almost uniformly from arbitrary region.

1. Use the CRT, so that $R \equiv 1 \pmod{2p^-}$ and $R \equiv -1 \pmod{p^+}$:
$$R = (((p^+)^{-1} \bmod 2p^-) \cdot p^+) - (((2p^-)^{-1} \bmod p^+) \cdot 2p^-).$$
2. Generate a random number X from the interval required for p.
3. Find the first odd integer $Y \geq X$, such that p^- is a prime factor of $Y-1$ and p^+ is a prime factor of $Y+1$:
$$Y = X + ((R - X) \bmod (2p^- \cdot p^+)).$$
4. If $|Y| > |p|$, go to step 2.
5. Increment Y by $2p^- \cdot p^+$, until Y is a probable prime.
6. Return $p = Y$.

A small modification to the algorithm will make it possible to generate p which is also p^{--}-*strong*. One does not select a random p^-, instead it is constructed to be congruent to one modulo p^{--}.

To generate (p^-, p^{--}, p^+)-*strong* primes, [14] recommends the Gordon's algorithm [25]. If p^{--} is set to 1, the algorithm generates (p^-, p^+)-*strong* primes.

1. Generate two large primes p^+ ($|p^+| \approx \frac{|p|}{2}$) and p^{--} ($|p^{--}| < |p^+|$).
2. Select an integer i_0 and find the smallest $i \in \{i_0, i_0+1, i_0+2, \dots\}$, such that $p^- = 2i \cdot p^{--} + 1$ is a prime.
3. Compute $p_0 = 2((p^+)^{p^- - 2} \bmod p^-) \cdot p^+ - 1$.
4. Select an integer j_0 and find the smallest $j \in \{j_0, j_0+1, j_0+2, \dots\}$, such that $p = p_0 + 2j \cdot p^- \cdot p^+$ is a prime.
5. Return p.

As in the case of the FIPS algorithm, incremental search is used to find the prime values satisfying the congruence conditions. However, the initial value is chosen by multiplying by random numbers, rather than adding them. Hence the bit length of p depends on the careful choice of the size of p^+, p^{--} and i_0, j_0. As a result, the primes will not be uniformly distributed and it is much more difficult to obtain $n = p \cdot q$ of correct length.

Size of p (in bits)	Size of B (in bits)	Probability, that random $p-1$ is B-smooth	
		u^{-u}	Knuth
256	100	$2.56^{-2.56} \approx 9.01 \cdot 10^{-2}$	$1.30 \cdot 10^{-1} \approx 2^{-3}$
512	100	$5.12^{-5.12} \approx 2.34 \cdot 10^{-4}$	$3.55 \cdot 10^{-4} < 2^{-11}$
1024	140	$7.314^{-7.314} \approx 4.78 \cdot 10^{-7}$	$< 8.75 \cdot 10^{-7} < 2^{-20}$
1536	170	$9.035^{-9.035} \approx 2.31 \cdot 10^{-9}$	$< 1.02 \cdot 10^{-9} < 2^{-29}$

Table 2.1: *Estimate of the probability, that a random prime does not pass the requirement, that $p-1$ has a large factor. The size of factors is taken from FIPS 186-4 (except for 256-bit primes, since the standard does not allow 512-bit keys). Two approximations for B-smoothness are used, u^{-u}, where all prime factors of n are $\leq n^{\frac{1}{u}}$ and a more precise approximation given by Knuth and Trabb Pardo in [16]. Since an RSA modulus requires two primes and $p+1$ must also have large factors, random keys will be FIPS-compliant with quarter the probability.*

Probability, that p is a strong prime. Dickman function $\rho(u)$ gives the probability, that for a real $u \geq 1$ all prime factors of n are $\leq n^{\frac{1}{u}}$. An approximation is $\rho(u) \approx u^{-u}$.

An integer is called *B-smooth* or *smooth with respect to a bound B*, if all its prime factors are $\leq B$.

The minimum length of large factor can be chosen according to FIPS 186-4 [21, Table B.1]. Table 2.1 gives the approximate probabilities, that for a random prime p, $p-1$ will be B-smooth for bound given by FIPS 186-4. Similar probabilities apply for B-smoothness of $p+1$. The standard requires (p^-, p^+)-*strong* primes. Assuming the sizes of largest factors of $p-1$ and $p+1$ are independent, the probabilities should be doubled. It should be noted, that the requirements are strong and much smaller smoothness bounds would suffice to protect against special-purpose factorization in practice. We discuss the issue in more detail in section 2.4.

FIPS 186-4 does not mandate p^{--}-*strong* primes. The probability that this additional condition is broken can be similarly estimated, since p^- was chosen at random.

2.3 Generating pairs of primes

The techniques for generating RSA primes can be used with several methods for selecting pairs of primes. In most cases the primes will have bit length equal to half of the modulus bit length. Additionally, prime pairs will be selected from such intervals, that the modulus always has precise bit length, equal to the intended size of the key k.

2.3.1 Rejection sampling

It is possible to use the von Neumann's rejection sampling method for generating keys from arbitrary regions. The region for prime pairs is enclosed in a larger region, from

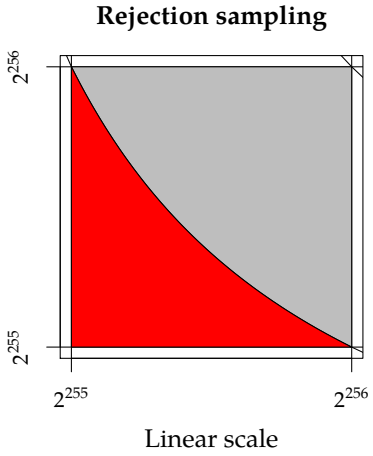

Figure 2.2: *Generating a pair of primes with rejection sampling. An irregular region is enclosed in a regular region, from which pairs of integers are generated efficiently. Values are rejected until a pair from the correct region is chosen.*

which values can be generated efficiently with uniform distribution. Samples that are not in the desired region are rejected.

A common example of this method generates primes of length equal to half of the modulus length and rejects the pairs that produce moduli of incorrect length (short by one bit). The primes are selected from the interval $\left[2^{\frac{k}{2}-1}, 2^{\frac{k}{2}} - 1\right]$, where k is the bit length of the modulus (assuming even k). See figure 2.2 for illustration.

An unbiased distribution is achieved by disposing of both primes in the unfavourable case. Some bias is introduced when the larger prime is kept and significant bias exists if the first (or second) generated prime is kept for the next attempt.

For efficiency, the rejection sampling should be performed before primality testing.

2.3.2 Practical intervals

Calls to PRNG can be expensive, especially when a cryptographically secure PRNG is used. A popular technique for prime pair generation minimizes the number of calls by ensuring that each prime is generated with exactly one call to the PRNG.

To avoid high usage of random bits, the interval from which primes are chosen can be reduced to $\left[3 \cdot 2^{\frac{k}{2}-2}, 2^{\frac{k}{2}} - 1\right]$. The lower bound can be achieved by fixing the two most significant bits of random number to binary one, the multiple achieves the correct length. Alternatively, the primes may be chosen from the interval $\left[\sqrt{2} \cdot 2^{\frac{k}{2}-1}, 2^{\frac{k}{2}} - 1\right]$ (refer to section 2.2.3 on uneven intervals). Since the primes are chosen independently, there is no additional bias introduced to their distribution.

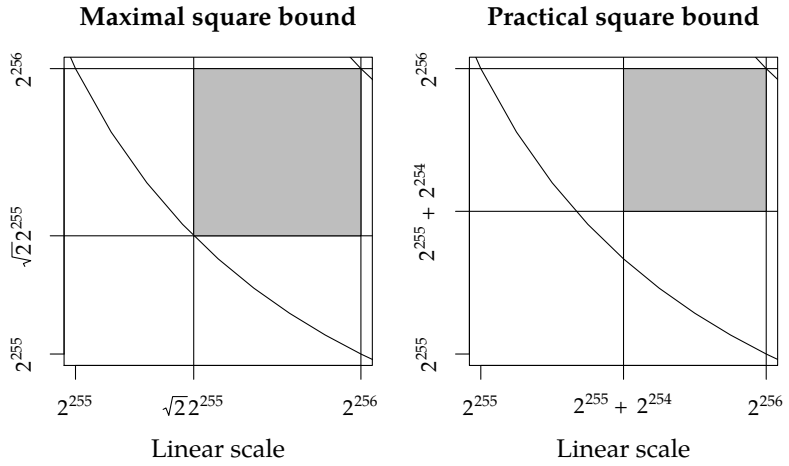

Figure 2.3: *Practical intervals for prime pair generation. The primes are selected from intervals* $\left[\sqrt{2} \cdot 2^{\frac{k}{2}-1}, 2^{\frac{k}{2}} - 1\right]$ *on the left and* $\left[3 \cdot 2^{\frac{k}{2}-2}, 2^{\frac{k}{2}} - 1\right]$ *on the right.*

The practical regions cover no less than one half and one quarter of the maximal region, therefore the entropy loss will be only approximately 1 or 2 bits, respectively.

2.3.3 Inverse transform sampling

Excess calls to the PRNG can be avoided even when the prime pairs are generated from an almost arbitrary region. It is necessary to compute the marginal distributions of a selected notion, which are the probability density functions of the individual primes. One of the primes is then selected from its distribution using inverse transform sampling. The second prime is selected uniformly from an interval influenced by the value of the first prime.

Inverse transform sampling generates random samples from any probability distribution given its cumulative distribution function. More details are provided by [1].

The cumulative distribution function is difficult to compute since it requires counting RSA integers from arbitrary regions. Thanks to [1], the distribution can be approximated as the ratio of two areas, which is much easier to compute. Specifically, for any notion of RSA integers $\mathcal{A}_n$ (a subset of the positive quadrant $\mathbb{R}^2_{>1}$) and prime p:

$$F_{\mathcal{A}_n} : \begin{array}{ccc} \mathbb{R} & \longrightarrow & [0,1] \\ p & \longmapsto & \frac{area(\mathcal{A}_n \cap ([1,p] \times \mathbb{R}))}{area(\mathcal{A}_n)} \end{array}$$

Figure 2.4 illustrates the cumulative distribution function of the notion $\mathcal{A}^{DM(r)}$ introduced in chapter 2. This notion is characterized by uniform distribution of moduli. The difficulty of the process outweighs its benefits, therefore the method is not commonly used in practice.

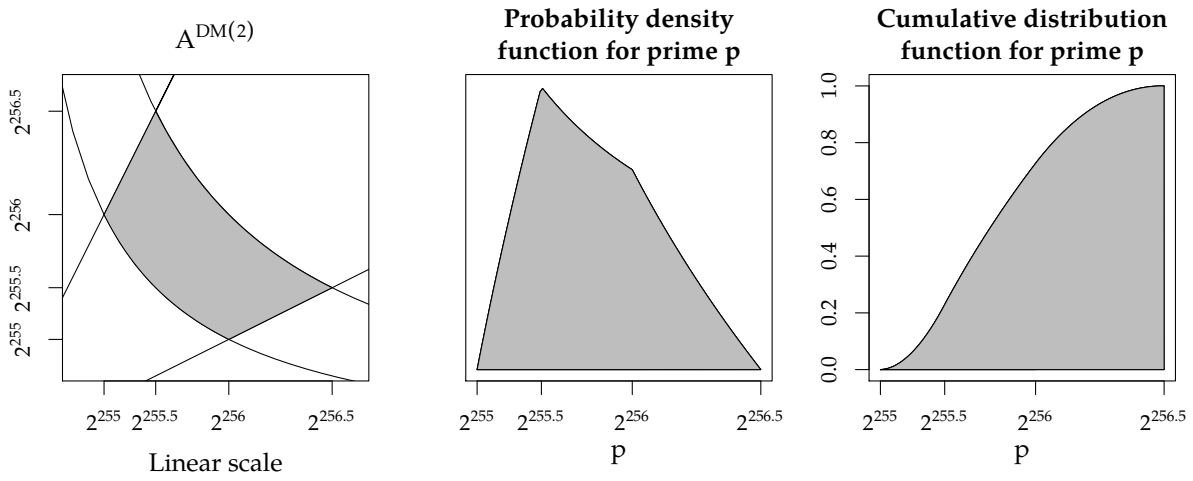

Figure 2.4: *Arbitrary region of prime pairs. Cumulative distribution function of one of the primes is computed. The prime p is generated from this distribution (e.g., by using inverse transform sampling). The second prime q is generated from uniform distribution, the interval depends on the specific p.*

2.4 Attacks on RSA keys

Rivest and Silverman introduced the notion of a *weak key* into the context of the RSA algorithm [3]. *General-purpose attacks* do not rely on the choice of the key and are always successful. Hence the purpose of this section is to show different types of weak keys abused by *special-purpose attacks*, that use a specific property of the key to break it. These attacks are successful only under certain conditions.

Specifically, we will consider factorization of RSA moduli with weak primes, methods that use improperly selected parameters of the keys and methods that were successfully used in large scale attacks. Some factorization methods are not suitable for RSA integers (such as the Pollard's rho algorithm), since they are not able to handle integers with two factors of similar length.

An attacker may first try to mount a special-purpose attacks, hoping that the key is weak. If not successful in a predetermined amount of time, the attacker moves to a more complex general-purpose attack.

2.4.1 Pollard's $p-1$ factorization algorithm

Pollard's $p-1$ factorization algorithm [26] can efficiently find a factor of a composite number n, if for some prime factor p of n, $p-1$ is B-smooth for a reasonably small B.

For any multiple m of $p-1$ and some a, $GCD(a,p) = 1$, Fermat's little theorem implies $a^m \equiv 1 \pmod{p}$. Therefore if $f = GCD(a^m - 1, n)$, then p divides f.

One first guesses the smoothness bound B, then computes the product of all prime powers less than B as $m = \prod_{primes\ q \leq B} q^{log_q n}$. The algorithm returns a factor of n if $p-1$ is indeed B-smooth, otherwise it reports failure.

1. Choose (a random) a, $1 < a < n$.
2. Compute $f = GCD(a, n)$. If $f > 1$, return f.
3. Compute $x = a^m \pmod{n}$.
4. Compute $f = GCD(x-1, n)$. If $f > 1$, return f.
5. Could not find a factor of n.

Mitigation. The attack requires B-smooth $p-1$ or $q-1$. If they are both generated to have a large prime factor, the bound B will not be reachable by an attacker. Algorithms that construct primes with such properties are mentioned in section 2.2.6.

Probability of success. The probability of success depends on the definition of large factor. The algorithm requires $O\left(\frac{Bln(n)}{lnB}\right)$ modular multiplications [14]. Then it is sufficient to estimate what bound B is safe from a motivated attacker. For example, FIPS 186-4 recommends factors of length > 140 bits, for 2048-bit keys. We estimated the probability that a random 1024-bit number fails this requirement to be $< 2^{-20}$. See table 2.1 and section 2.2.6 for justification. It should be noted that 2^{140} is far from reasonably small bound B, hence the probability of successfully mounting the attack is much smaller in practice.

2.4.2 Williams' $p+1$ factorization algorithm

Williams' $p+1$ factorization algorithm [27] is a special-purpose method similar to the Pollard's $p-1$ method. It factors n if $p+1$ is B-smooth for some prime factor p of n. The computation is based on Lucas sequences.

Mitigation and feasibility. Both $p+1$ and $q+1$ must have at least one large (distinct) factor. The attack succeeds on random keys with probability similar to the $p-1$ attack.

2.4.3 Fermat's factorization method

The Fermat's factorization algorithm belongs to a family of square factoring methods. In general, if $a^2 \equiv b^2 \pmod{n}$ and $a \not\equiv \pm b \pmod{n}$, then n divides the difference of squares $a^2 - b^2 = (a+b) \cdot (a-b)$. Since n does not divide neither $(a+b)$ nor $(a-b)$, $GCD(a+b, n)$ and $GCD(a-b, n)$ are factors of n.

Fermat's factorization method tries to find a such that $a^2 - n = b^2$ for some integer b. Then n is factored as $n = (a + b) \cdot (a - b)$.

The algorithm in pseudocode follows:

1. $a = \lceil \sqrt{n} \rceil$.
2. Compute $x = a^2 - n$.
3. While x is not a square, repeat:
   3.1 $x = x + 2a + 1$.
   3.2 $a = a + 1$.
4. Return factors $(a + \sqrt{x})$ and $(a - \sqrt{x})$.

Mitigation. The algorithm is usually published with the initial guess $a = \lceil \sqrt{n} \rceil$, which works for every composite n. However, large primes p and q would have to be very close to $\lceil \sqrt{n} \rceil$ in order to find them in reasonable time. If the extremely unlikely event occurs, new key (or a replacement prime) should be generated. It is possible to search in an interval around any a using this method. Hence the primes should be selected from large enough intervals.

Probability of success. To prevent this attack, p and q should not be too close together. FIPS 186-4 and ANSI X9.44 require the difference $|p - q|$ to be larger than $2^{\frac{k}{2}-100}$, where k is the size of the key. To fail this requirement, two primes of same length would have to be identical in all of their first 100 bits. Assuming uniform distribution of prime generator, the probability is approximately 2^{-100}, neglecting a few (1 or 2) upper bits, that may be fixed.

2.4.4 Lehman's improvement to Fermat's method

Let the ration of p/q be near a ratio of two small numbers r/s. Then an attacker computes $nrs = ps \cdot qr$, ps and qr are close to $\sqrt{nrs}$ and Fermat's method will efficiently find them. Then $GCD(ps, n) = p$ and $GCD(qr, n) = q$.

Lehman discovered a systematic way of choosing the rational numbers [28].

Mitigation and feasibility. After computing p and q, one might check if the ratio p/q is near a ratio of two small integers. In such case a new prime or a new key is generated. This requirement appeared in ANSI X9.31-1997 but the probability of small ratio is negligible [2]. The check is not performed in any cryptographic library.

2.4.5 General-purpose factorization methods

Before general-purpose factorization methods were known, it was often recommended to actively avoid properties of the primes that would make any of the previous factorization attacks feasible. As we have already shown, the special-purpose factoring attacks succeed with negligible probability for practically large RSA keys. General-purpose attacks do not rely on any conditions.

Elliptic curve factorization. Lenstra's elliptic curve factorization method [29] (ECM) generalizes Pollard's $p-1$ method. ECM replaces $\mathbb{Z}_p^*$ (which has order $p-1$) with a random elliptic curve group over $\mathbb{Z}_p$. The algorithm will factor n with high probability, if the order of the randomly selected group is smooth with respect to some small bound. Otherwise the method may be repeated with a different random group. The group orders are uniformly distributed in $\left[p+1-2\sqrt{p}, p+1+2\sqrt{p}\right]$ [14]. The method is not as efficient as the quadratic sieve.

Quadratic sieve factorization. The quadratic sieve factorization invented by Pomerance [30] searches for a congruence of squares modulo n, but does so systematically, as opposed to random square factoring methods.

While Fermat's method searches for an a such that $x = a^2 - n$ is a square ($x = b^2$), quadratic sieve computes $a_i^2 \bmod n$ for several a_i and then searches for a subset $\{a_1, \ldots, a_j\}$ of a_i whose product $(a_1^2 \bmod n \cdot \cdots \cdot a_j^2 \bmod n)$ is a square. If found, it yields a congruence of squares and thus a factorization of n:

$$(a_1^2 \bmod n \cdot \cdots \cdot a_j^2 \bmod n) \equiv ((a_1 \cdot \cdots \cdot a_j) \bmod n)^2 \pmod n$$

Number field sieve factorization. General number field sieve (GNFS or just NFS), originally proposed by Pollard [31], is the most efficient general factorization method for large numbers. As the quadratic sieve method, the algorithm searches for a congruence of squares modulo n. The algorithm is considerably more complicated than QS, but it achieves large speed-up in factorization numbers larger than approximately 350 bits. The current record of factoring 768-bit RSA modulus was achieved with the NFS [4].

2.4.6 Attacks on small private exponent

Wiener [32] devised an attack based on continued fractions which makes it possible to recover the private exponent d from public information, if d is small ($d < n^{\frac{1}{4}}$).

Boneh and Durfee [33] later extended the result to $d < n^{0.292}$ using lattice basis reduction and estimated the correct bound as $d < n^{\frac{1}{2}}$.

Mitigation. The private exponent is small ($d < n^{\frac{1}{2}}$) with very low probability [21]. In such case a new key should be generated.

2.4.7 Cycling attacks

RSA encryption is a permutation on the message space $\{0, 1, \ldots, n-1\}$, hence there always exists some k, such that $c^{e^k} \equiv c \pmod{n}$. Then the message can be recovered by repeated encryption, until the value k is found. The message is computed as $m \equiv c^{e^{k-1}} \pmod{n}$. A generalized version of the cycling attack either factors n or (much less frequently) succeeds as the basic cycling attack [14].

Mitigation and feasibility. Some authors use the cycling attacks to justify the requirement for (p^-, p^{--}, p^+)-*strong* primes [14, 34]. Since this attack typically gives a factorization of the modulus or the private key (from which the prime factors can be determined), it is assumed that it performs no better than other factorization methods. A more precise analysis in [3] shows that the attack is extremely unlikely to succeed.

2.4.8 Small public exponents and Coppersmith's attack

Small exponents are desirable to speed up encryption. The exponent $e = 3$ was commonly used. If the message m is short and the ciphertext is shorter than the modulus ($c = m^3 < n$), the message can be recovered by integer cube root. If three recipients with different moduli and common public exponent $e = 3$ receive the same message, an eavesdropper can compute the message using the CRT (the Hastad's broadcast attack [9]). Protection against these and related attacks rely on salting the messages with different random strings.

Coppersmith [35] published an attack, where the knowledge of two thirds of a message encrypted with public exponent 3 enables an attacker to recover the message.

Mitigation. Even if a padding scheme is used, do not use short public exponents. A slightly larger exponent with low Hamming weight (e.g., $2^{16} + 1 = 65537$) still provides fast encryption and there are fewer known attacks against RSA with such exponents.

2.4.9 Attacks on keys generated with low entropy

Algorithms for prime number generation and PRNGs are deterministic. When given the same seed, they will generate the same prime. For correctly randomly seeded generator, it is extremely unlikely that two parties will obtain identical primes. However, when a PRNG is incorrectly seeded, the probability of shared factors may become much higher than anticipated (e.g., consider the shared keys generated due to a bug with low entropy seed in OpenSSL in Debian [36]). Other problems arise from malfunctioning PRNGs,

if they generate predictable output. Simple attacks that exploit insufficient entropy can factor many keys at once, however they are not suitable for targeted attacks, where the victim has a correctly generated key.

Batch GCD and shared prime factors. If two moduli share exactly one prime factor, it can be efficiently determined as their greatest common divisor. When presented with a large set of keys of which some share a prime, a naïve way to find shared factors would be to compute GCD of every pair. A batch GCD algorithm [37] can efficiently compute GCD of millions of RSA moduli. As results of independent researches [5, 6] show, thousands of keys collected on the internet can be factored using this method.

Coppersmith's partial key exposure attack. Coppersmith [38] showed how to find the factors of RSA modulus $n = p \cdot q$, if the $\frac{1}{4}log_2 n$ most significant bits of p are known. The method has practical applications, as shown in [7]. Smart cards used as national ID cards in Taiwan produced keys, where the upper bits of some primes were predictable due to errors in PRNG.

3 Standards for RSA

In this chapter we examine standards for the RSA cryptosystem. We identify requirements and algorithms concerned with generating RSA keys and note where the standards differ in their requirements. We will denote the length of the modulus as $nLen$.

All standards in this chapter compute the private exponent as $d = e^{-1} \bmod \lambda(n)$, where $\lambda(n)$ is the least common multiple of $p-1$ and $q-1$.

3.1 NIST FIPS 186-4

The FIPS 186-4 standard published by NIST [21] mandates that 512-bit primes are (p^-, p^+)-*strong*. The 1024-bit and 1536-bit primes may be either (p^-, p^+)-*strong*, probable or provable. The criteria for integer factorization cryptography (IFC) key pairs are listed in Appendix B.3.1 of the standard.

The odd public exponent $2^{16} < e < 2^{256}$ is fixed or randomly selected before generating the primes. Key generation is repeated in the rare case $d \leq 2^{nLen/2}$.

Primality tests

Appendix C.3 prescribes the required number of iterations of Miller-Rabin test with random bases. Alternatively, fewer Miller-Rabin tests can be performed, if they are then followed by one Lucas probabilistic primality test specified in Appendix C.3.3.

Prime generation

The primes are always selected from the interval $[\sqrt{2} \cdot 2^{nLen/2-1}, 2^{nLen/2} - 1]$.

Probable primes. Prime p is generated by random sampling. This method is only used as a part of Appendix B.3.3. A candidate c is generated by rejection sampling random odd numbers produced by a PRNG, until $c \geq \sqrt{2} \cdot 2^{nLen/2-1}$.

Strong probable primes. Given auxiliary primes p^- and p^+, the algorithm in Appendix C.9 constructs a probable prime p, such that p^- divides $p-1$ and p^+ divides $p+1$. We described the method in section 2.2.6.

Provable primes. Appendix C.10 describes the process for generating a provable prime p. If lengths of auxiliary primes are given, the algorithm will generate (p^-, p^+)-*strong* p for some provable primes p^- and p^+. Provable primes are constructed by Shawe-Taylor algorithm (Appendix C.6).

RSA key generation

Several methods to generate keys are specified. The length of both primes is always equal to $nLen/2$. The second prime is repeatedly generated in the unlikely case that $|p - q| \leq 2^{nLen/2-100}$.

Pair of provable primes. The Appendix B.3.2 gives an algorithm to generate random provable primes p and q. The primes are generated independently using the method for provable primes (Appendix C.10).

Pair of probable primes. The algorithm from Appendix B.3.3 independently generates a pair of probable primes.

Pair of strong primes. The standard allows (p^-, p^+)-*strong* primes. The large prime factors p^- and p^+ are called auxiliary primes.

Again, there is a choice of provable strong primes (Appendix B.3.4) and probable strong primes. The auxiliary primes for probable strong primes can be provable (Appendix B.3.5) or probable (Appendix B.3.6) primes.

3.2 ANSI X9.31 and X9.44

We do not have a copy of the ANSI X9.31-1998 or the X9.44-2007 standard, hence we will rely on various citations [1, 2, 21].

Primality tests. The primes must pass a probabilistic test with probability of error less than 2^{-100}. The recommended method is the Miller-Rabin test.

Prime generation. The standard requires (p^-, p^+)-*strong* primes. Primes will be in the interval $[\sqrt{2} \cdot 2^{nLen/2-1}, 2^{nLen/2} - 1]$. The auxiliary primes p^-, p^+ should be in the range 2^{100} to 2^{120} (101 to 120 bits). An algorithm is given in X9.31 Appendix B.4.

RSA key generation. The modulus length is $1024 + 256x$ bits for $x \in \{0, 1, \dots\}$. The primes must differ in some of their top 100 bits. According to [2], the version ANSI X9.31-1997 also requires that p/q is not near a ratio of two small integers, $GCD(p - 1, q - 1)$ is small and $p - q$ has a large prime factor. The last requirement seems to be verifiable only by factoring the difference $p - q$ and no algorithm is given for construction of such primes.

3.3 IEEE 1363-2000

The key generation algorithm from IEEE 1363-2000 standard [39] produces pairs of primes from an irregular region. The region is bounded by precise bit length of p and precise bit length of modulus n. However, the distribution of q is not uniform. Since one of the primes is distributed uniformly, it is not necessary to use inverse transform sampling (section 2.3.3).

The algorithm generates moduli in the interval $\left[2^{nLen-1}, 2^{nLen}-1\right]$ with almost uniform distribution. However, the second prime ranges from $\left[2^{nLen/2-1}, 2^{nLen/2+1}-1\right]$ (i.e., it can be one bit longer than usual). Hence these keys may not be suitable if primes of length equal to half of the modulus length are expected. Indeed, we did not encounter this algorithm in practice, even though it is easy to implement. Since q is generated from larger interval for small p than for large p, the distribution of q is very interesting. We included this algorithm in the further analysis of software libraries and the output (using random probable primes) is presented in figure A.4.

Primality tests. Candidates for primality are tested by the Miller-Rabin primality test (Annex A.15.1). Number of trials is prescribed in Annex A.15.2 of the standard.

Prime generation. The standard allows randomly generated primes (i.e., generated by random sampling – Annex A.15.6) or (p^-, p^{--}, p^+)-*strong* primes (with an algorithm in Annex A.15.8).

The algorithm for strong primes is similar to the Gordon's algorithm from section 2.2.6, with small differences. The standard gives precise instructions for the choice of the parameters, so that the primes have correct length. It is remarked, that random primes are almost certain to be strong enough in practice.

The value p_0, which satisfies $p_0 \equiv 1 \pmod{p^-}$ and $p_0 \equiv -1 \pmod{p^+}$, is constructed as $p_0 = (((p^+)^{-1} \bmod p^-) \cdot p^+ - ((p^-)^{-1} \bmod p^+) \cdot p^-) \bmod p^- \cdot p^+$.

The primes of the form $p^- = i \cdot p^{--} + 1$ and $p = j \cdot p^- \cdot p^+ + p_0$ are generated by random sampling the values i and j, until p^- and p are prime, rather than with incremental search starting with random values i and j.

The algorithm is more similar to the Gordon's algorithm than to the FIPS algorithm, because of the construction of primes with congruence conditions. Based on an analysis of similar algorithm in PGP (section 4.2), we expect that the distribution is not uniform (as opposed to the FIPS algorithm), even when i and j are chosen uniformly.

RSA key generation. An algorithm for generating RSA keys is given in Annex A.16.11. The first prime p is selected uniformly from the interval $\left[2^{nLen/2-1}, 2^{nLen/2}-1\right]$. The second prime q is selected uniformly from the interval $\left[\lfloor \frac{2^{nLen/2-1}}{p} + 1 \rfloor, \lfloor \frac{2^{nLen/2}}{p} \rfloor\right]$.

3.4 RSAES-OAEP specification

The specification of RSAES-OAEP encryption scheme gives an efficient algorithm without additional conditions. The possibility of multi-prime RSA is discussed, where modulus is the product of more than two primes.

Primality tests. Integers are tested for primality with the Miller-Rabin primality test, as a part of prime generation algorithm in Section 1.5.2 of the standard. The minimal number of iterations is also specified.

Prime generation. Primes are generated according to Section 1.5.2. Incremental search is used with trial division by the first 2000 primes. If intervals are not bounded by powers of two, the standard is using the efficient method we discussed in section 2.2.3.

RSA key generation. The prime pairs are generated independently from the maximal square region, therefore the primes range in the interval $[\sqrt{2} \cdot 2^{nLen/2-1}, 2^{nLen/2} - 1]$.

3.5 Other standards

Some standards neither provide algorithms for key generation, nor place any requirements on RSA keys, other than the necessary ones. PKCS #1 and ISO/IEC 18033-2 are examples of such specifications.

PKCS #1. Key generation is not in the scope of the PKCS #1 specification [12]. There are no conditions placed on RSA primes and keys. Interestingly, the modulus may be a product of more than two primes.

ISO/IEC 18033-2. The ISO/IEC 18033-2 standard [40] prescribes the functional interfaces for RSA key generation, however no specific algorithm is given. The primes should have similar length.

4 Analysis of cryptographic libraries

We carried out a sanity check of RSA key generation methods used in cryptographic libraries. We found that all the implementations follow methods from standards or reputable publications. Hence the algorithms were approved for generating RSA keys based on cryptoanalysis performed by skilled cryptographers. However, there may be bugs in the implementations, especially when seeding random number generators, which we did not explore. We provide a detailed analysis for all the libraries in section 4.2.

4.1 Methodology and criteria

We performed several analyses of large number of keys of different sizes generated from different cryptographic libraries (usually two million keys per library). The criteria were selected based on the published standards and algorithms for prime selection and RSA key generation and our analysis of source code of cryptographic libraries. We automated the data processing to obtain statistical properties of the keys. We observed the relations between algorithms and their outputs and identified when the outputs provide meaningful information about the process used for key generation.

The main advantage of the black-box approach is the possibility to use it with proprietary implementations. We demonstrated the method by analysing keys obtained from the default cryptographic provider in Microsoft Windows. The methodology also assists with source code analysis, since it is difficult to visualize the distribution of primes and moduli just from the algorithm.

The main disadvantage comes from the fact, that some properties of keys simply cannot be observed or require large computational effort to discover.

Since the majority of libraries let the user select their public exponent and offer the same default (65537), we do not have enough data to perform meaningful analysis on public exponents.

4.1.1 The most significant byte of primes

The distribution of primes is interesting for two reasons – it is affected both by the prime generator (section 2.2) and by the method of selecting prime pairs (section 2.3).

Since the two primes are often selected differently, their distributions must be examined separately. This can be done by computing distributions of the two primes and visualizing them as two bar plots. Since such representation does not properly capture the relations between the prime values, we plot them in a two-dimensional image. The frequencies are represented by colors. Figure 4.1 contains an example graph which illustrates the relation between the most significant bytes of primes. The visualization was generated automatically and can be easily repeated for other sources. More examples are given in appendix A.

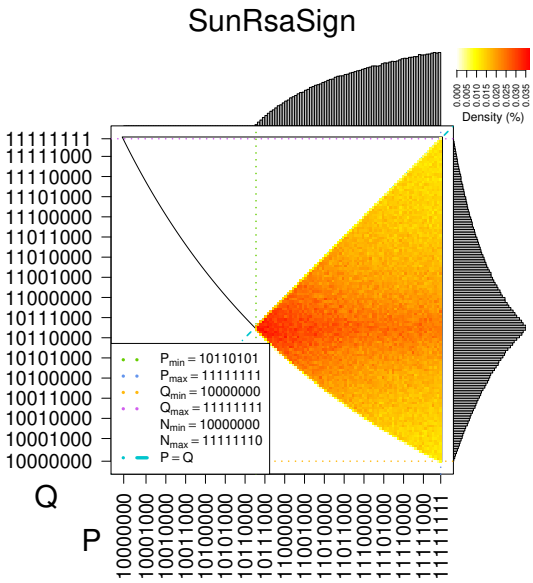

Figure 4.1: *An example distribution of the most significant byte (MSB) of the primes p and q. A million keys generated by the Java SunRsaSign Cryptography Provider are plotted. On the horizontal axis, the values of MSB of p are plotted. On top of the graph, the distribution is illustrated by a bar plot. The distribution of MSB of q is plotted vertically. The colors represent the likelihood that the MSBs of primes of a random key $n = p \cdot q$ have the respective values. The color ranges from white (not likely) over yellow (more likely) to red (very likely). The black curve bounds the maximal region, where the primes and the moduli have correct length.*

Although we examine the distribution of the most significant byte, in the case of most libraries, only the first few bits are biased. To determine which bits are already not biased, it is useful to compute frequencies for individual bits.

4.1.2 The most significant byte of modulus

The distribution of moduli depends on the distributions of the primes. The primes are sometimes ordered such that $p < q$ or $p > q$. This fact will affect the observed distribution of the primes, however it is not visible from the distribution of the moduli.

Whereas it may seem desirable to have uniform distribution of the moduli [1, 39], in practice no libraries attempt to obtain it. Either the authors of the libraries are not aware of this metric, or they consider that the difficulty of obtaining such distribution outweighs potential benefits. Nevertheless, the distribution is important, since it can be computed even only from public keys and it helps with determining the underlying algorithm. To see examples how the distribution of primes affects the modulus, refer to appendix B. The figures are also automatically generated from the collected keys.

4.1.3 Prime and modulus congruences

Recall from section 2.1, that according to the Dirichlet's theorem, prime numbers are roughly uniformly distributed modulo all primes. This fact can be used to test whether or not the prime generation algorithm manipulates with small factors of $p - 1$ and $q - 1$. The distribution is visible from moduli as well. Especially, when the primes are always $p \equiv q \equiv 2 \pmod 3$, it can be detected even from small number of public keys. In such case, the modulus is always $n \equiv 1 \pmod 3$ instead of $n \equiv 1 \pmod 3$ and $n \equiv 2 \pmod 3$ with similar probability.

Using this method, OpenSSL was singled out as the only library which manipulates with small prime factors of $p - 1$ and $q - 1$. We believe this is caused by a bug rather than done intentionally.

Similarly, it is possible to detect when Blum integers[1] are used as RSA moduli. However, in the case of the reviewed cryptographic libraries, only GNU Crypto is using moduli in such form (and not intentionally).

4.1.4 Time of key generation

We measured the time of the key generation process[2]. The length of the process differs, affected by the time it takes to find two primes. We expect that other parts of the process, such as computation of private key, take almost the same time in every run. We experimentally obtained the distribution for amount of random odd numbers it takes to find a prime by random search. The distribution corresponds also to the distance from random point to nearest prime, as illustrated by figure 4.2. Distribution of sum of distances for p and q creates a log-normal distribution, similar as we typically observe for the time of the process in a software library.

We compared the number of candidates that are tested for primality in case when sieving or trial division is used[3]. Some of the results are illustrated in figure 4.3. When no sieve is used, the number of primality tests corresponds to the distance from random number to a prime. We hypothesize, that comparing the time distribution obtained from a software library to a set of precomputed distributions may give some information about the size of the sieve used by a black-box implementation of incremental search. More precise results would require distributions also for rejection sampling, provable primes and strong primes.

---

1. Blum integers $n = p \cdot q$, p and q prime, satisfy the congruences $n \equiv 1 \pmod 4$, $p \equiv q \equiv 3 \pmod 4$. The primes p and q are Blum primes.
2. Usually measured in number of clock ticks. The time does not include the period when the system gathers entropy for `/dev/random` and other interrupts, which is the longest part of the process in some libraries, such as libgcrypt.
3. Out of 4 million 256-bit composite candidates, all were ruled out by one Miller-Rabin test. Hence we can use the number of candidates interchangeably with variable number of tests performed. The last candidate, a prime, will be tested with more than one test, but the amount is fixed (e.g., 15).

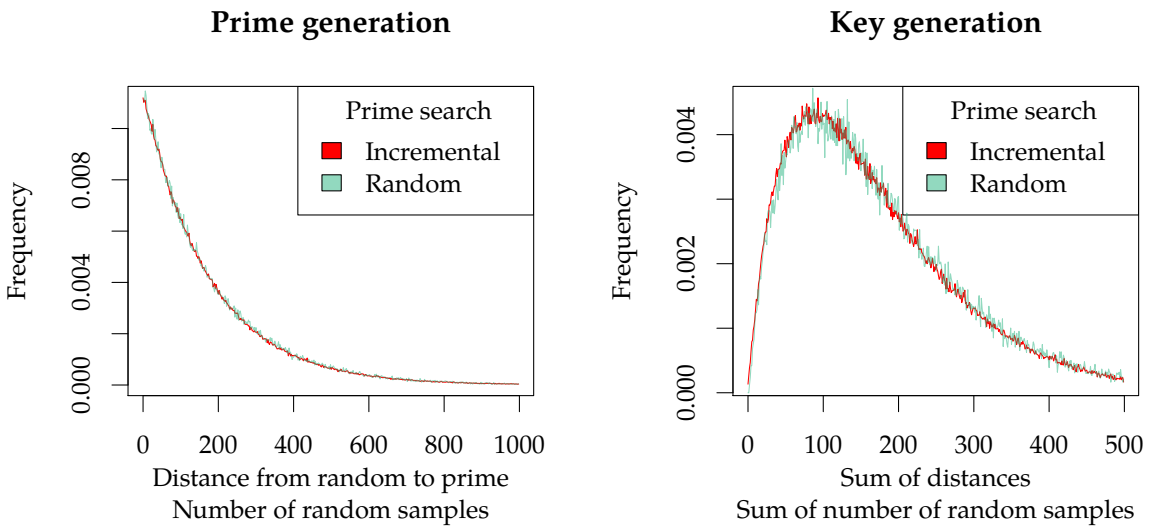

Figure 4.2: *Left: distribution of amount of random odd numbers generated, until a prime is found, exactly matches the distribution of distance from random number until a prime is found by incremental search (in 2-increments). Right: summing the number of trials for p and q creates a log-normal distribution.*

4.1.5 Primes with conditions

Our methodology lacks when it comes to the question of primes with conditions. Since some conditions, such as large bit length of the difference $|p - q|$, large bit length of the private exponent d and high Hamming weight of the modulus are extremely unlikely to be broken, we cannot detect from the distribution, whether or not they are being enforced. It is possible to detect the condition, when unreasonably high limits are required. For example, we can see the requirement that p and q differ somewhere in their top 6 bits as enforced by PGP. However, we cannot see if a library checks for the difference in the top 100 bits, since the opposite would happen with negligible probability.

Some limited results can be obtained from factoring the values $p \pm 1$ for prime factors p and q. Since it is not excessively difficult to factor 256-bit integers, it can be verified if strong primes are being used even without the source code. We demonstrate this fact in appendix C. Some implementation enforce 1024 bits as the minimal size of the modulus, which can be disabled in the source code. However, the situation is more complicated for hypothetical black-box implementation which would allow only 1024-bit keys or stronger, since factoring 512-bit integers is costly. Since the lengths of the factors of $p \pm 1$ (auxiliary primes) required by FIPS 186-4 increase with larger key sizes, it may not be precise to generalize about large keys based on small key sizes. In other aspects, the libraries seem to behave consistently for different key sizes.

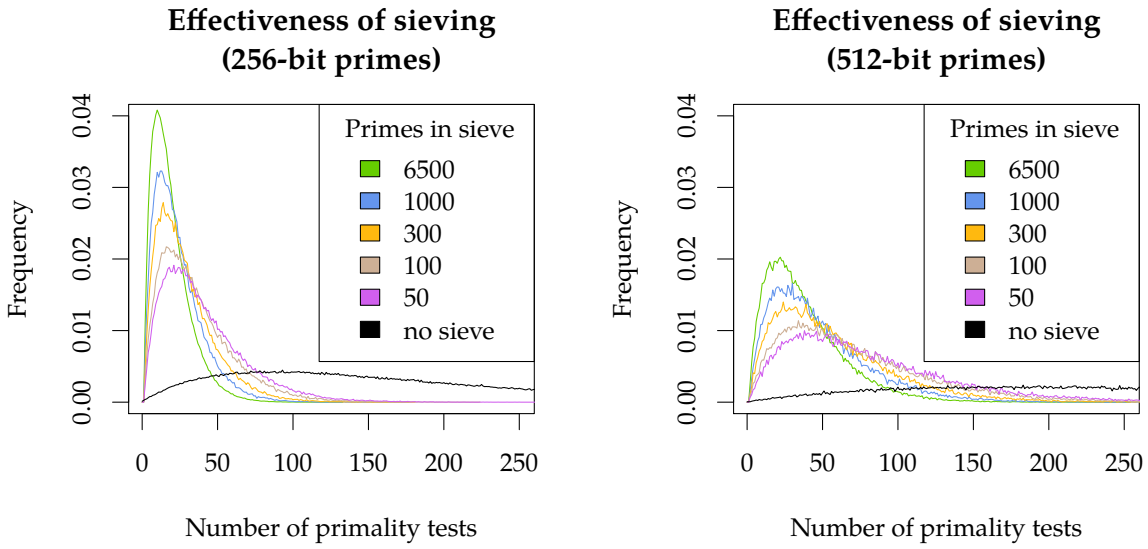

Figure 4.3: *Trial division (or sieving) by a different number of small primes. Testing a candidate with division by a few primes decreases the number of required probabilistic primality tests and speeds up prime generation. The expected number of primality tests decreases when sieving phase runs with more primes.*

It is also not possible to detect provable primes without (partial) factorization of the integers $p - 1$. Only one library uses provable primes and we did not explore further methods for detecting such conditions.

4.2 Detailed analysis

Almost all algorithms for generating RSA keys which we discussed appear in some cryptographic libraries. They are freely combined with different prime generation methods. Since some libraries use very simple methods, implementing almost text-book RSA key generation, we ordered the libraries rather arbitrarily, starting with the most interesting. To reduce effort with possible future analysis, we list the relevant methods and file names in appendix D.

4.2.1 OpenSSL 1.0.2g

OpenSSL [41] supports different cryptographic engines, we used the default one. We also examined the source code of LibreSSL 2.3.4 [42], which was forked from OpenSSL in 2014. We observed no differences in the prime and key generation methods.

Primes. Incremental search with trial division is used to generate primes. Primality is tested with Miller-Rabin test. Interestingly, trial division is performed on both p and $p/2$. Rejecting $p/2$ with small factors improves efficiency when searching for a safe prime (e.g., for Diffie-Hellman key exchange), however primality test is not performed on $p/2$. We hypothesize, that this a bug, since no other library or publication removes small factors of $p/2$. The relevant code has been a part of OpenSSL at least since version 0.9.6 (September 2000). This behaviour effectively fingerprints RSA keys generated by OpenSSL, as noticed by [43].

Keys. Two highest bits of primes are fixed to binary one, hence keys are generated from a square region.

4.2.2 OpenSSL FIPS 2.0.12

We used the OpenSSL FIPS module version 2.0.12 [44] combined with OpenSSL 1.0.2g. The module is created for the purpose of FIPS 140-2 validation. Keys are generated with algorithms from ANSI X9.31. We modified the source code to allow 512-bit keys, since it enforces minimum key size of 1024 bits.

Primes. The module generates probable (p^-, p^+)-*strong* primes. The auxiliary primes are also probable, since they are found by incremental search (with trial division) starting at a random value. We observed, that the auxiliary primes are always 101 bits long. The value is hardcoded and does not change with key size, as required by FIPS 186-4.

Keys. Two highest bits of primes are fixed to binary one, as opposed to the maximal square region used in ANSI X9.31 and FIPS 186-4. The primes must differ in their first 100 bits. The size of private exponent is not checked.

4.2.3 Libgcrypt 1.6.5

Libgcrypt 1.6.5 [45] may be used in regular mode or in FIPS 140-2 mode. We examined key generation in both modes. The library is used by GnuPG, a popular open-source implementation of OpenPGP standard.

We decreased the minimal allowed key size in FIPS mode from 1024 to 512 bits. The library uses continuously seeded PRNG. On Unix systems, new entropy is collected from `/dev/random`. We modified the source code to use `/dev/urandom` instead, solely for the purpose of making the generation of millions of keys possible in reasonable time. It must be noted that `/dev/urandom` should never be used for cryptographic purposes.

Regular mode. Primes are generated by incremental search. Trial division is performed with a table based method. Primality is tested with a Fermat test and several iterations

of the Miller-Rabin test with random base. The prime pairs are generated from a square region obtained by setting two most significant bits to one. No conditions are enforced.

FIPS 140-2 mode. The algorithm from ANSI X9.31 (Appendix B.4) is used to generate (p^-, p^+)-*strong* primes. It is ensured that the primes differ in their top 100 bits. The auxiliary primes always have 101 bits. This value is hardcoded. The two top bits of primes are equal to binary one.

4.2.4 PGP SDK

We examined several versions of PGP. We sporadically found the software on different websites, hence there is no sensible representation of versions. SDK 4, the newest version we were able to obtain with source codes, comes from PGP Desktop 10.0.1 [46]. It supports a FIPS mode and a regular mode. We also located PGP 5.0 (1997) [47] and 6.5.8 (1999) [48], where the FIPS mode is not present, however the key generation in the regular mode follows the same algorithm.

Primes. In regular mode, incremental search with sieve is used to generate primes. Euler's criterion with base 2 and Miller-Rabin tests with a few small prime bases are used to determine primality.

Keys. In regular mode, the two upper bits are fixed to one. It is ensured, that the primes differ at least in one of their top 6 bits, by rejection sampling the candidates. Amongst all the libraries, this is the only condition on $|p - q|$ that is broken with high probability.

FIPS mode. In FIPS mode, (p^-, p^{--}, p^+)-*strong* primes are generated similarly as in the Gordon's algorithm described in section 2.2.6. The intended bit length of the prime is $l = nLen/2$. The size of p^{--} is $\frac{l}{2} - 96$ and p^+ has length $\frac{l}{2} - 32$. This results in $|p^-| \approx \frac{l}{2} - 32$. The prime p will be often short by one bit. The output of the algorithm is visibly biased (see figure A.3). The length of the generated modulus is usually equal to $nLen - 1$, quite often to $nLen - 2$ and only rarely to $nLen$, as intended.

4.2.5 Nettle 3.2

Nettle 3.2 [49] stands out among other libraries, since it is the only one which uses provable primes. It relies on the GNU Multiple Precision arithmetic library (we used GMPlib version 6.1.0).

Primes. Provable primes are constructed recursively using the Maurer's algorithm. As a result, $p - 1$ is guaranteed to have a large factor. The algorithm has no effect on the

prime factors of $p + 1$. Small primes are selected by rejection sampling random numbers and are tested for primality by trial division.

Keys. The keys are generated uniformly from the smaller square region and no checks on conditions are performed.

4.2.6 Apple corecrypto 337

Apple provides source codes of a low level FIPS 140-2 certified cryptographic library corecrypto (version 337) [50]. The library does not directly provide programming interface for developers and is used by higher level libraries. We examined the source code but did not generate RSA keys for further analyses. The library generates two types of keys, depending on the active mode.

In both modes, the prime pairs have the two highest bits set to one, hence the keys are selected from the smaller square region.

Regular mode. Primes are generated by random sampling. Primality is tested with trial division and Miller-Rabin tests with small prime bases. No conditions are enforced.

FIPS 140-2 mode. FIPS 186-4 standard is followed, probable primes with conditions are generated according to Appendix B.3.6. The length of auxiliary primes are set to 101, 141 and 171 bits for primes of ≤ 512, ≤ 1024, > 1024 bits, respectively. It is ensured, that the primes differ in some of the highest 100 bits. Miller-Rabin algorithm with small prime bases is used for primality test.

4.2.7 Crypto++ 5.6.3

No changes were necessary to perform our analysis of Crypto++ 5.6.3 [51].

Primes. The library uses incremental search with sieving. If the size of the interval for a random prime is not a power of two, rejection sampling is used to select a suitable starting point. Primality of the candidate is tested with Miller-Rabin test with base 2. The general-purpose method that verifies the result uses trial division with small primes, a Miller-Rabin test with base 3 and finally a Lucas probabilistic primality test.

The source codes also contain an implementation of Maurer's algorithm for constructing provable primes, however it is not used internally.

Keys. When generating prime pairs, the most significant byte $MSB(p)$ of prime p will be in the interval $\lceil \sqrt{2} \cdot 128 \rceil = 182 \leq MSB(p) \leq 255$. The interval approximates $\left[\sqrt{2} \cdot 2^{\frac{nLen}{2}-1}, 2^{\frac{nLen}{2}} - 1 \right]$. The keys are therefore generated from the almost largest possible square. No conditions are ensured.

4.2.8 SunRsaSign Provider (OpenJDK)

SunRsaSign is the default cryptographic provider for RSA in Java 8. We used OpenJDK 8u91. OpenJDK is the reference implementation since version 7.

Primes. Primes are generated using incremental search with an efficient implementation of sieve of Eratosthenes. Primality is tested with Miller-Rabin test with random bases followed by a Lucas test (referenced as Lucas-Lehmer test).

Keys. Primes are generated from the interval $\left[2^{\frac{nLen}{2}-1}, 2^{\frac{nLen}{2}} - 1\right]$. Prime pairs are selected by rejection sampling. As a result, the keys are selected from the maximal possible region, however there is a certain bias in the distribution. One new prime is generated, when the modulus has incorrect length. The greater prime is preserved in such case. No conditions are placed on the prime pairs.

The primality test is performed before rejection sampling. Since incremental search is unlikely to change the number substantially, the bit length of the modulus could be computed already from the starting points.

4.2.9 Bouncy Castle 1.54

Bouncy Castle [52] is an alternative cryptographic provider for Java. In the version 1.54, the library was updated with a new algorithm. The version 1.53 performed the key generation exactly as OpenJDK.

Primes. Primes are generated by random sampling, until a prime is found. Since the interval is not bounded by powers of 2, rejection sampling is used for the selection of the candidate.

The size of the candidate is checked before primality test is performed by the method `BigInteger.probablePrime` (Miller-Rabin with random bases followed by a Lucas test). However, the candidates are still constructed using the version of `BigInteger` constructor, which generates probable primes using a sieve. Even though certainty (number of Miller-Rabin tests) equal to 1 is selected, the Miller-Rabin and Lucas tests will still ensure that a prime is returned with large probability, where just a random candidate was sought after. Therefore in an attempt to save primality tests, more than necessary are performed. This shows, that `java.math.BigInteger` could benefit from a method, which generates primes from arbitrary intervals.

Keys. The prime pairs are generated from the maximal square region and are selected independently. This is the only library, where the Hamming weight of the modulus is computed. The check is justified by citing [53]. Reportedly, some composite numbers with low-weight may be weak against a version of the number field sieve factorization. It

is required that the weight is larger that quarter of the modulus length. The modulus is converted into non-adjacent form to ensure the representation has the minimal weight. Additionally, the primes must be distinct in some of the highest 100 bits. It is ensured, that the length of the private exponent is at least half of the modulus length.

4.2.10 GNU Crypto 2.0.1

The last stable version 2.0.1 of GNU Crypto [54] was released in 2004. Since then, the project has been a part of GNU Classpath [55]. We compared the code to GNU Classpath 0.99 (latest version from 2012). The keys must be at least 1024 bits long, we changed the requirement to 512.

Primes. Primes are generated by random sampling. The integers are represented as `java.math.BigInteger`. In GNU Classpath 0.99, the primality is tested with the standard method `BigInteger.probablePrime`. GNU Crypto does not take advantage of the implemented primality test. Instead, a custom testing is performed:

1. Test of equality with the first 1000 primes.
2. Trial division by the first 1000 primes.
3. Fermat test with base 2 and with 20 random bases.
4. Euler's criterion with base 2.[4]
5. Miller-Rabin test with 20 random bases.
6. `BigInteger.probablePrime` test is performed for comparison.

The implementation of Miller-Rabin test in GNU Crypto contains a bug, where some prime numbers are evaluated as composite. Recall from section 2.2.1, that an integer n is tested for primality, $n = 2^s r + 1$, r is odd. It either passes $a^r \equiv 1 \pmod{n}$, or $a^{2^j r} \equiv -1 \pmod{n}$ for some j, $0 \leq j < s$. However, in the library the second sufficient condition is not reachable for j other than 0. As a result, p and q are Blum primes ($p \equiv q \equiv 3 \pmod{4}$) and all moduli are Blum integers ($n \equiv 1 \pmod{4}$).

Keys. Prime pairs are generated by rejection sampling. The maximal region is targeted, however the first generated prime is fixed during key generation. Only the second prime is being regenerated, when the modulus has incorrect length. Therefore, if the first prime p was small, the second prime is selected from much shorter interval, than for large p. This results in very biased distribution of most significant bytes.

4. A `HashMap` of weak references to known primes is updated with primes discovered by this method. At the beginning of the test, it is checked if the prime is already in the map.

Instead of checking for the necessary conditions $GCD(p-1,e)=1$ and $GCD(q-1,e)=1$, the check $GCD(p,e)=1$ and $GCD(q,e)=1$ is performed. This leads to an occasional `ArithmeticException`, when the public exponent 65537 is not invertible modulo $\phi(n)=(p-1)\cdot(p+1)$. No additional conditions are placed on the primes. Prime testing is done after the size of the modulus is computed.

4.2.11 Cryptix JCE 20050328

The development of Cryptix JCE [56] cryptographic provider has been stopped in 2005.

Primes. Primes are generated with `java.math.BigInteger`, hence the same as for SunRsaSign applies.

Keys. Prime pairs are selected by rejection sampling and the maximal region is targeted. New pair is generated, when the size of the modulus is insufficient, hence there is no bias in the distribution. Prime testing is done before the size of the modulus is computed.

4.2.12 FlexiProvider 1.7p7

FlexiProvider 1.7p7 [57] is a toolkit for the Java Cryptography Architecture. The keys are generated in the exact same way as in Cryptix. The library support multi-prime RSA. We sometimes refer to the library as FlexiCore, which is the default provider for RSA.

Primes. Primes are generated with `FlexiBigInt`, however it is just a wrapper for `java.math.BigInteger`, hence the same as for SunRsaSign applies.

Keys. Rejection sampling is responsible for prime pair generation. The maximal region is achieved with no bias, since new prime pair is generated when the modulus is too short. Prime testing is done before the size of the modulus is computed.

4.2.13 Cryptlib 3.4.3

We decreased the minimal key length from 1024 to 512 bits in cryptlib 3.4.3 [58]. In order to avoid exporting RSA keys using the API, we added a simple workaround for printing the key material directly. The library with this modification should not be used for other purposes, since it compromises security of otherwise excellent security toolkit.

Primes. Incremental search generates primes using an efficient sieving method. A series of small primes is used for Miller-Rabin test. A Fermat test to the base of 2 is used to verify the result, however if 2 is a strong liar, it is also a Fermat liar (section 2.2.1).

Keys. Prime pairs are selected from the smaller square region. It is ensured that $|p - q| > 128$ bits and $|d| > nLen/2$.

4.2.14 mbedTLS 2.2.1

We did not have to modify the source code for generating keys using mbedTLS 2.2.1 [59] (formerly known as PolarSSL) in any way, however a small change in printing large integers made gathering our dataset easier.

Primes. Primes are generated by incremental search starting at a random value. Trial division phase is followed by several iterations of the Miller-Rabin test with random bases.

Keys. The keys are generated from the maximal region (apart from ordering the primes $p > q$) given by precise length of modulus and equal length of primes. The method of choice is rejection sampling. Both primes are generated again in case that the pair is from the incorrect region, therefore the distribution is not biased. No checks are performed.

4.2.15 LibTomCrypt 1.17

LibTomCrypt 1.17 [60] requires an arithmetic library from the LibTom project (in our case TomsFastMath 0.13). In order to generate 512-bit keys, we changed the bound from 1024 bits. We noticed that the LibTomCrypt library is also used by PyCrypto 2.6.1.

Primes. Candidates for primality testing are selected randomly. Testing is performed by trial division followed with Miller-Rabin tests with small prime bases.

Keys. Keys are selected from the smaller square region without checking for conditions.

4.2.16 WolfSSL 3.9.0

WolfSSL 3.9.0 [61] (formerly CyaSSL) is an embedded SSL library.

Primes. The source code for prime generation is based on LibTomMath 0.38. Hence, identically as in LibTomCrypt, primes are selected randomly and subjected to trial division and Miller-Rabin tests with small prime bases.

Keys. Prime pairs are selected independently from the smaller square region and no conditions are enforced.

4.2.17 Botan 1.11.29

The requirements on minimal key size in Botan 1.11.29 [62] were relaxed from 1024 bits to 512 bits for our analysis.

Primes. Incremental search with table based sieving generates candidates that are tested for primality using Miller-Rabin with random bases.

Keys. No conditions are placed on prime pairs. The keys are generated from the small square region, since the two high order bits are set to one.

4.2.18 Microsoft Cryptographic Service Provider

We performed a black-box analysis of the RSA provider in Microsoft Windows 10. We tested CryptoAPI, CNG and the default RSA provider in .NET. We did not discover any differences.

By factoring 256-bit primes (using YAFU [63]), we discovered that (p^-, p^+)-*strong* primes are generated. The auxiliary primes have length of 101 to 120 bits (see appendix C). We hypothesize, that algorithms from ANSI X9.31 or FIPS 186-4 are implemented and that for larger keys, the size of auxiliary primes will follow the requirements set by the standards. The primes are uniformly distributed in the maximal square region.

4.3 Comparison

Table 4.1 contains a comparison of the surveyed cryptographic libraries, with respect to the described properties. Most notably, libraries can be sorted to a few categories with respect to the prime search methods and the prime pair selection methods.

Random sampling and incremental search are often used to generate primes. Large factors of $p - 1$ and $p + 1$ are usually ensured by the algorithm described in FIPS 186-4 [21, Appendix B.3.6], however PGP implements a different algorithm. Only a single library (Nettle) is using provable primes.

Several methods of prime pair selection are used. Some libraries target the maximal region defined by the precise bit length of primes and modulus (performing rejection sampling to eliminate moduli short by one bit). The practical region obtained by fixing the top two bits of the primes to binary one is strongly favored. A larger square region may be targeted, characterized by the prime intervals $\left[\sqrt{2} \cdot 2^{\frac{n}{2}-1}, 2^{\frac{n}{2}} - 1\right]$. The only irregular region that is not easily described is produced by PGP library in FIPS mode.

Some libraries perform checks for difference of p and q and size of the private exponent d, however the probability that these are not suitable is extremely low. PGP is an exception, since it requires that the primes differ somewhere in their top 6 bits. The opposite happens with considerably high probability.

Library / Standard	Version	Language	Prime search method	Prime pair selection	Large factor of $p-1$	Large factor of $p+1$	$\lvert p-q \rvert$ check	$\lvert d \rvert$ check	Notes
Apple corecrypto	337 / 337 FIPS	C	Rand./ FIPS	11_2/ 11_2	×/ ✓	×/ ✓	×/ ✓	×/ ×	
Botan	1.11.29	C++	Inc.	11_2	×	×	×	×	
Bouncy Castle	1.54	Java	Inc.	$\sqrt{2}$	×	×	✓	✓	Checks Hamming weight of the modulus
Cryptix JCE	20050328	Java	Inc.	RS	×	×	×	×	Rejection sampling is not biased
cryptlib	3.4.3	C	Inc.	11_2	×	×	✓	✓	
Crypto++	5.6.3	C++	Inc.	$\sqrt{2}$	×	×	×	×	$255 \geq$ prime MSB $\geq 182 = \lceil \sqrt{2} \cdot 128 \rceil$
FlexiProvider	1.7p7	Java	Inc.	RS	×	×	×	×	Rejection sampling is not biased
GNU Crypto	2.0.1	Java	Rand.	RS	×	×	×	×	Rejection sampling is more biased
libgcrypt	1.6.5 / 1.6.5 FIPS	C	Inc./ FIPS	11_2/ 11_2	×/ ✓	×/ ✓	×/ ✓	×/ ×	GnuPG 2.0.30 / FIPS mode
LibTomCrypt	1.17	C	Rand.	11_2	×	×	×	×	
mbedTLS	2.2.1	C	Inc.	RS	×	×	×	×	Rejection sampling is not biased
MS CryptoAPI	Windows 10	C	FIPS?	$\sqrt{2}$	✓	✓	-	-	Source code not available
Nettle	3.2	C	Maurer	11_2	✓	×	×	×	
OpenSSL	1.0.2g	C	Inc.	11_2	×	×	×	×	$(p-1)/2$ does not have small factors
OpenSSL	2.0.12 FIPS	C	FIPS	11_2	✓	✓	✓	×	OpenSSL 2.0.12 FIPS 140-2 module
PGP SDK	4.x / 4.x FIPS	C	Inc./ PGP	11_2/ PGP	×/ ✓	×/ ✓	✓/ ✓	×/ ×	PGP Desktop 10.0.1 / FIPS mode
SunJCE Provider	OpenJDK 8u91	Java	Inc.	RS	×	×	×	×	Rejection sampling is less biased
WolfSSL	3.9.0	C	Rand.	11_2	×	×	×	×	
NIST FIPS 186-4	2013	-	FIPS/ Rand.	$\sqrt{2}$	✓/ ×	✓/ ×	✓	✓	Probable, provable or (p^-, p^+)-*strong* primes
ANSI X9.31 / X9.44	1998 / 2007	-	FIPS	$\sqrt{2}$	✓	✓	✓	✓	(p^-, p^+)-*strong* primes
IEEE 1363	2000	-	IEEE$^+$/ Rand.	IEEE	✓/ ×	✓/ ×	×	×	Probable or (p^-, p^+, p^{--})-*strong* primes
RSAES-OAEP	2000	-	Inc.	$\sqrt{2}$	×	×	×	×	
ISO/IEC 18033-2	2006	-	-	-	×	×	×	×	Only necessary conditions
PKCS #1 (RFC 3447)	2003	-	-	-	×	×	×	×	Only necessary conditions

Table 4.1: *Comparison of cryptographic libraries and standards.* ***Prime search method:*** *incremental search (Inc.); random sampling (Rand.); FIPS 186-4 Appendix B.3.6 or equivalent algorithm from ANSI X9.31 (FIPS); Maurer's algorithm (Maurer); PGP strong primes (PGP); IEEE 1363 strong primes (IEEE$^+$).* ***Prime pair selection:*** *rejection sampling (RS); primes with the two most significant bits* 11_2 *(11_2); primes from the interval* $\left[\sqrt{2} \cdot 2^{\frac{n}{2}-1}, 2^{\frac{n}{2}} - 1\right]$ *($\sqrt{2}$); IEEE 1363 algorithm (IEEE).*

5 Conclusions

Our analysis shows, that there is no one correct method for generating RSA keys, which would always have to be followed. On the contrary, several algorithms can be used in various combinations without compromising security of the cryptosystem. To our satisfaction, all cryptographic libraries produce secure RSA keys, free from trapdoors and critical properties of the keys.

A few bits of entropy are sometimes sacrificed to make the key generation process more efficient. As a result, a lot can be said even about processes inside black-box implementations. The distribution of primes uncovers the intervals, from which the values are being generated. It may even reveal the algorithm responsible, if the choice is not done uniformly. Interestingly, some of these properties manifest also in the public modulus.

The benefits of methods, which produce uniformly distributed keys, seem to be outbalanced by their relative complexity. If such distribution would be spotted, it would paradoxically reveal more about the process than a biased distribution.

We confirmed that bugs and implementation choices may cause that the keys have some identifying properties. These may be harmless for security purposes, but may have further implications for reverse engineering. Some of them, such as the distribution of remainders of primes and moduli modulo small primes, only become apparent from large sets of keys.

The debate on usage of "strong" primes in practice seems to be nearing a conclusion, since software libraries only generate such primes when it is mandated by certification. Even standardization bodies seem to have acknowledged, that large random primes are "strong" with overwhelming probability. The standard IEEE 1363 states this fact directly and FIPS 186-4 only mandates the special construction techniques for smaller keys. For the purpose of our analysis, the detection of this property in generated keys required the most computational effort and may be infeasible for large keys.

5.1 Recommendations

We recommend to choose freely from almost all of the described methods, based on the available skills and the requirements mandated by the platform where the keys are generated.

Small bias in the distribution of RSA primes and moduli does not seem to give an advantage to an attacker. We have shown how to minimize amount of random bits in both generation of primes and prime pairs with only small sacrifice of uniformity of the output.

We discussed which primality tests provide their results with highest confidence. In case that one would doubt the quality of probable primes, we described how to construct provable primes.

We demonstrated, that random keys are safe from many special-purpose factorization methods. The conclusion was supported by citing research on the distribution of size of the largest factor of a random integer. We also estimated the probability that other negative properties occur. However, if "strong" primes must be ensured, we recommend to use the newer algorithm, as described by FIPS 186-4, rather than the older Gordon's algorithm. The latter does not achieve uniform distribution of primes, and as revealed by the implementation in PGP, in many cases not even the correct length of the modulus.

Generally, we would recommend to generate primes independently. When the negative outcome of rejection sampling is obtained, generate both primes, rather than persisting one from the previous iteration.

5.2 Future work

Software implementations seem to provide keys that are good enough, under the assumption that other parts of the library function properly. And yet, thousands of keys on the internet are susceptible to simple attacks, due to shared primes or moduli, as has been repeatedly proven [5, 6]. We believe that software engineers would benefit from analysis of pseudorandom number generators used in cryptographic libraries. Even when standardized generators are used and their functionality is verified with test vectors, seeding them with insufficient entropy weakens RSA up to the point of no security. We would like to see which libraries rely on insecure sources of randomness, ideally by performing an in-depth analysis as done by [6] for OpenSSL.

A comparison of cryptographic libraries across several versions could provide insights into trends in RSA key generation or show that the algorithms are rarely changed, once implemented.

We believe, that our black box approach could be used to explore more sources of RSA keys, even beyond software libraries. The analysis would strengthen our results and potentially discover other methods, not used in the surveyed libraries and literature.

Bibliography

[1] D. Loebenberger and M. Nüsken. "Notions for RSA Integers". In: *International Journal of Applied Cryptology*. Inderscience Publishers, 2014, pp. 116–138. DOI: 10.1504/IJACT.2014.062723.

[2] R. D. Silverman. "Fast Generation Of Random, Strong RSA Primes". In: *CryptoBytes Technical Newsletter*. RSA Laboratories, 1997.

[3] R. L. Rivest and R. D. Silverman. "Are 'Strong' Primes Needed for RSA?" In: *The 1997 RSA Laboratories Seminar Series. Proceedings*. 1999.

[4] T. Kleinjung et al. "Factorization of a 768-Bit RSA Modulus". In: *Advances in Cryptology – CRYPTO 2010. Proceedings*. Springer-Verlag, 2010, pp. 333–350. DOI: 10.1007/978-3-642-14623-7_18.

[5] A. K. Lenstra et al. "Ron was wrong, Whit is right." In: *IACR Cryptology ePrint Archive* (2012). URL: http://eprint.iacr.org/2012/064.

[6] N. Heninger, Z. Durumeric, E. Wustrow, and J. A. Halderman. "Mining Your Ps and Qs: Detection of Widespread Weak Keys in Network Devices". In: *21st USENIX Conference on Security Symposium. Proceedings*. USENIX Association, 2012, pp. 35–35.

[7] D. J. Bernstein et al. "Factoring RSA Keys from Certified Smart Cards: Coppersmith in the Wild". In: *Advances in Cryptology - ASIACRYPT 2013. Proceedings*. Springer-Verlag, 2013, pp. 341–360. DOI: 10.1007/978-3-642-42045-0_18.

[8] R. L. Rivest, A. Shamir, and L. Adleman. "A Method for Obtaining Digital Signatures and Public-key Cryptosystems". In: *Communications of the ACM* (1978), pp. 120–126. DOI: 10.1145/359340.359342.

[9] D. Boneh. "Twenty Years of Attacks on the RSA Cryptosystem". In: *Notices of the AMS* 46 (1999), pp. 203–213.

[10] J. J. Quisquater and C. Couvreur. "Fast decipherment algorithm for RSA public-key cryptosystem". In: *Electronics Letters* 18.21 (1982), pp. 905–907. DOI: 10.1049/el:19820617.

[11] P. Rogaway and S. M. Matyas. *RSAES-OAEP Encryption Scheme Algorithm specification and supporting documentation*. 2000. DOI: 10.1.1.720.5462.

[12] J. Jonsson and B. Kaliski. *Public-Key Cryptography Standards (PKCS) #1: RSA Cryptography Specifications Version 2.1*. RFC 3447. 2003. URL: https://tools.ietf.org/html/rfc3447.

[13] A. Decker and P. Moree. "Counting RSA-Integers". In: *Results in Mathematics* 52.1 (2008), pp. 35–39. DOI: 10.1007/s00025-008-0285-5.

[14] A. J. Menezes, P. C. V. Oorschot, S. A. Vanstone, and R. L. Rivest. *Handbook of Applied Cryptography*. 1st Edition. CRC Press, 1996. ISBN: 0849385237.

[15] M. Joye and P. Paillier. "Fast Generation of Prime Numbers on Portable Devices: An Update". In: *Cryptographic Hardware and Embedded Systems - CHES 2006. Proceedings*. Springer-Verlag, 2006, pp. 160–173. DOI: 10.1007/11894063_13.

[16] D. E. Knuth and L. Trabb Pardo. "Analysis of a simple factorization algorithm". In: *Theoretical Computer Science* 3.3 (1976), pp. 321–348. DOI: http://dx.doi.org/10.1016/0304-3975(76)90050-5.

[17] R. Solovay and V. Strassen. "A Fast Monte-Carlo Test for Primality". In: *SIAM Journal on Computing* 6.1 (1977), pp. 84–85. DOI: 10.1137/0206006.

[18] G. L. Miller. "Riemann's Hypothesis and Tests for Primality". In: *Journal of Computer and System Sciences* 13.3 (1976), pp. 300–317. DOI: 10.1016/S0022-0000(76)80043-8.

[19] M. O. Rabin. "Probabilistic algorithm for testing primality". In: *Journal of Number Theory* 12.1 (1980), pp. 128–138. DOI: 10.1016/0022-314X(80)90084-0.

[20] P. Berrizbeitia and T. Berry. "Generalized Strong Pseudoprime Tests and Applications". In: *J. Symb. Comput.* 30.2 (Aug. 2000), pp. 151–160. DOI: 10.1006/jsco.1999.0343.

[21] National Institute of Standards and Technology. *Digital Signature Standard (DSS)*. FIPS 186-4. 2013. DOI: 10.6028/NIST.FIPS.186-4. URL: http://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.186-4.pdf.

[22] J. Brandt and I. Damgård. "On Generation of Probable Primes by Incremental Search". In: *Advances in Cryptology — CRYPTO' 92. Proceedings*. Springer-Verlag, 1993, pp. 358–370. DOI: 10.1007/3-540-48071-4_26.

[23] M. Joye, P. Paillier, and S. Vaudenay. "Efficient Generation of Prime Numbers". In: *Cryptographic Hardware and Embedded Systems – CHES 2000. Proceedings*. Springer-Verlag, 2000, pp. 340–354. DOI: 10.1007/3-540-44499-8_27.

[24] U. M. Maurer. "Fast generation of prime numbers and secure public-key cryptographic parameters". In: *Journal of Cryptology* 8.3 (1995), pp. 123–155. DOI: 10.1007/BF00202269.

[25] J. Gordon. "Strong Primes are Easy to Find". In: *Advances in Cryptology: EUROCRYPT '84. Proceedings*. Springer-Verlag, 1985, pp. 216–223. DOI: 10.1007/3-540-39757-4_19.

[26] J. M. Pollard. "Theorems on factorization and primality testing". In: *Mathematical Proceedings of the Cambridge Philosophical Society* 76 (03 1974), pp. 521–528. DOI: 10.1017/S0305004100049252.

[27] H. C. Williams. "A $p + 1$ Method of Factoring". In: *Mathematics of Computation*. Vol. 39. American Mathematical Society, 1982, pp. 225–234.

[28] R. S. Lehman. "Factoring large integers". In: *Mathematics of Computation*. Vol. 28. American Mathematical Society, 1974, pp. 637–646. DOI: 10.1090/S0025-5718-1974-0340163-2.

[29] H. W. Lenstra Jr. "Factoring integers with Elliptic Curves". In: *The Annals of Mathematics* 126 (3 1987), pp. 649–673. URL: http://wstein.org/edu/124/lenstra/lenstra.pdf.

[30] C. Pomerance. "The Quadratic Sieve Factoring Algorithm". In: *Advances in Cryptology: EUROCRYPT '84. Proceedings*. Springer-Verlag, 1985, pp. 169–182. DOI: 10.1007/3-540-39757-4_17.

[31] J. M. Pollard. "Factoring with cubic integers". In: *The development of the number field sieve*. Springer-Verlag, 1993, pp. 4–10. DOI: 10.1007/BFb0091536.

[32] M. J. Wiener. "Cryptanalysis of Short RSA Secret Exponents". In: *Advances in Cryptology — EUROCRYPT '89. Proceedings*. Springer-Verlag, 1990, pp. 372–372. DOI: 10.1007/3-540-46885-4_36.

[33] D. Boneh and G. Durfee. "Cryptanalysis of RSA with Private Key d less than $N^{0.292}$". In: *IEEE Transactions on Information Theory* 46.4 (2000), pp. 1339–1349. DOI: 10.1109/18.850673.

[34] M. Gysin and J. Seberry. "Generalised Cycling Attacks on RSA and Strong RSA Primes". In: *Information Security and Privacy: ACISP'99. Proceedings*. Springer-Verlag, 1999, pp. 149–163. DOI: 10.1007/3-540-48970-3_13.

[35] D. Coppersmith. "Small Solutions to Polynomial Equations, and Low Exponent RSA Vulnerabilities". In: *Journal of Cryptology* 10.4 (1997), pp. 233–260. DOI: 10.1007/s001459900030.

[36] L. Bello. *Debian Security Advisory. DSA-1571-1 openssl – predictable random number generator*. cit. [2016-05-20]. URL: https://www.debian.org/security/2008/dsa-1571.

[37] D. J. Bernstein, N. Heninger, and T. Lange. *Batch gcd*. cit. [2016-05-20]. 2012. URL: http://facthacks.cr.yp.to/batchgcd.html.

[38] D. Coppersmith. "Finding a Small Root of a Bivariate Integer Equation; Factoring with High Bits Known". In: *15th Conference on Theory and Application of Cryptographic Techniques, EUROCRYPT'96, Proceedings*. Springer-Verlag, 1996, pp. 178–189.

[39] IEEE. *Standard Specifications for Public-Key Cryptography*. IEEE Std 1363. 2000. DOI: 10.1109/IEEESTD.2000.92292.

[40] International Organization for Standardization. *Information technology – Security techniques – Encryption algorithms – Part 2: Asymmetric ciphers*. ISO/IEC 18033-2. 2006.

[41] *OpenSSL 1.0.2g*. cit. [2016-04-21]. URL: https://www.openssl.org/source/openssl-1.0.2g.tar.gz.

[42] *LibreSSL 2.3.4*. cit. [2016-05-13]. URL: http://ftp.openbsd.org/pub/OpenBSD/LibreSSL/libressl-2.3.4.tar.gz.

[43] I. Mironov. *Factoring RSA Moduli. Part II*. cit. [2016-05-20]. 2012. URL: https://windowsontheory.org/2012/05/17/factoring-rsa-moduli-part-ii/.

[44] *OpenSSL FIPS 2.0.12*. cit. [2016-04-21]. URL: https://www.openssl.org/source/openssl-fips-2.0.12.tar.gz.

[45] *libgcrypt 1.6.5*. cit. [2016-04-21]. URL: https://gnupg.org/ftp/gcrypt/libgcrypt/libgcrypt-1.6.5.tar.bz2.

[46] *PGP Desktop 10.0.1*. cit. [2016-04-21]. URL: http://www.symantec.com/connect/sites/default/files/PGPDesktop10.0.1_Source.zip.

[47] *PGP 5.0i*. cit. [2016-04-27]. URL: http://www.pgpi.org/cgi/download.cgi?filename=pgp50i-unix-src.tar.gz.

[48] *PGP 6.5.8*. cit. [2016-04-27]. URL: http://www.pgpi.org/cgi/download.cgi?filename=pgpsrc658unix.tar.gz.

[49] *Nettle 3.2*. cit. [2016-04-21]. URL: https://ftp.gnu.org/gnu/nettle/nettle-3.2.tar.gz.

[50] *Apple corecrypto 337*. cit. [2016-05-01]. URL: https://developer.apple.com/cryptography/.

[51] *Crypto++ 5.6.3*. cit. [2016-04-21]. URL: https://www.cryptopp.com/cryptopp563.zip.

[52] *Bouncy Castle 1.54*. cit. [2016-04-21]. URL: https://www.bouncycastle.org/download/bcprov-jdk15on-154.jar.

[53] K. Hayasaka and T. Takagi. "An Experiment of Number Field Sieve over GF(P) of Low Hamming Weight Characteristic". In: *3rd International Conference on Coding and Cryptology – IWCC'11. Proceedings*. Springer-Verlag, 2011, pp. 191–200.

[54] *GNU Crypto 2.0.1*. cit. [2016-04-28]. URL: https://www.gnu.org/software/gnu-crypto/.

[55] *GNU Classpath 0.99*. cit. [2016-04-28]. URL: https://www.gnu.org/software/classpath/announce/20120307.html.

[56] *Cryptix JCE 20050328*. cit. [2016-04-28]. URL: http://www.cryptix.org/cryptix-jce-20050328-snap.zip.

[57] *FlexiProvider 1.7p7*. cit. [2016-04-28]. URL: https://www.flexiprovider.de/download/FlexiCoreProvider-1.7p7.signed.jar.

[58] *cryptlib 3.4.3*. cit. [2016-04-21]. URL: ftp://ftp.franken.de/pub/crypt/cryptlib/cl343.zip.

[59] *mbedTLS 2.2.1*. cit. [2016-04-21]. URL: https://github.com/ARMmbed/mbedtls/archive/mbedtls-2.2.1.tar.gz.

[60] *LibTomCrypt 1.17*. cit. [2016-04-21]. URL: https://github.com/libtom/libtomcrypt/releases/download/1.17/crypt-1.17.tar.bz2.

[61] *WolfSSL 3.9.0*. cit. [2016-04-21]. URL: https://github.com/wolfSSL/wolfssl/archive/v3.9.0.tar.gz.

[62] *Botan 1.11.29*. cit. [2016-04-21]. URL: https://github.com/randombit/botan/archive/1.11.29.tar.gz.

[63] *YAFU: Yet Another Factorization Utility*. cit. [2016-05-12]. 2013. URL: http://sourceforge.net/projects/yafu/.

A The most significant byte of primes

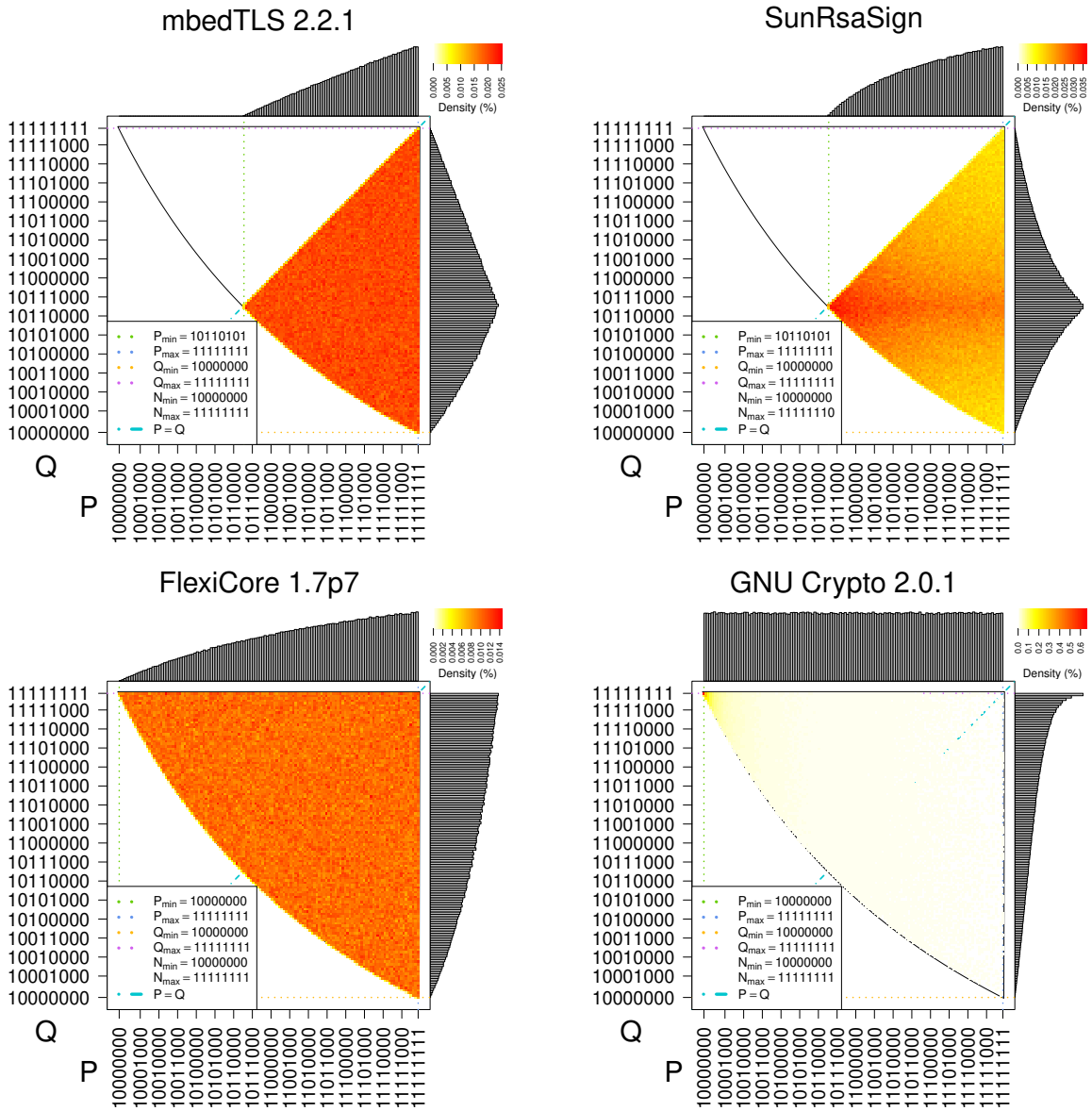

Figure A.1: *The libraries mbedTLS, SunRsaSign, FlexiProvider and GNU Crypto all target the maximal region for RSA integers, characterized by precise bit length of the primes and precise bit length of the modulus. However, their distributions are not identical due to small differences in the implementation of rejection sampling. FlexiProvider and mbedTLS differ, because the latter orders the primes such that $p > q$.*

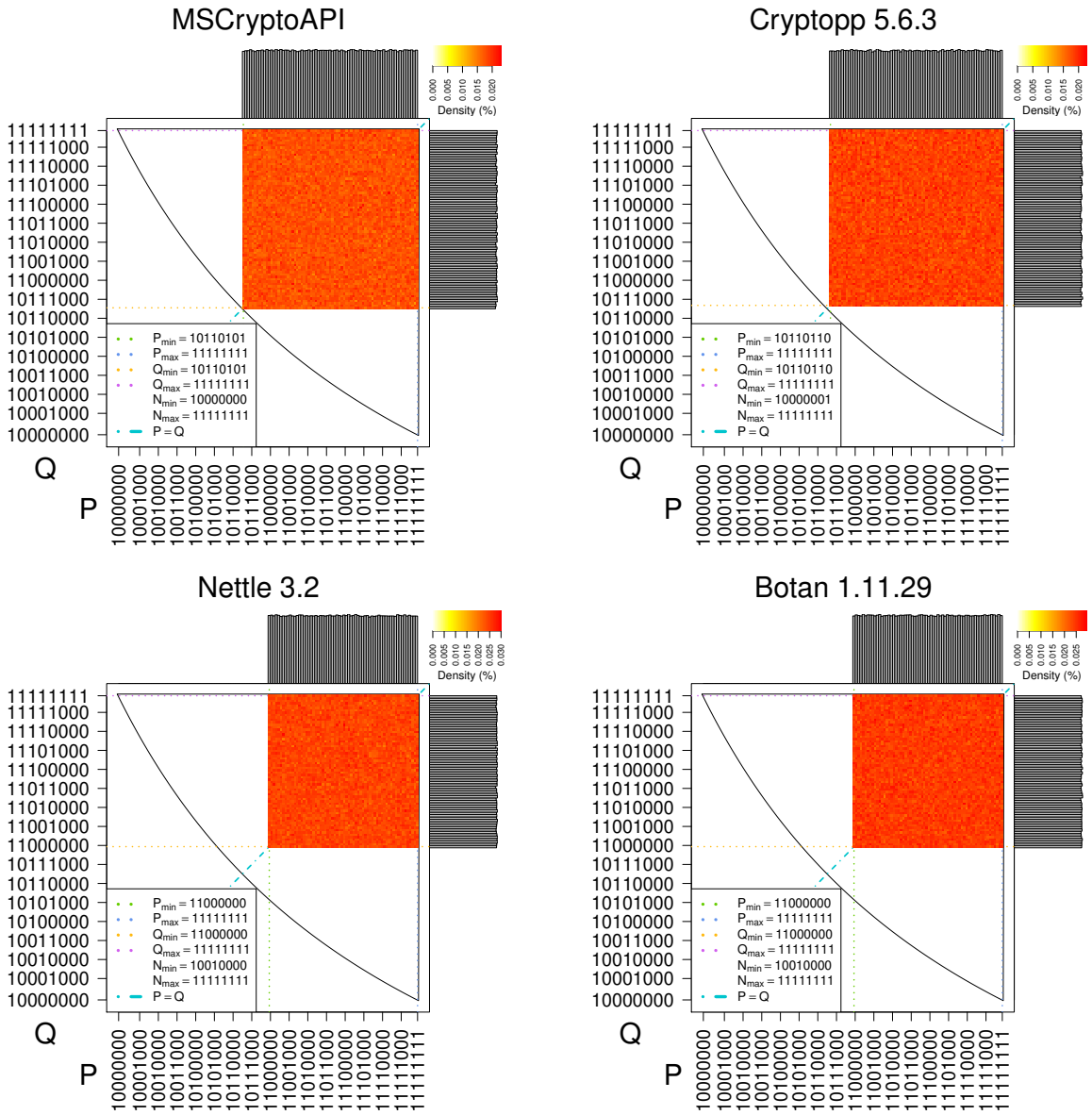

Figure A.2: *Practical regions for selecting prime pairs. Microsoft libraries target the interval* $\left[\sqrt{2}\cdot 2^{\frac{n}{2}-1}, 2^{\frac{n}{2}}-1\right]$. *Crypto++ approximates the lower bound by generating the MSB from* $\lceil\sqrt{2}\cdot 128\rceil = 182$ *to 255. From the distribution of Nettle, it is not noticeable that provable primes are used. The distribution is similar to Botan, which uses probable primes. Nettle and Botan generate the primes from the interval* $\left[3\cdot 2^{\frac{n}{2}-2}, 2^{\frac{n}{2}}-1\right]$, *achieved by fixing the two upper bits of primes to one.*

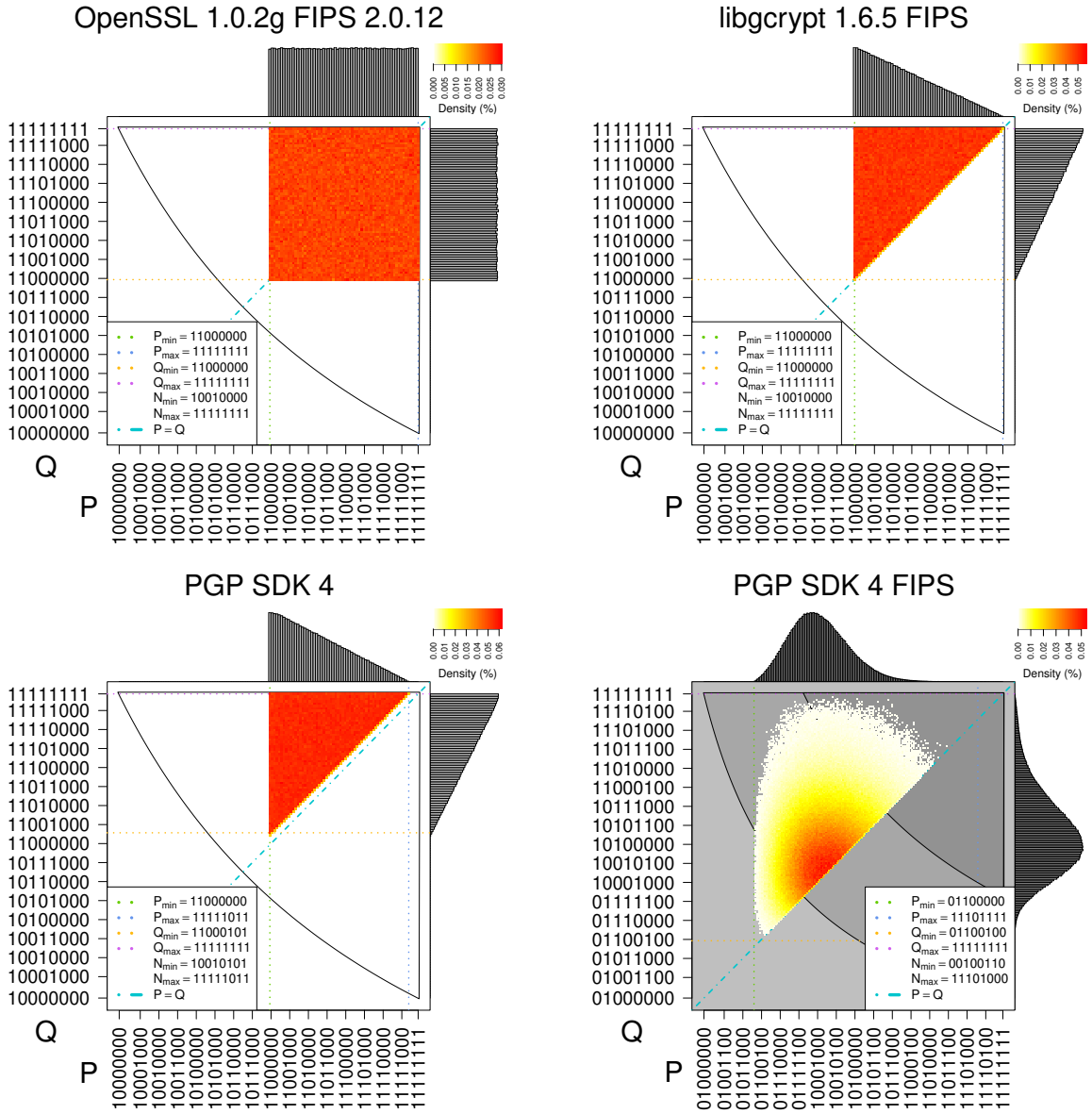

Figure A.3: *OpenSSL FIPS 140-2 module and libgcrypt in FIPS mode generate strong primes uniformly. PGP SDK in regular mode ensures that primes differ in their top 6 bits. PGP SDK in FIPS mode generates strong primes similarly as Gordon's algorithm. The distribution is not uniform. Additionally, the length of the primes may differ by one bit (notice the different values on the axes). As a result, the moduli can be one or two bits shorter than desired (notice the three distinctly shaded gray areas representing different modulus length).*

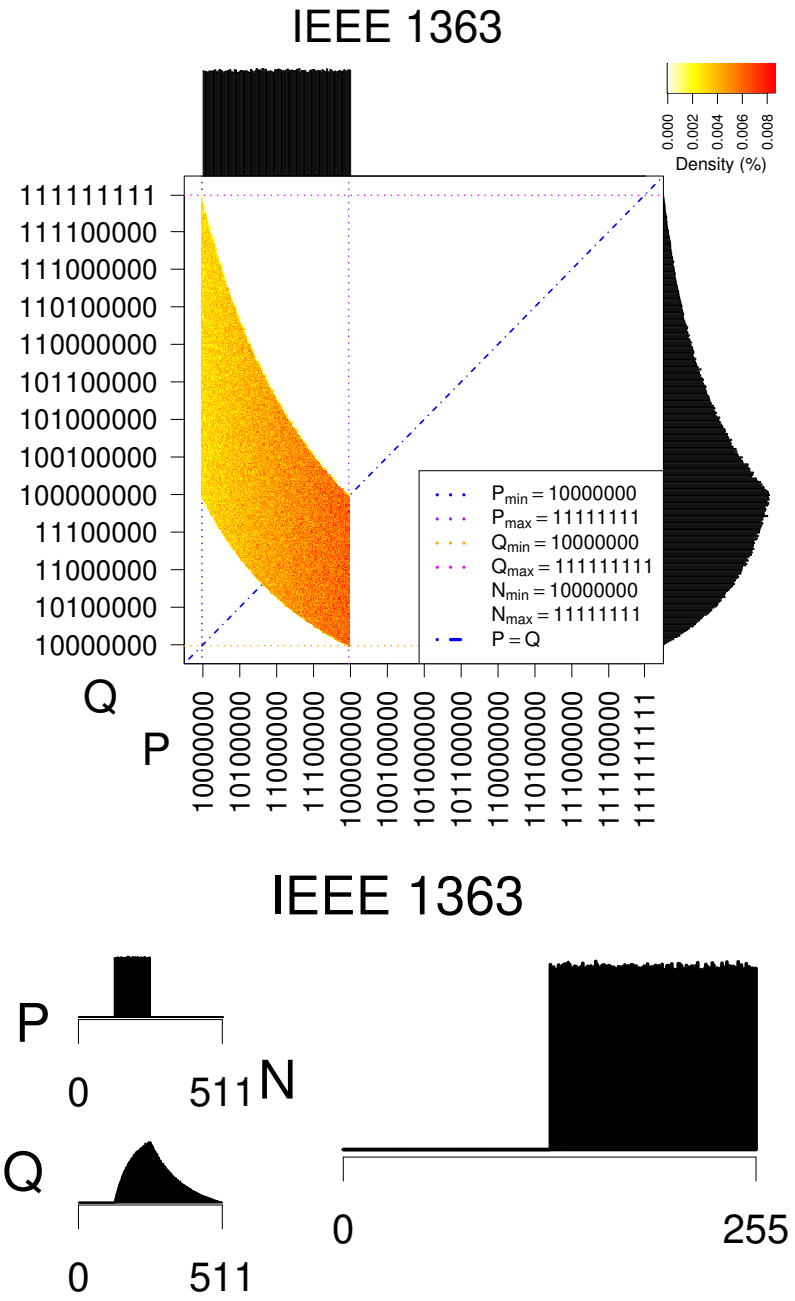

Figure A.4: *The algorithm from the standard IEEE 1363-2000 achieves uniform distribution of the modulus. However, one of the primes is sometimes one bit longer than half of the key size.*

B The most significant byte of modulus

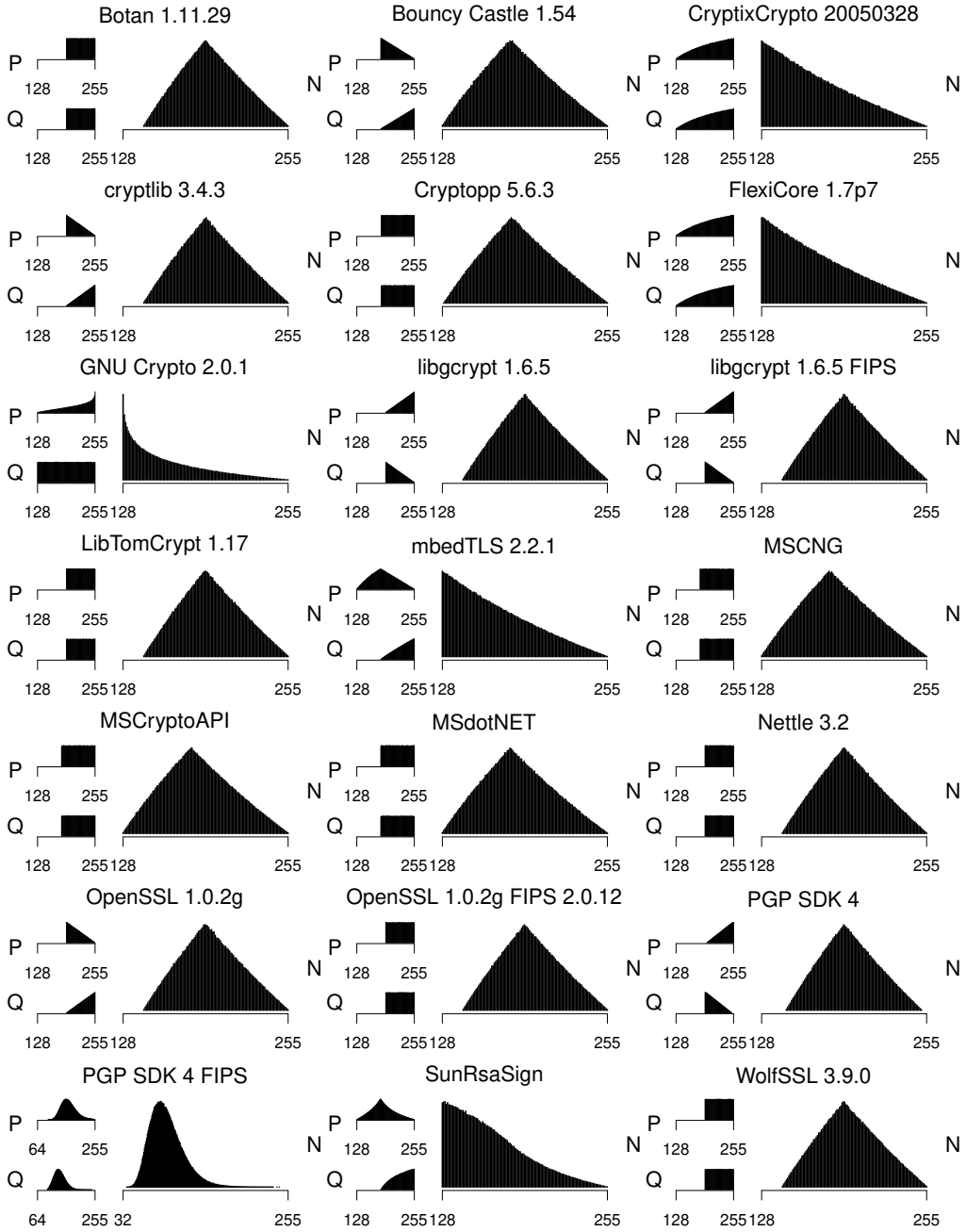

Figure B.1: *The different distributions of the most significant byte of primes also affect the distribution of the modulus.*

C Factorization of $p - 1$

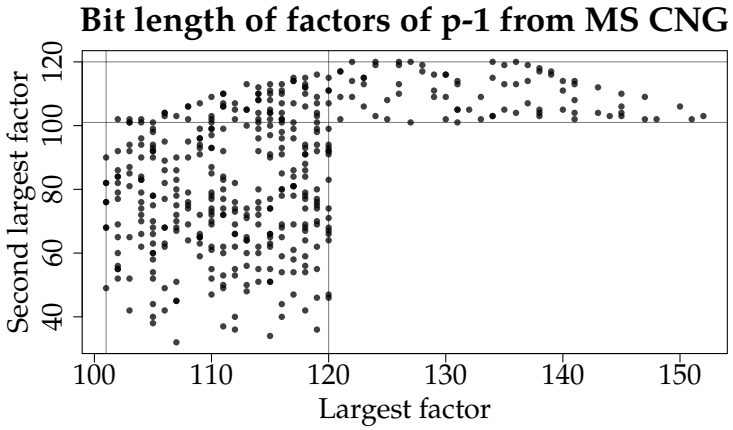

Figure C.1: *Size of the largest and the second largest prime factor of a 256-bit $p - 1$ value obtained from a 512-bit RSA key generated with Microsoft CNG. Each $p - 1$ has at least one prime factor of length 101 to 120 bits. Therefore, the library is generating strong primes with auxiliary primes of randomized length, as opposed to fixing the length to 101 bits. The prime factors of $p + 1$ also fall in the specified region.*

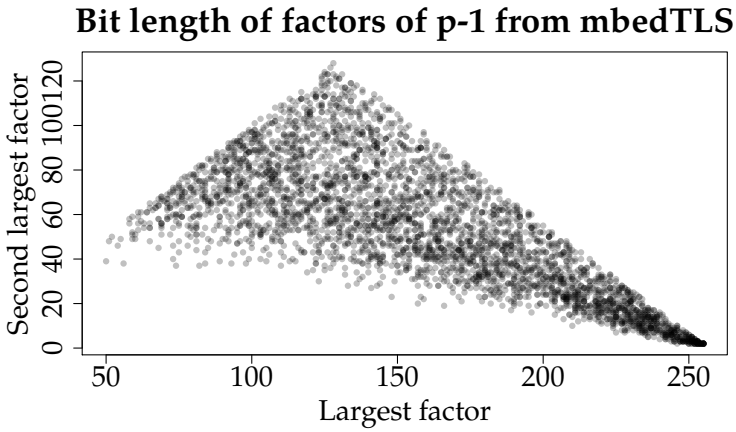

Figure C.2: *The library mbedTLS does not generate strong primes, hence the size of the prime factors of $p - 1$ is random. Yet none of the 3600 random 256-bit values is 2^{50}-smooth. Factorization of a weak key using Pollard's $p - 1$ method would require more than 2^{53} modular multiplications.*

D List of files and methods in cryptographic libraries

Library	Version	Source file / Class	Method
Apple corecrypto	337	`ccrsa/src/ccrsa_generate_key.c`	`ccrsa_generate_key`
Apple corecrypto	337 FIPS	`ccrsa/src/ccrsa_generate_fips186_key.c`	`ccrsa_generate_fips186_key`
Botan	1.11.29	`src/lib/pubkey/rsa/rsa.cpp`	`RSA_PrivateKey`
Bouncy Castle	1.54	`RSAKeyPairGenerator`	`generateKeyPair`
Cryptix JCE	20050328	`RSAKeyPairGenerator`	`generateKeyPair`
cryptlib	3.4.3	`context/kg_rsa.c`	`generateRSAkey`
Crypto++	5.6.3	`rsa.cpp / InvertibleRSAFunction`	`GenerateRandom`
FlexiProvider	1.7p7	`RSAKeyPairGenerator`	`genKeyPair`
GNU Crypto	2.0.1	`RSAKeyPairGenerator`	`generate`
libgcrypt	1.6.5	`cipher/rsa.c`	`generate_std`
libgcrypt	1.6.5 FIPS	`cipher/rsa.c`	`generate_x931`
LibTomCrypt	1.17	`src/pk/rsa/rsa_make_key.c`	`rsa_make_key`
mbedTLS	2.2.1	`library/rsa.c`	`mbedtls_rsa_gen_key`
Nettle	3.2	`rsa-keygen.c`	`rsa_generate_keypair`
OpenSSL	1.0.2g	`crypto/rsa/rsa_gen.c`	`rsa_builtin_keygen`
OpenSSL	2.0.12 FIPS	`crypto/rsa/rsa_x931g.c`	`RSA_X931_generate_key_ex`
PGP	10.0.1	`libs2/pgpsdk/priv/crypto/pubkey/pgpRSAKey.c`	`rsaSecGenerate`
PGP	10.0.1 FIPS	`libs2/pgpsdk/priv/crypto/pubkey/pgpRSAKey.c`	`rsaSecGenerate`
PGP	6.5.8	`libs/pgpcdk/priv/keys/pubkey/pgpRSAkey.c`	`rsaSecGenerate`
SunRsaSign	JDK 8u91	`RSAKeyPairGenerator`	`generateKeyPair`
WolfSSL	3.9.0	`wolfcrypt/src/rsa.c`	`wc_MakeRsaKey`

Table D.1: *The methods responsible for generating RSA keys and their location.*

Library	Version	Source file / Class	Method
Apple corecrypto	337	`cczp/src/cczp_random_prime.c`	`cczp_random_prime`
Apple corecrypto	337 FIPS	`ccrsa/src/ccrsa_generate_fips186_key.c`	`ccrsa_crt_make_fips186_key`
Botan	1.11.29	`src/lib/math/numbertheory/make_prm.cpp`	`random_prime`
Bouncy Castle	1.54	`RSAKeyPairGenerator`	`chooseRandomPrime`
Cryptix JCE	20050328	`BigInteger`	`probablePrime`
cryptlib	3.4.3	`context/kg_prime.c`	`generatePrime`
Crypto++	5.6.3	`integer.cpp nbtheory.cpp`	`GenerateRandom FirstPrime`
FlexiProvider	1.7p7	`FlexiBigInt`	`constructor`
GNU Crypto	2.0.1	`Prime`	`isProbablePrime`
libgcrypt	1.6.5	`cipher/primegen.c`	`_gcry_generate_secret_prime`
libgcrypt	1.6.5 FIPS	`cipher/primegen.c`	`_gcry_derive_x931_prime`
LibTomCrypt	1.17	`src/math/rand_prime.c (TomsFastMath)`	`rand_prime`
mbedTLS	2.2.1	`library/bignum.c`	`mbedtls_mpi_gen_prime`
Nettle	3.2	`bignum-random-prime.c`	`nettle_random_prime`
OpenSSL	1.0.2g	`crypto/bn/bn_prime.c`	`BN_generate_prime_ex`
OpenSSL	2.0.12 FIPS	`crypto/bn/bn_x931p.c`	`BN_X931_generate_prime_ex`
PGP Desktop	10.0.1	`libs2/pgpsdk/priv/crypto/bignum/bnprime.c`	`bnPrimeGen`
PGP Desktop	10.0.1 FIPS	`libs2/pgpsdk/priv/crypto/bignum/bnprime.c`	`bnPrimeGenStrongFIPS`
PGP	6.5.8	`libs/pgpcdk/priv/crypto/bignum/bnprime.c`	`bnPrimeGen`
SunRsaSign	JDK 8u66	`BigInteger`	`probablePrime`
WolfSSL	3.9.0	`wolfcrypt/src/integer.c`	`mp_rand_prime`

Table D.2: *The methods responsible for generating RSA primes and their location.*

E Data attachment

- `c_rsa`
  Scripts for building `C` and `C++` cryptographic libraries from source codes and generating RSA keys (Linux only).

- `java_rsa`
  Source codes for generating RSA keys from Java cryptographic libraries.

- `ms_rsa`
  Source codes for generating RSA keys with Microsoft cryptography providers `CryptoAPI`, `CNG` and `.NET` (Windows only).

- `rsa_statistics`
  Java tool for generating statistics from data.

- `figures`
  R scripts for generating figures – heatmaps and modulus histograms from statistics (`rsa-heatmaps`), example notions for RSA integers (`notions`) and example time distributions (`time-explanation`).

- `rsa_data`
  Statistics and figures generated from 40 million RSA keys from 20 different sources.

- `pgp_converter`
  Java tool for converting output of PGP library to CSV format.

- `rsa_distribution`
  Java tool for generating keys with parametrized incremental search and random search. It records the number of primality tests and distance from random numbers to primes.

- `rsa_distribution_data`
  Statistics for keys generated by random sampling and with incremental search with different size of the sieve.