Masaryk University
Faculty of Informatics

The statistical properties of ECC keys generated from software libraries

Master Thesis

Bathini Srinivas Goud

Brno, December 2016

Masaryk University
Faculty of Informatics

The statistical properties of ECC keys generated from software libraries

Master Thesis

Bathini Srinivas Goud

Brno, December 2016

Replace this page with a copy of the official signed thesis assignment and the copy of the Statement of an Author.

Declaration

Hereby I declare that this thesis 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.

Bathini Srinivas Goud

Advisor: RNDr.Petr Svenda Ph.D.

Acknowledgement

I would like to express my sincere thanks of gratitude to my thesis supervisor RNDr. Petr Švenda, Ph.D. for his guidance, valuable advices and feedback.

I would also like to express my sincere thanks of gratitude to prof. RNDr. Vaclav Matyas, M.Sc., Ph.D. for his continuous support.

Abstract

In this thesis the properties of ECC keys and key generation process adopted by various cryptographic libraries are examined with emphasis on the variance of key pair generation, the possibility of detecting source library based on a given public key is discussed. Also the supported elliptic curve domain parameters from various standards are described. Additionally the properties of ECC keys extracted from smart cards also examined for the variance in key pair generation, also demonstrated guessing a probable standard domain parameters used in a black-box implementation.

Keywords

cryptographic library, key generation, Smart cad, ECC, domain parameters, key validation

Contents

1 Introduction

Elliptic curve cryptography (ECC) is a public key cryptosystem just like RSA. In recent days ECC gaining its popularity due to its smaller key sizes as compared to RSA and DSA, which results in performance and bandwidth advantages.

As like in other public key cryptosystems, in ECC every user has a public key and private key. The public key is used for encryption or signature verification and the private key for decryption or signature generation.

Cryptographic libraries are used by individuals for secure online transactions, communicating via secure email and audio/video calling. As a result, there are a number of cryptographic libraries [1] that have been developed, which implements ECC as one of the public key cryptosystems for encryption, signature, and key agreement.

If ECC is selected for a secure email, a compromise of private key leads to compromise of confidentiality of the communication. So the selection of proper domain parameters and the process of key generation are utmost important for the security of the system.

In this work we surveyed widely used cryptographic libraries and described the domain parameters used in practice. While most of the libraries adhere to the standards like SECG [2], ANSI [3], Brainpool [4], NIST [5], some libraries like NaCl [6] / libsodium [7] adhere to custom domain parameters.

We also studied the ECC keys generated from cryptographic libraries for a given domain parameters with emphasis on the variance of their key pair generation and discussed possibility of a source library detection based on a given public key.

We also extended our analysis to EC keys generated from some of the widely used smart cards with the emphasis on variance of their key pair generation.

2 Fundamentals of elliptic curve cryptography

2.1 Introduction

Elliptic curve cryptography (ECC) is a public key cryptography method, in which each user taking part in secure communication requires a pair of keys, a public key, private key, and set of mathematical operations. The public key is being distributed to all the users who are taking part in communication but the private key is only known by the particular user.

The reason to propose elliptic curve cryptosystem as an alternative to RSA and DSA is, it offers the following advantages:

i ECC provides the same level of security with smaller key size (for example 160-bit security of ECC is equivalent to the 1024-bit security of RSA).

ii The algorithms for solving the underlying mathematical hard problem of ECC (the elliptic curve discrete logarithm problem) takes exponential time compared to RSA (the integer factorization) and DSA (the discrete logarithm problem) which takes sub-exponential time.

iii Faster key and signature generation.

iv Reduction in processing power, reduction in storage space, bandwidth, electrical power.

Because of these advantages, the ECC is well suitable for constrained environments like smart cards.

2.2 Fundamental concepts of elliptic curves

An elliptic curve is a special type of polynomial in two variable and its coefficients are from the underlying field. The underlying fields used for cryptography are finite fields. The elliptic curves are defined over prime finite field F_p or a characteristic 2 finite field F_{2^m}.

2.2.1 Elliptic curve over F_p

The elliptic curve over field F_p [8], where p is an odd prime and p>3, is the set of solutions P = (x, y) for x, y $\in F_p$ together with an imaginary point at infinity O to the equation:

$$y^2 = x^3 + ax + b \ mod\ p \tag{1}$$

where a,b $\in F_p$ and $4a^3 + 27b^2 \neq 0 \ \ mod\ p$.

The equation (1) is called a Weierstrass equation, the condition $4a^3 + 27b^2 \neq 0 \ \ mod\ p$ implies that the curve has no singular points. The elliptic curve over F_p is denoted by E(F_p).

Group operations on elliptic curves over F_p

The group operation denoted by '+' can be defined as addition operation on the set E(F_p) such that (E(F_p), +) form an abelian group with identity element as O (point at infinity).

The addition operation in E(F_p) [9] is described below:

- P+O = O+P = P for all P $\in$ to E(F_p).
- If P = (x, y) $\in$ E(F_p), then (x, y) + (x, -y) = O
  (the point (x, -y) $\in$ E(F_p) is denoted -P, called the negative of P).
- Let P = (x_1, y_1) $\in$ E(F_p) and Q = (x_2, y_2) $\in$ E(F_p), where P $\neq \pm Q$, then P+Q = (x_3, y_3), where
  $x_3 = \lambda^2$-x_1-x_2, $y_3 = \lambda(x_1$-$x_3)$-y_1, and $\lambda = \frac{y_2 - y_1}{x_2 - x_1}$.
- Let P = (x_1,y_1) $\in$ E(F_p) , then P + P = 2P = (x_3, y_3),
  where $x_3 = \lambda^2$-$2x_1$, $y_3 = \lambda(x_1$-$x_3)$-y_1 and $\lambda = \frac{3x_1^2 + a}{2y_1}$. This operation is called point doubling.

2.2.2 Elliptic curve over F_{2^m}

The elliptic curve E(F_{2^m}) over F_{2^m} is the set of solutions P = (x, y) for x, y $\in F_{2^m}$ together with an imaginary point at infinity O to the equation:

$$y^2 + xy = x^3 + ax^2 + b \tag{2}$$

where a , b $\in F_{2^m}$ and b $\neq 0$

Group operations on elliptic curves over F_{2^m}

The addition operation in E(F_{2^m}) [9] is described below:

- P+O =O+P = P for all P $\in$ E(F_{2^m}).
- If P = (x, y) $\in$ E(F_{2^m}),then (x, y) + (x, -y) = O
  (the point (x, -y) $\in$ E(F_{2^m}) is denoted -P and is called the negative of P).
- if P = (x_1, y_1) $\in$ E(F_{2^m}) and Q = (x_2, y_2) $\in$ E(F_{2^m}), where P $\neq \pm Q$, then P+Q = (x_3, y_3), where
  $x_3 = \lambda^2+\lambda+x_1+x_2+$a, $y_3 = \lambda(x_1+x_3)+x_3+y_1$ and $\lambda = \frac{y_2+y_1}{x_2+x_1}$.
- Let P = (x_1, y_1) $\in$ E(F_{2^m}), then P+P=2P = (x_3, y_3), where
  $x_3 = \lambda^2+\lambda+$a, $y_3 = \lambda(x_1+x_3)+x_3+y_1$ and $\lambda = x_1+ \frac{x_1}{y_1}$.

2.2.3 Elliptic curve cryptosystem

Elliptic curves are used as an extension to other currently existing cryptosystems like Diffie-Hellman key exchange (DH), Digital Signature Algorithm (DSA).

The Elliptic curve cryptosystems include the Elliptic Curve Diffie-Hellman key exchange (ECDH), the Elliptic Curve Digital Signature Algorithm [10] (ECDSA), the Elliptic Curve Authenticated Encryption Scheme (ECAES) [11] (this is a variant of the ElGamal public-key encryption scheme). The security of ECC is based on the computational in-feasibility of solving elliptic curve discrete log problem (ECDLP).

2.2.4 ECC domain parameters

In order to use elliptic curve cryptography, the users must agree on all the elements defining the elliptic curve which are called domain parameters. The ECC domain parameters [9] are given by (q,FR,a,b,G,r,h), where q specifies a prime power (q=p for prime field or q=2^m for binary field). FR (field representation) is an indication of the method to represent the field elements in F_q. The parameter G is the base point called generator, by which the cyclic sub group is defined. r is the order of base point (the smallest positive integer such that rG = O)

and h is an integer called cofactor and is given by h = $\frac{\#E(F_p)}{r}$ or $\frac{\#E(F_{2m})}{r}$. The value of h should be less than 4 and preferably the value of h is 1 for efficiency reason[5] and a, b are the coefficients of the equation (1) and (2).

The primary security of ECC depends on the selection of order r which depicts the length of ECC key and also determines the difficulty of solving elliptic curve discrete-log problem. In general, the value of r should be a large prime.

2.2.5 ECC key generation mechanism

To use elliptic curve cryptosystem, the users need to generate both private and public key. This key pair is associated with a set of elliptic curve domain parameters as described above.

The mechanism for generating the private and public key by an entity-A is given below:

- Generate a random number d in the interval [1, r-1] where r is the order.
- Compute public key Q = dG.
- The entity-A's public key is Q and private key is d.

The key pair (d, Q) is used for cryptographic operations.

Since public key Q is a point on the curve with coordinates (x, y), for a cryptographic operation entity-A need to send this ordered pair to entity-B. The point compression technique [12] (a method where it requires only x coordinate and one-bit information of y coordinate to send a point) is adopted to save the bandwidth. entity-B can compute the corresponding y coordinate using x and one bit of y.

2.2.6 ECC public keys validation

The public key is a point on the elliptic curve, validation of public key Q ensures that Q is the point of order r in E(F_q). The purpose of public key validation is to prevent malicious insertion of an invalid public key to enable small subgroup attacks[13].

A public key Q = (x,y) is validated using the following procedure [9] :

i Verify that Q is not the point at infinity (Q != O).

ii Verify that x and y are elements in the field F_q, (i.e.verify that x and y are integers in the interval [0, p-1] in the case that q = p, or that x and y are bit strings of length m bits in the case that q = 2^m).

iii If q = p is an odd prime, verify that $y^2 = x^3 + ax + b \ \ mod\ p$. If q = 2^m, verify that $y^2 + xy = x^3 + ax^2 + b$ in F_2^m.

iv Verify that rQ = O.

If any one of the first three conditions is violated, then Q is invalid. If condition `iv` alone violated then the validation is called partial key validation.

The condition `iv` can be verified much faster than expensive point multiplication rQ [14]. For example if h = 1 then condition 4 is implied by the other three conditions and in some protocols the check rQ = O may be replaced by the check hQ != O this guarantees that Q is not in a small subgroup of E(F_q) of order dividing h.

2.3 Attacks against elliptic curve cryptosystem

The security of ECC is based on elliptic curve discrete logarithm problem [9] (ECDLP).

Given a curve E(F_q), a point P∈E(F_q) of order r and Q $\in$ E(F_q), finding the integer k (where 0 < k $\leq$ r-1) such that

$$Q = kP \qquad (3)$$

The parameter k often is denoted as the elliptic curve discrete logarithm of Q for given P (k = $\log_P$(Q)).

2.3.1 Known algorithms to solve the ECDLP

The well-known attacks on ECC have exponential complexity. The solution k of equation (3) i.e. the ECDLP can be computed by the following techniques [15].

Exhaustive search

This method adds the point P to itself (P+2P+3P……) till the sum equals to Q, which yields the solution to k with kP = Q. In the worst case scenario, this computation goes up to r-1 steps. This makes the attack infeasible in practice for a large value of r.

Baby Step Giant Step(BSGS) algorithm

This algorithm is based on a space-time trade off. It works for every cyclic group. For a given order r, it requires $\sqrt{r}$ memory space and $\sqrt{r}$ computations. It is possible to use less memory by choosing a smaller size of r in the first step of the algorithm, doing so the running time will be increased. Usually, this algorithm is used for the groups whose order is prime. If the order of the group is composite then the Pohlig-Hellman algorithm is used [16]. Due to requirement of high memory complexity BSGS is not so attracted to solve the ECDLP.

Pollard's Rho algorithm

The most efficient general algorithm is Pollard-Rho method, this was proposed by J.Pollard in 1978 [17]. It works by defining a pseudo-random sequence and then detecting a match in the sequence. The modifications to the algorithm by Gallant, Lambert and Vanstone [18] and Wiener and Zuccherato [19] described that this algorithm requires $\sqrt{\frac{\pi r}{2}}$ elliptic group operations. Van Oorschot and Wiener [20] described that Pollard rho method can be parallelized, and the expected running time using n processors is roughly $\sqrt{\frac{\pi r}{2n}}$ group operations. This is also an exponential run time in r.

2.4 Elliptic curve crypto primitives

The primary underlying mathematical operation in ECC is elliptic curve scalar multiplication (analogous to exponentiation in multiplicative groups). Given a point G and an integer d, the scalar multiplication dG is the result of adding G to itself d times.

The crypto primitives of ECC are explained with the following protocols.

2.4.1 Elliptic Curve Diffie-Hellman (ECDH) key exchange

Entity-A and entity-B wants to agree on a shared key, the both entities compute their public and private key as follows:

- Private key of entity-A = a.
- Public key of entity-A = aG (where G is the base point of the selected curve).
- Private key of Entity-B = b.
- Public key of Entity-B = bG.

  Both entities send each other their public keys.
  Both compute the product of their private key and others public key.
- Entity-A's shared Key K_{AB} = a(bG)
- Entity-B's shared key K_{BA} = b(aG)
- The shared secret key = K_{AB} = K_{BA} = abG.

2.4.2 EC encryption scheme analogous to ElGamal

For encryption [21], both entities A & B will agree on common domain parameters, both creates their key pair.

- Entity-A's private key = a.
- Public key P_A = aG.
- Entity-B's private key = b.
- Public key P_B = bG.

Entity-A takes plain-text message M, and encodes it on to a point P_M from the elliptic group. Entity-A chooses another random integer k from the interval [1, r-1].

The computed cipher-text is a pair of points as shown below:

- $P_C = ((kG), (P_M + kP_B))$

To decrypt, entity-B computes the product of the first point of P_C and his private key b i.e.

b(kG) and then takes this product and subtracts it from the second point of P_C

- $(P_M + kP_B) - [b(kG)] = P_M + k(bG) - b(kG) = P_M$

Entity-B then decodes P_M to get the message M.

3 Recommended elliptic curves by standards

The cryptographic standards are important because of the following reasons:

i To facilitate widespread use of cryptographically well-specified systems.

ii To promote interoperability between various implementations.

This section describes a brief details about the recommended elliptic curves and domain parameters by various standards.

3.1 National Institute of Standards and Technology (NIST)

NIST recommends [5] the following type of curves:

i. Pseudo random curves over F_p (denoted by P-X) where X is the key length and defined by equation:

$$E : y^2 = x^3 - 3x + b \; mod \; p \tag{4}$$

for efficiency reason a = -3 is considered.

ii. A pseudo random curve over F_{2^m} (denoted by B-X) where X is the key length and defined by equation:

$$E : y^2 + xy = x^3 + x^2 + b \tag{5}$$

iii. A special curve over F_{2^m} called a Koblitz curve (denoted by K-X) where X is the key length and defined by equation:

$$E_a : y^2 + xy = x^3 + ax^2 + 1 \tag{6}$$

The recommended parameters for above curves are explained in [5]. The Table 3.1 summarizes the recommended curves.

S.No	Curve over prime field F_P	Curve over F_{2^m}	Koblitz Curve
1	P-192	B-163	K-163
2	P-224	B-223	K-223
3	P-256	B-283	K-283
4	P-384	B-409	K-409
5	P-521	B-571	K-571

Table 3.1: *List of NIST recommended elliptic curves [5]*

3.2 Brainpool elliptic curves

The Brainpool curves are defined over prime field F_p, the curves equations and base fields are selected to allow for efficient arithmetic. The difference between Brainpool curves and NIST curves is, Brainpool curves uses random primes and NIST curves use quasi-Mersenne primes[22]. The reason for Brainpool to use random primes is to avoid potential security issues with non-random primes and also to avoid possible patent issues with fast reduction algorithms. The recommended domain parameters are specified in [4]. The Table 3.2 summarizes the recommended curves.

S.No	Curve over prime field F_P	Twisted curve over F_P
1	brainpoolP160r1	brainpoolP160t1
2	brainpoolP192r1	brainpoolP192t1
3	brainpoolP224r1	brainpoolP224t1
4	brainpoolP256r1	brainpoolP256t1
5	brainpoolP320r1	brainpoolP320t1
6	brainpoolP384r1	brainpoolP384t1
7	brainpoolP512r1	brainpoolP512t1

Table 3.2: *List of Brainpool elliptic curves [4]*

3.3 American National Standards Institute (ANSI)

The objective of these standards is to achieve a high degree of security and interoperability.

In ANSI standards, the underlying field is restricted to prime field F_P and Binary field F_{2^m}. The recommended domain parameters are described in [3]. The Table 3.3 summarizes the recommended curves by ANSI X9.62.

S.No	Curve Over prime field F_P	Curve over F_{2^m}
1	prime192v1	c2pnb163v1
2	prime192v2	c2pnb163v2
3	prime192v3	c2pnb163v3
4	prime239v1	c2pnb176w1
5	prime239v2	c2tnb191v1
6	prime239v3	c2tnb191v2
7	prime256v1	c2tnb191v3
8	—	c2pnb208w1
9	—	c2tnb239v1
9	—	c2tnb239v2
9	—	c2tnb239v3
9	—	c2pnb272w1
9	—	c2pnb304w1
9	—	c2tnb359v1
9	—	c2pnb368w1
9	—	c2tnb431r1

Table 3.3: *List of ANSI X9.62 recommended elliptic curves [3]*

3.4 Standards for Efficient Cryptography Group(SECG) recommended curves

The SECG recommends curves [2] over

i The prime field F_p (both pseudorandom curve denoted by 'r' and Koblitz curve denoted by 'k').

ii Over the field F_{2^m} (both pseudorandom curve 'r' and Koblitz curve 'k').

The recommended domain parameters for the curves over the both fields are specified in [2]. The Table 3.4 summarizes the recommended curves over prime field F_p and F_{2^m}.

3.5 Other curves

Apart from the curves recommended by the standards, the following curves are also used in most of the cryptographic libraries.

3.5.1 Edwards Curve

Edwards introduced a normal form for elliptic curves over a finite field p, these curves are defined by the equation [23]:

$$x^2 + y^2 = c^2 + c^2x^2y^2 \tag{8}$$

Bernstein and Lange [24] showed that it is possible to obtain more curves by taking the following equations:

$$x^2 + y^2 = c^2(1 + d^2x^2y^2) \tag{9}$$

where c, d $\in$ p with cd(1-c^4d) $\neq$ 0, and in fact these curves are isomorphic to curves of the form

$$x^2 + y^2 = 1 + dx^2y^2 \tag{10}$$

for some scalar d $\in$ p {0, 1}.

Edwards curve is birationally equivalent to an elliptic curve in Weierstrass form [25].

S.No	Curve over prime field F_P	Curve over F_{2^m}
1	secp112r1	sect113r1
2	secp112r2	sect113r2
3	secp128r1	sect131r1
4	secp128r2	sect131r2
5	secp160k1	sect163k1
6	secp160r1	sect163r1
7	secp160r2	sect163r2
8	secp192k1	sect193r1
9	secp192r1	sect193r2
10	secp224k1	sect233k1
11	secp224r1	sect233r1
12	secp256k1	sect239k1
13	secp256r1	sect283k1
14	secp384r1	sect283r1
15	secp521r1	sect409k1
16	–	sect409r1
17	–	sect571k1
18	–	sect571r1

Table 3.4: *List of SECG recommended elliptic curves [2]*

Group operation on Edwards curve

Let P = (x_1, y_1) is a point and Q = (x_2, y_2) another point, addition of points P+Q is given by

- $(x_1, y_1) + (x_2, y_2) = \left(\frac{x_1y_2+x_2y_1}{1+dx_1x_2y_1y_2}, \frac{y_1y_2-x_1x_2}{1-dx_1x_2y_1y_2}\right)$.

The point doubling is given by

- $(x_1, y_1) + (x_1, y_1) = 2(x_1, y_1) = \left(\frac{2x_1y_1}{1+dx_1^2y_1^2}, \frac{y_1^2-x_1^2}{1-dx_1^2y_1^2}\right)$.

3.5.2 Twisted Edwards Curves

The equation (10) is generalized to Twisted Edwards form [23] defined by

$$E_{E,a,d} : ax^2 + y^2 = (1 + d^2x^2y^2) \qquad (11)$$

where a,d $\in$ p with ad(a-d)$\neq$ 0. The Edwards curves are a special case of Twisted Edwards curve where a is rescaled to 1.

Group operation on Twisted Edwards curve

Point addition

- $(x_1, y_1) + (x_2, y_2) = \left(\frac{x_1y_2+x_2y_1}{1+dx_1x_2y_1y_2}, \frac{y_1y_2-ax_1x_2}{1-dx_1x_2y_1y_2}\right) = (x_3, y_3)$.

The point doubling independent of d is given by

- $(x_1, y_1) + (x_1, y_1) = 2(x_1, y_1) = \left(\frac{2x_1y_1}{ax_1^2+y_1^2}, \frac{y_1^2-ax_1^2}{2-ax_1^2-y_1^2}\right)$.

The above point addition law is complete if d is a non-square in p so that $1 + dx_1x_2y_1y_2 \neq 0$ and $1 - dx_1x_2y_1y_2 \neq 0$. It works for all pair of inputs [26] and eliminates the subgroup and invalid point attacks.

Twisted Edwards curves are the heart of the digital signature scheme EdDSA which is designed to be faster than existing schemes without sacrificing security.

3.5.3 Curve25519

This is a Montgomery curve over the prime field F_P and is one of the fastest ECC curve [27] offering 128 bits of security and is designed to use with Elliptic Curve Diffe-Hellman (ECDH) key agreement scheme. The Curve25519 is defined by the equation:

$$y^2 = x^3 + 486662x^2 + x \qquad (11)$$

The prime number p is given by $2^{255} - 19$, and the base point is given by x = 9. The protocols using this curve uses only x coordinate (compressed point). This curve is birationally equivalent to Edwards curve.

Curve 25519 is constructed to avoid many potential implementation pitfalls, it avoids many side channel attacks [25] and issues with poor quality random number generators.

4 Classification of public keys

This work is inspired from the paper [28], the classification of public keys is based on the study of statistical properties of public key bytes.

4.1 Experimental methodology

From the ECC key generation algorithm described in section 2.2.5, it is clear that a private key is a random number in the interval [1, r-1] and the public key is a field element from the set (x, y) of the given elliptic curve field. The elliptic curve field operation is only defined between two points in the same field. It is an error to apply the field operations to two elements that are from different fields.

A public key is said to be valid only when it satisfies the conditions described in 2.2.6. So the valid public key generated from a cryptographic library should be a random element of the given elliptic curve field. For a given field size (size of the modulus) and a random private key, there should be a unique public key within the given field, the size of public key should be the size of the field. If the generated key size is smaller than the field size, the key is padded with zeros to field size. To classify the public keys, in this work we inspected the generated keys from various cryptographic libraries with the focus on following:

i Whether the generated key size is of order of field size or not.

ii Whether the generated keys are random or any significant bias exist.

iii Whether the generated keys from a particular library are valid or not.

We adopted the following methodology:

i Analysis of the distribution of most significant byte (MSB) of the private and public keys.

ii Analysis of the frequency of ones and zeros.

iii Inspection of ECC implementation adopted by various libraries for point validation.

With help of this analysis, we discussed the possibility to distinguish source library based on a given public key.

4.2 List of cryptographic libraries considered for inspection

The following libraries are considered for inspection and classification of ECC public keys:

i OpenSSL 1.0.1f

ii Libgcrypt 1.7.2

iii Crypto++ 5.6.3

iv Bouncy Castle 1.55

v libsodium 1.0.11

vi mbed TLS 2.3.0

vii Nettle 3.2

viii Microsoft CNG (Cryptography API: Next Generation)

Most significant byte of the keys

The reason for analyzing the most significant byte of the generated keys is to observe whether the size of generated keys is of the size of the underlying field or padded with zero. If a zero is padded to the key, obviously the most significant byte of the key will be affected. We have generated one million keys from each cryptographic library for a given domain parameters and studied the frequency distribution of the most significant byte.

Figure 4.1 and 4.2 contains an example graphs which illustrates the distribution of MSBs of one million private and public keys generated from libsodium library with curve *Curve25519*.

In the figures, the horizontal axis shows the value of MSB and vertical axis shows the frequency of the MSB values. The more examples are given at Appendix –A. From the figures 4.1 and 4.2, it is observed that the MSB distribution is not uniform. The reason for this behaviour explained in section 4.6.

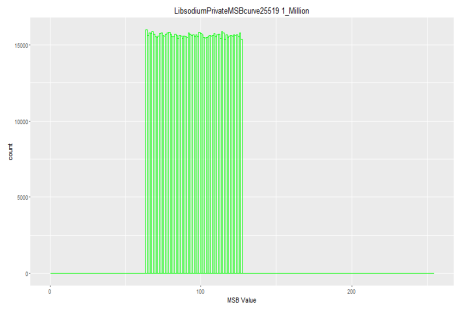


Figure 4.1: Private key MSB distribution

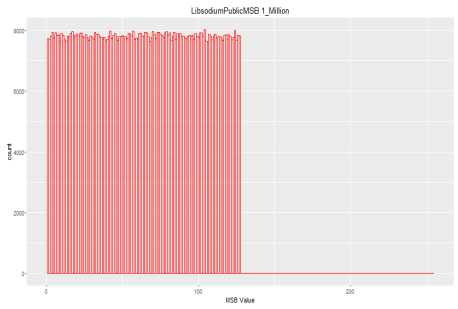


Figure 4.2: Public key MSB distribution

Analysis of frequency of ones and zeros of private and public keys

The purpose of analyzing the frequency of 1's and 0's is to determine whether the number of ones and zeros in a sequence are approximately same as would be expected for a truly random sequence of bits [29] & [30] because EC keys are random numbers in the given field.

Figure 4.3 contains an example graph which illustrates the distribution of frequency of ones and zeros of one million public keys generated from libsodium library with selected curve *Curve25519*.

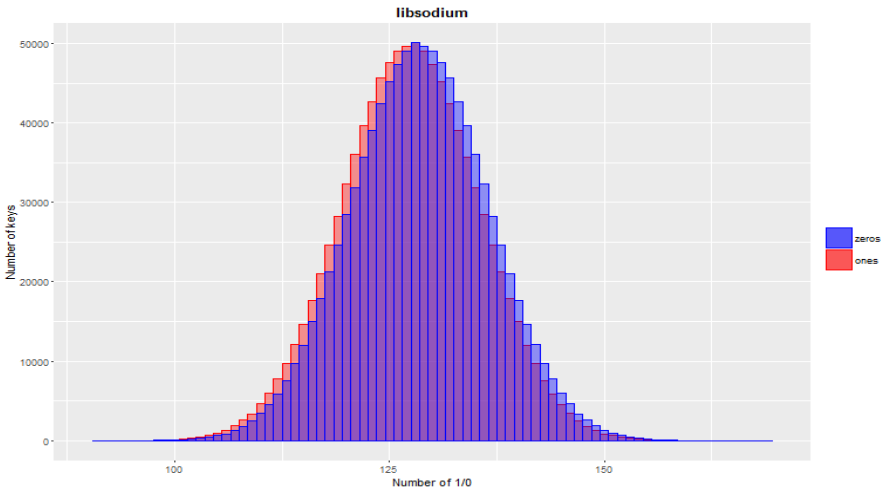


Figure 4.3: *Frequency distribution of public key ones and zeros*

In the figure, the horizontal axis shows the count value of ones/zeros and vertical axis shows number of keys for a given count value.

The more examples are given at Appendix–B (for public keys) and Appendix-C (for private keys).

As per the definition given in [30], for a given random sequence the number of ones and number of zeros approximately the same and should follow the Gaussian distribution. The figure shows a superimposed distribution of frequency of ones and zeros, from this it is observed that there is an imbalance between the number of ones and zeros of the public keys generated from this library. The reason for the same explained in section 4.6.

With help of this analysis, we can able to detect the variance in key pair if any bias is observed from the expected distribution.

4.3 Detailed analysis of cryptographic libraries

The libraries specified in section 4.2 are analyzed in detail with the methodology described in section 4.1 with emphasis on variance in key pair generation.

Since the private key is a random number within the interval [0, r-1], we looked at the pseudo random number generator (PRNG) used by a particular library, and also inspected the source code of each library to find the checking mechanism used for key validation. In the MSB analysis, for the curves in Weierstrass form, we considered MSB of X coordinate of the public key. The source files and subsequent methods for key generation and key validation are listed at Appendix-H & I.

4.4 OpenSSL version1.0.1f

The OpenSSL [31] supported elliptic curve standards are specified in Table 4.1. No default elliptic curve parameters are set for this library, we considered *secp256k1* [2] elliptic curve parameters for our analysis.

Brief description of pseudo-random number generator

The OpenSSL by default uses a software pseudo-random number generator [31] implementation. This PRNG must be seeded with un-

predictable data. OpenSSL makes sure that the PRNG state is unique for each thread. For the systems that support */dev/urandom*, the randomness device is used and on other systems the application takes care of seeding the PRNG (for example calling function RAND_add() which mixes the suitable input comes from user interaction such as random key press, mouse movement and certain hardware events into the PRNG state).

Analysis of MSB

One million keys generated on system with Linux OS, which uses */dev/urandom* as seeding device for PRNG. The histograms of most significant bytes of private key and public key are plotted (the results are enclosed in Appendix-A). As seen from the histograms, the MSB value distributed uniformly over all 256 values, leaving no bias towards particular bits.

From this we inferred that, the size of generated keys from OpenSSL are of the order of field size (for secp256k1 the field size is 256-bits) with no zero padding.

Analysis of frequency of ones and zeros

As described above, for a given key size of 256 bits, the sequence to be random, the number of ones and zeros are expected to be 128 and should follow the Gaussian distribution. As seen from the histograms of ones and zeros of both private and public keys for a given one million keys, it follows the distribution as expected from randomly generated keys with mean value 128.

From this, we inferred that that the generated keys may be random. We observed no significant bias.

Key validation

Private key: The private key is a random number in the interval [1, r-1], OpenSSL ensures the generated random number within this interval.
Public key: OpenSSL checks the following conditions for the validation of public key.

i Checks for whether given point is at infinity.

ii Check for whether the public key is on the elliptic curve.

iii Check for whether the product of public key and order is the point at infinity (rQ = O)

iv In case the private key is present, check if the product of generator and private key is equal to public key (dG = Q).

Table 4.2 summarizes the list of checks performed by OpenSSL. The source file and method responsible for key validation are given at Appendix-I.

4.5 Libgcrypt version 1.7.2

Libgcrypt 1.7.2 [32] supports regular mode and FIPS 140-2 mode. The supported elliptic curve parameters of this library are specified in Table 4.1. We used this library in regular mode to examine the ECC keys. This library also doesn't set any default curve parameters, we have selected NIST-P256 [5] curve parameters for our analysis.

Brief description of PRNG

Libgcrypt [32] uses two different type random generators CSPRNG (continuous seeded pseudo-random number generator) [33] and a FIPS approved ANSI X9.31 PRNG [34] using AES with a 128 bit key. CSPRNG is used by default if the Libgcrypt is not in FIPS mode otherwise X9.31 PRNG is used.

Libgcrypt provides 3 levels of random quality such as very strong random, strong random and weak random. Usually, very strong random is used for key generation.

Both PRNGs [32] make use of the following entropy gathering modules :

rndlinux : Uses */dev/random* and */dev/urandom* devices.
rndw32 : This is used on Microsoft windows operating system, it uses certain properties of the system.

The CSPRNG alone uses following entropy modules:

rndunix : Collects entropy from virtual machine and process statistics with several operating system commands.
rndegd : Uses EGD (entropy gathering daemon provided by OS) as a system daemon, it keeps on running and does not waste the collected entropy if the application doesn't need.
rndhw : This is an extra module to collect additional entropy by using hardware random number generator.

Analysis of MSB

One million keys are generated on the system with Linux OS. We modified the source code to use */dev/urandom* instead of */dev/random* for seeding the PRNG to generate one million keys in a reasonable time. However, it is not advisable to use */dev/urandom* for the cryptographic purpose.

We plotted the histogram of MSBs of keys to observe the behavior. The results (Appendix-A) shows that the MSB value is uniformly distributed over all the 256 values, and no significant bias towards particular bits is observed. From this we inferred that the size of keys generated by this library also of the order of field size.

Analysis of frequency of ones and zeros

As seen from the histograms distribution of ones and zeros of both private and public keys (Appendix-B, C) for a given one million keys, it follows the distribution as expected from randomly generated keys with mean value 128.

We noticed no significant bias, so we inferred that the keys generated from Libgcrypt 1.7.2 may be random and detected no variance in keys.

Key validation

From the source code inspection, we inferred that the Libgcrypt ensures the key validation by performing the checks as described in section 2.2.6. The Table 4.2 summarizes list of checks that Libgcrypt performs. The source file and the method responsible for key validation are given at Appendix-I.

4.6 libsodium version 1.0.11

libsodium [7] is a portable and packageable fork of Nacl [6]. Nacl is a new easy to use high-speed library for network communication, encryption and decryption. Nacl advances the other cryptographic libraries by improving the security, usability, and speed. Many of the algorithms and implementations used in Nacl were developed as part of Daniel J. Bernstein's High-Speed Cryptography project funded by the U.S. National Science Foundation[6].

The libsodium version 1.0.11 implements a variant of digital signature scheme EdDSA called Ed25519 proposed and implemented by Daniel J. Bernstein and his team [25] where the elliptic curves *Twisted Edwards curve* and *Curve25519* are used.

Brief description of PRNG

The libsodium [7] provides following set of functions to generate random keys;

i On Windows systems, the *RtlGenRandom()* function is used, this function generates random data as specified in FIPS 186-2 appendix 3.1 [35] with SHA-1 as the G function, and with entropy from the current process ID, the current thread ID, the tick count since boot time, the current time.

ii On OpenBSD and Bitrig, the *arc4random()* function is used. This function uses arc4 stream cipher, the seed is provided from */dev /urandom/* device.

iii On Linux kernels, from libsodium 1.0.3 onward the *getrandom()* system call is used. The *getrandom()* draws entropy from the */dev/urandom/* device.

iv On other Unices, the */dev/urandom* device is used.

There is a provision to hook custom implementation if none of the above options used safely.

Analysis of MSB

The libsodium supports elliptic curves *Twisted Edwards curve* and *Curve25519*. We examined keys generated from both the curves. The analysis description is given below:

Twisted Edwards curve

As described above, many of the algorithms and implementations used in this library were developed by Daniel J. Bernstein and his team with a goal to achieve high speed and security.

As part of that the author proposed the fastest implementation of Twisted Edwards curve operations on the given prime field [36]. The same is implemented in libsodium.

The field size of *Twisted Edwards curve* is 256-bits, the private key is a 32-byte random number generated by the PRNG. The public key is generated by multiplying private scalar '*a*' (described in section *Curve25519* key generation) with the base point, which yields a 256-bit group element. We have examined one million keys generated by libsodium on system with Linux OS (by default */dev/urandom* device is used to seed PRNG).

Figure 4.4 and 4.5 shows the histogram plots of MSB value of one million keys generated from libsodium with *Twisted Edwards curve*. The results shows that the MSB value uniformly distributed over all 256 values. From this we inferred that the size of keys is of the order of field size and we observed no significant bias.

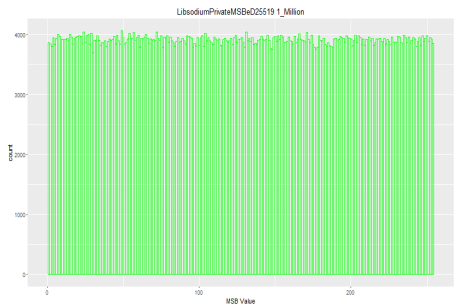


Figure 4.4: Private key MSB distribution

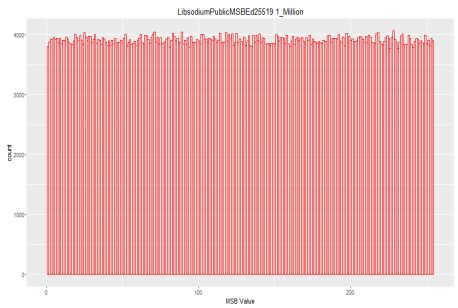


Figure 4.5: Public key MSB distribution

Curve25519

Daniel J. Bernstein defined a new curve [27] with a goal to achieve high speed and should be resistant to side channel attacks. The size of the private and public key of *Curve25519* is 32-byte, and it offers 128-bit security.

The key generation mechanism for *Curve25519* is quite different from other curves, the following sections explain about private and public key generation:

Private key generation

The specifications of private key defined by author is:

$\{n : n \in 2^{254} + 8\{0, 1, 2, 3, ..., 2^{251} - 1\}$

The key generation mechanism implemented in libsodium is described below:

A 32 byte seed (a uniform random binary number) is hashed using SHA512, which gives 64 byte value, this value then split into left half (first 32 bytes) and right half. The left half is altered into a *Curve25519* private scalar '*a*' by clearing the bits 0, 1, 2 of the first byte (LSB) and clearing the bit 7 of the last byte (MSB) and by setting the bit 6 of the last byte to 1.

Setting the lowest bits to zero makes the group order is multiple of 8 and sets the private scalar to a multiple of 8. This ensures that the points are in the prime order subgroup without small subgroup interfering.

With this *Curve25519* solves the problem of subgroup attacks by making every secret scalar (private key) a multiple of eight.

Curve25519 also takes advantages of Montgomery ladder [37] for point multiplication and completely avoids timing attacks by setting the 254^{th} bit of private key to 1.

As we know that the *Curve25519* keys are specially crafted to address the side-channel and subgroup attacks, in order to verify the behaviour of keys and to find any variance in key pair generation, we examined one million keys generated on the system with Linux OS (*/dev/urandom* is used as a default seeding device to PRNG).

The figure 4.6 and 4.7 illustrate the behavior of MSB and LSB of ten thousand private keys.

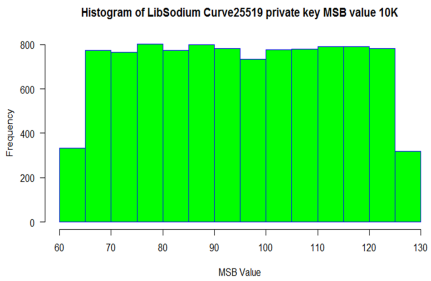


Figure 4.6: Private key MSB distribution

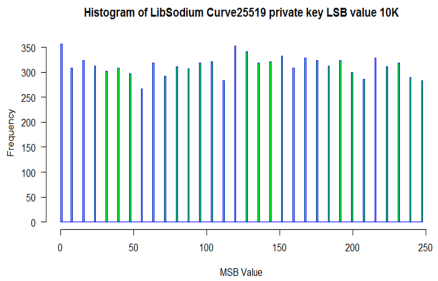


Figure 4.7: Private key LSB distribution

As seen from the results, the LSB value of the private key is always multiple of 8 (since last three bits are zero) and the MSB value floats between 64 and 128 (since 254^{th} bit of the private key is always 1 and 255^{th} bit is 0).

Public key generation

The public key is generated by multiplying this secret scalar '*a*' with the base point (generator) which yields a 256 bit field element and is reduced over modulo $2^{255} - 19$.

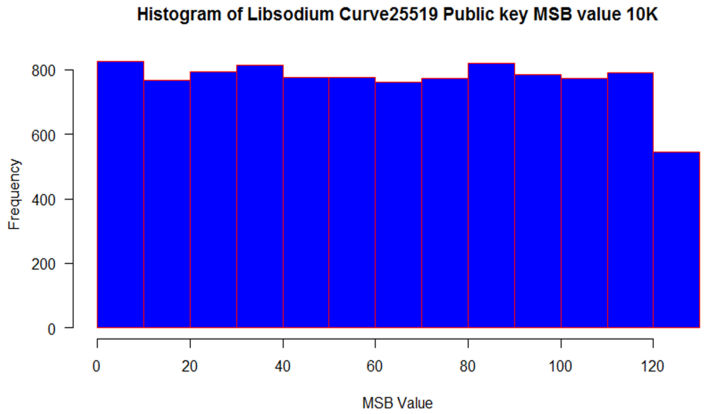


Figure 4.8: *Public key MSB distribution*

The figure 4.8 contains histogram plot of MSB of *Curve25519* ten thousand public keys generated from the libsodium library.

As seen from the plot, it is observed that the MSB value floats between 0 to 128 (since it suggested by the author Daniel J. Bernstein, the 32-byte public key is reduced over modulo 2^{255}-19, which results in the most significant bit of public key is set to zero).

From the results, we inferred that though the private and public key size is 32-bytes, the effective size of private key is 252-bit and the public key is 255-bit.

We also inferred that this may be a significant observation to classify a library based on given private or public key.

Analysis of frequency of ones and zeros

Curve25519

As seen from the histogram distribution (Appendix-B, C) of libsodium for the selected *Curve25519*, there exists significant bias in private key distribution because the private key is specially crafted for the reasons explained in the previous section. In fact, there is a slight bias in the distribution of public keys also, this is because of the most significant bit of public key is always set to zero.

Key validation

As inspected from the source code, there is no key validation mechanism described in 2.2.6 is implemented for both *Twisted Edwards curve* and *Curve25519*.
We hypothesized the following reason :

The author Daniel J. Bernstein claims in the paper [27] that *Curve 25519* computations can avoid checking for point at infinity, and also as explained in above section clearing least significant bits of private key avoids sub group attacks.

The Ed25519 signature scheme implemented using *Twisted Edwards curve* with parameter $d = -\frac{121665}{121666} \in p$ [25]. As discussed in section 3.5.2, if the value of d is non-square in p then the addition law is complete and avoids subgroup and invalid point attacks. So the selection of d in Ed25519 depicts that there may not be additional point validation mechanism is required. This may be the reason for not implementing the key validation in libsodium. Our hypothesis also supported by S. Josefsson [38].

4.7 Crypto++ version 5.6.3

Crypto++ [39] supported elliptic curve standards are given in Table 4.1. No default curve is set for this library also, we selected curve *secp256r1* [2] for our analysis.

Brief description of PRNG

This library doesn't support the generators specified by NIST SP800-90A [40]. The library uses *RandomNumberGenerator* class in which *AutoSeedRandomPool* mechanism is used, where the seed is automatically fed to the generator using underlying operating system's entropy pools. Entropy is retrieved using Crypto++ *OS_GenerateRandomBlock*. On Linux, *OS_GenerateRandomBlock* uses */dev/random* or */dev/urandom* and on windows, it uses *CryptGenRandom* [41].

Analysis of MSB

We examined one million keys generated on system with Linux OS with */dev/urandom* as default seeding device to PRNG. The histogram of the MSBs of both private and public key are plotted (Appendix-A). As seen from the results, the MSB value is uniformly distributed over all the 256 values and no significant bias is observed.

Analysis of frequency of ones and zeros

By observing the histograms of ones and zeros of both private and public keys (Appendix-B, C) for a given one million keys, it is understood that the distribution follows the Gaussian as expected from randomly generated keys with mean value 128.

We noticed no significant bias in the distribution, we inferred that generated keys from the Crypto++ 5.6.3 library seem to be random.

Key validation

The source code of the library inspected for the checks that the library performs to validate EC keys, we inferred that the Crypto++ ensures the key validation by performing the checks as described in section 2.2.6. The Table 4.2 summarizes the list of checks that Crypto++ 5.6.3 performs.

4.8 Bouncy Castle version 1.55

The Bouncy Castle library [42] supported elliptic curve standards are given in Table 4.1.
This library also doesn't set any default curve, we have selected *secp256r1* [2] for our analysis.

Brief description of PRNG

The Bouncy Castle implements the following random number generators [42]:
DigestRandomGenerator **:** Random data generation based on digest of secret seed and counter.
ReversedWindowGenerator: The order of generated random bytes by an underlying random number generator are reversed by a configurable size window.
ThreadedSeedGenerator: Implements a thread entropy collector scheme [43]. It uses two threads one for incrementing a counter in loop and other is to wait 1*ms* until the counter changes. The new value is added to an output array.

Analysis of MSB

We have written our source code in java to generate keys using Bouncy Castle on the system with Windows 10 OS. We used *SecureRandom()* method to generate the random data, which uses *ThreadedSeedGenerator*. It is seeded with entropy from the system by using *CryprGenRandom()*. We examined one million keys generated from Bouncy Castle library. The histogram of the MSBs of both private and public key are plotted (Appendix-A).

The results shows, the MSB value is uniformly distributed over all 256 values, no significant bias is observed from the generated keys.

Analysis of frequency of ones and zeros

As seen from the histograms of ones and zeros of both private and public keys (Appendix-B, C), it is observed that the distribution follows the Gaussian as expected from randomly generated keys with mean value 128. We observed no significant bias in the distribution, we inferred that the generated keys may be random.

Key validation

The Bouncy Castle library Java versions before 1.51 does not implement the key validation [44]. From the inspection of source code, we inferred that the implementation ensures the key validation by performing the checks as described in section 2.2.6 and also takes the advantage of method described in section 2.2.6 suggested by [14] to verify the condition `iv` much faster than expensive point multiplication rQ. The Table 4.2 summarizes list of checks that Bouncy Castle performs. The source file and the method responsible for key validation are given at Appendix-I.

4.9 mbed TLS version 2.3.0

The supported elliptic curve standards by mbed TLS [45] are described in Table 4.1. Since the mbed TLS doesn't have any default curve parameters, we have selected *secp256r1* [5] for our analysis.

Brief description of PRNG

The random number module of mbed TLS uses an algorithm Counter-mode block-cipher based Deterministic Random Bit Generator from NIST SP 800-901 [40]. The underlying algorithm is AES-256 in counter mode. Entropy is gathered using a given entropy callback function which is given at initialization. This callback is called both on initialization and when reseeding is required. The CTR_DRBG can be seeded on Linux by */dev/urandom* and on windows by *CryptGenRandom()*.

Analysis of MSB

We examined one million keys generated on the system with Linux OS, with default seeding device as */dev/urandom*. The histograms of MSBs of the keys are plotted (Appendix-A). As seen from results, there is no significant bias is observed towards particular bits. From this we inferred that the size of generated keys is of the order of field size.

Analysis of frequency of ones and zeros

The histograms distribution results of ones and zeros of both private and public keys (Appendix-B, C) shows that the distribution follows the Gaussian as expected from randomly generated keys with mean

value 128. From this results we inferred that the keys generated from mbed TLS seem to be random as no bias is observed in the observed distribution.

Key validation

From the source code inspection we inferred that the mbed TLS ensures key validation by performing the checks as described in section 2.2.6. The Table 4.2 summarizes list of checks that mbed TLS performs for key validation.

4.10 Nettle version 3.2

The Nettle cryptographic library version 3.2 [46] supported curve are specified in Table 4.1. This library also doesn't set any default curve, we have selected *secp256r1* [2] for our analysis.

Brief description of PRNG

Yarrow is a family of random number generators [47] designed for cryptographic use. Nettle implements Yarrow-256, which is similar Yarrow-160, but uses AES and SHA256 to get the internal state of 256 bits. Yarrow uses two random sources, the entropy collected from two sources is pooled in slow pool and fast pool. Whenever the contribution of one of the sources reaches to 100 bits of entropy to the fast pool, a fast reseed happens and the fast pool is mixed into the internal state. Whenever two sources contributed at least 160 bits each to the slow pool, a slow reseed takes place. The content of both pools is mixed into internal state of PRNG.

Analysis of MSB

We examined one million keys generated on system with Linux OS with */dev/urandom* as default seeding device to PRNG. The histogram of the MSBs of both private and public key are plotted (Appendix-A). As seen from the results, there is no significant bias is observed towards particular bits, this may be due to reason that the size of generated keys is of the order of field size.

Analysis of frequency of ones and zeros

As seen from the histograms of ones and zeros of both private and public keys (Appendix-B, C), it is observed that the distribution follows the Gaussian as expected from randomly generated keys with mean value 128. We observed no significant bias in the distribution.

Key validation

From the source code inspection, it is observed that the condition *iv* (rQ = *O*) not implemented, this indicates a partial key validation, but the Nettle supports, only prime field curves with cofactor h=1 (Table 4.1). As described in section 2.2.6, for a prime field curve with cofactor *h*=1, performing the checks for conditions *i, ii, iii* also implies the check for condition *iv*. From this we hypothesized that the Nettle implements full key validation for the supported curves.

4.11 Microsoft CNG

The MS CNG [41] supported curves are specified in Table 4.1. This library also doesn't set any default curve, we selected NIST P-256 [5] curve parameters for our analysis.

Brief description of PRNG

The MS CNG uses *CryptGenRandom()* for generation of random data. The system and user data such as the process ID, thread ID, system clock, system time, system counter, memory status, free disk clusters and hashed user environment block is fed to the SHA-1. The output is used to seed an RC4 key stream [48]. This key stream is used to produce pseudorandom data.

Analysis of MSB

We examined one million keys generated on the system with windows 10 OS, with default seeding device as *CryptGenRandom()*. The histogram of MSBs of the keys is plotted (Appendix-A). As seen from results, there is no significant bias is observed towards particular bits.

Analysis of frequency of ones and zeros

The histograms distribution (Appendix-B, C) shows that the keys generated from this library also follow the distribution as expected

from randomly generated keys with mean value 128. We noticed no significant bias, so we inferred that generated keys seem to be random.

Key validation

The MS CNG performs EC key validation using the function *BCryptImportKeyPair()* with flag *BCRYPT_NO_KEY_VALIDATION* = 0. Since we don't have access to the source code of the key validation implementation, we were unsure about whether it is partial or full key validation. But as seen from the Table 4.1, all MS CNG supported curves (except *NUMSP256T1, NUMSP384T1, NUMSP512T1* (Edwards curves), Curve25519) have cofactor h as 1. So as described in section 2.2.6, for a prime field curve with cofactor $h = 1$, performing the checks for first three conditions also implies the check for condition *iv* (rQ = O) and for the Edwards curves and Curve25519 as explained in the previous section the addition laws are complete and works for all pair of inputs. From this, we hypothesized that the MS CNG performs full key validation for the supported curves.

Since the libraries specified above implements the PRNG methods from standards or reputed publications, we hypothesized that these methods might approved by professional cryptologists so we didn't further investigate on testing the randomness of the data.

4.12 Possibility of source library detection

From our analysis results it is observed that the value of a most significant byte of extracted keys from a source library for a given field uniformly distributes across the MSB value of the field but the MSB value of a source library with domain parameters as Curve25519 slightly biased and distinguishable because of the most significant bit of it's public key always set to zero. This difference allowed us to identify a source library that uses Curve25519 domain parameters for generating the given public key.

Our analysis results also show that the source libraries which uses elliptic curve parameters other than Curve 25519 don't show any variance in their key pair generation.

S.NO	Library Name	Supported Curves	Support for custom curve set and use	Default curve if Any
1	OpenSSL 1.0.1f	All curves defined over binary and prime field of the following standards: i.SECG (excluding secp256r1) ii.NIST (excluding P-256, B-223, K-223) iii.X9.62	Yes	No, user has to select the curve
2	Libgcrypt 1.7.2	i.NIST standards (only prime field) ii.Brainpool standards (no twisted curves) iii.GOST standards [49] iv.Twisted Edwards curve, Curve25519 v. secp256k1	No	No, user has to select the curve
3	Crypto++ 5.6.3	i.All curves defined over binary and prime field of the SECG ii.Brainpool standards (no twisted curves)	Yes	No, user has to select the curve
4	Bouncy Castle 1.55	All curves defined over binary and prime field of the following standards: i.SECG (sec2) ii.NIST iii.X9.62	No	No, user has to select the curve
5	libsodium 1.0.11	i.Twisted Edwards curve ii.Curve25519	No	Twisted Edwards curve
6	mbed TLS 2.3.0	i.The following curves from SECG: secp192k1, secp192r1 sep224k1, secp224r1, secp256k1, secp256r1 secp384r1, secp521r1 ii.brainpoolP256r1, brainpoolP384r1, brainpoolP512r1	No	No, user has to select the curve
7	Nettle 3.2	i.secp192r1, secp224r1, secp384r1, secp521r1 ii.Curve25519	No	No, user has to select the curve
8	Microsoft CNG	i.NIST standards(only prime field) ii.SECG secp160r1, secp160k1,secp192r1, secp192k1,secp224r1, secp224k1, secp256r1,secp256k1,secp384r1,secp521r1 iii.Brianpool standards and iv. Curves prime192v1,v2,v3 & prime239v1, v2, v3, prime256v1 of X9.62 standards v. EC192WAPI, WTLS12, WTLS9, WTLS7,Curve25519, NUMSP256T1, NUMSP384T1,NUMSP512T1	No	No, user has to select the curve

Table 4.1: *List of supported curves by various libraries*

S.NO	Library Name	Privatekey validation	Check for point at infinity	Check for point on curve	Check for rQ=O	Check for dG=Q	Remarks
1	OpenSSL 1.0.1f	Yes	Yes	Yes	Yes	Yes	Performs full key validation.
2	Libgcrypt 1.7.2	Yes	Yes	Yes	Yes	Yes	Performs full key validation.
3	Crypto++ 5.6.3	Yes	Yes	Yes	Yes	Yes	Performs full key validation.
4	Bouncy Castle 1.55	Yes	Yes	Yes	Yes	No	Performs full key validation.
5	libsodium 1.0.11	Yes	No	No	No	No	Implements Ed25519 digital signature with Montgomery curve (Curve25519) & Twisted Edwards curve, The additional point validation may not required.
6	mbed TLS 2.3.0	Yes	Yes	Yes	Yes	Yes	Performs full key validation.
7	Nettle 3.2	Yes	Yes	Yes	Implied by the supported curves with $h = 1$	No	Performs full key validation for all supported curves.
8	Microsoft CNG	Yes	Yes	Yes	Implied by the supported curves with $h = 1$	No	Performs full key validation for all supported curves.

Table 4.2: *Supported EC key validation checks for various libraries*

5 Analysis of EC keys from smart cards

We analysed EC (elliptic curve) keys generated from software libraries as described in Chapter 4. The analysis results indicate that the libraries didn't exhibit any variability in their generated keys. We also showed that the elliptic curve *Curve25519* used in some of the libraries has peculiar characteristics as compared to regular elliptic curves.

We extended our analysis to EC keys generated from smart cards. The motivation is to compare the keys generated from the cards and discuss causes for similarities and differences with keys generated from software libraries.
We inspected keys generated from the following cards:

i FeitianA22CR

ii NXPJCOP31

iii InfCJTOP 80K SLJ 52GLA080AL M8.4

iv NXPJ2E145

v G&D Sm@rtcafe 6.0.

vi NXPJ2A081

5.1 Brief description of random number generators

The implementation details of random number generators used in smart cards not known to us, however, we hypothesized the use of probable random number generators by a particular card vendor from DRBG validation list [50].

Most of the card manufacturers use HASH_Based_DRBG or CTR_DRBG [40] with prediction resistance (the resistance which describes the effect on the security of future bits of DRBG even if DRBG internal state compromises) support and derivative function as AES or Triple DES or SHA256. The entropy gathering mechanism to seed the DRBG is not known to us. The Table 5.1 summarizes use of DRBG by a particular smart card manufacturer.

S.NO	Card manufacturer	Type of DRBG	Support for prediction resistance (Yes/No)	Used derivative function
1	Feitian	CTR_DRBG	Yes	AES_256
2	NXP	Hash_Based_DRBG	No	SHA_256
3	Infineon	CTR_DRBG	No	AES_128
4	Giesecke & Devrient	CTR_DRBG	No	AES_256

Table 5.1: *Type of DRBG used by smart cards [50]*

We primarily analysed the most significant byte (MSB) to verify that whether the size of keys generated from the cards are of the size of a given field or not.

We have also analysed the frequency of a number of ones and zeros from the keys to verify whether the generated keys are random or any bias exists (since EC private key is a random number, if any bias exists in the private key depicts the weakness in TRNG implementation of the card).

The inspection of a group of keys for the exceptionally longer key-generation times was also performed with no observable differences. The histogram distribution of key generation time for various cards is given at Appendix-G.

5.2 Detailed analysis

For all the cards we set the EC parameters to *secp192r1*, key pair generated and exported about 20k from card *FeitianA22CR* and 84k from *NXPJCOP31*, 330k from *InfCJTOP* and 530k from *NXPJ2E145*, and 50k from *G&D Sm@rtcafe 6.0* and 1.5 million keys from *NXPJ2A081*. We also measured the time taken for key generation.

5.2.1 Analysis of MSB

We plotted the histogram distribution of most significant byte of both private and public keys for all the six cards (Appendix-D).

The histogram distribution of the cards *FeitianA22CR, NXPJCOP31, InfCJTOP80K SLJ 52GLA080AL M8.4, NXPJ2E145, NXPJ2A081* shows that the generated key size is of the order of the field size (in this case the field size is 192-bit) with no zero padding to the key.

For the EC keys from card G&D Sm@rtcafe 6.0., we observed that the private key size is not the order of the field size (in this case field size is 192-bit). The private key size varies between 180 to 192 bits and public key size is the order of the field size.

Since some of the private keys sizes are not the size of the field, we padded with zeros. The histogram distribution of MSB value of both private and public key are plotted (Appendix-D).

As seen from the plots, the histogram distribution of public key MSB is uniformly spread across all the 256 values. The distribution of private key MSB slightly biased towards the MSB value zero, this is because of padding the private key to field size.

5.2.2 Analysis of frequency of ones and zeros

We plotted the histogram distribution of frequency of the number of ones and zeros in a given private and public keys from the above cards, the analysis for the same described below.

NXPJCOP31

Figure 5.1 and 5.2 contains an example graphs which illustrates the distribution of frequency of ones and zeros of private and public keys generated from *NXPJCOP31* card with selected curve *secp192r1.*

As seen from the plots, there is a slight bias is observed in the distribution, this apparently affected the distribution of public key ones and zeros. Though this bias may not be significant to find the variance in EC key pair generation but it opens the door to investigate the strength of TRNG implementation used in the card.

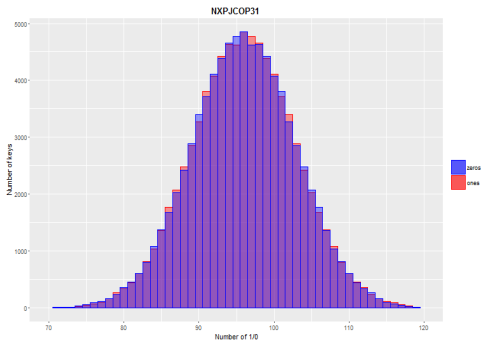


Figure 5.1: *Private key*

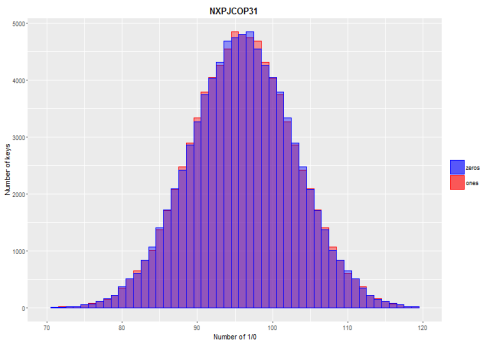


Figure 5.2: *Public key*

G&D Sm@rtcafe 6.0

Figure 5.3 and 5.4 shows histogram distribution of frequency of ones and zeros of private and public keys generated from G&D Sm@rtcafe 6.0. card with selected curve *secp192r1*.

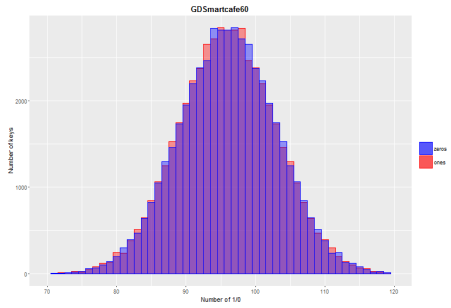


Figure 5.3: *Private key*

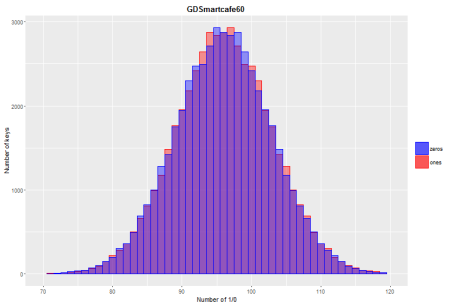


Figure 5.4: *Public key*

As seen from the figures, there is a slight bias exists in private keys; this bias is due to padding the private key with zeros.

The private key size should be of the size of order r but the private keys generated from this card differs in their size (varies from 180 to 192 bits). So from this, we hypothesized that there may a bug in its TRNG implementation or the card's elliptic curve implementation might not take care of key validation during the key generation phase. If a proper key validation mechanism is implemented, the card performs this check before producing a final key.

For the EC keys extracted from the cards *InfCJTOP, NXPJ2E145, NXPJ2A081,* the histogram distribution (Appendix-E, F) shows that it follows a Gaussian distribution with mean value as 96 (the key size is 192-bit) as expected for a random sequence, we observed no significant bias and for the EC keys extracted from card *FeitianA22CR* there is a slight bias observed in the distribution of the frequency of the number of ones and zeros (Appendix-E, F). Since the extracted keys only about 20k we hypothesized that this may be due to statistical fluctuation.

5.3 Detecting probable standard EC parameters from a black-box implementation

The MSB analysis also helps in guessing the probable standard elliptic curve parameters used by a card who's implementation details are unknown.

For example, for a given key of size 192-bit (extracted from a card with black-box implementation), the standards NIST, SECG and ANSI X9.62 and Brainpool only offer the 192-bit curve parameters over the prime field.

The prime number p used in NIST, SECG and ANSI X9.62 standards is 6277101735386680763835789423207666416083908700390 324961279 (0xFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFEFFFFFFFFFFFFFFFF), and the prime number p used in Brainpool standards is 478166898390616624 295500189434492377325911965525301319336 7 (0xC302F41D932A36CD A7A3463093D18DB78FCE476DE1A86297).

The figures 5.5, 5.6, 5.7 and 5.8 shows the public key MSB distribution of a 192-bit Brainpool curve (brainpoolp192r1), a NIST curve (P-192), an SECG curve (secp192r1) and ANSI X9.62 curve (prime192v1) generated from the software library Bouncy Castle. In the figures, the horizontal axis shows the value of MSB and vertical axis shows the frequency of the MSB values.

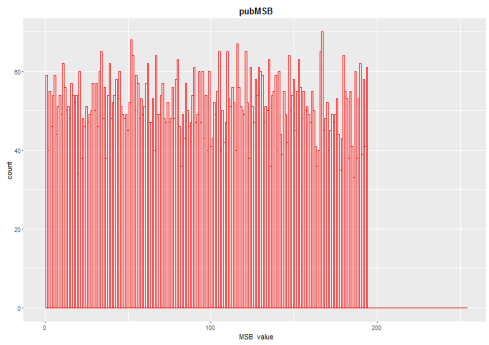


Figure 5.5: *brainpoolp192r1*

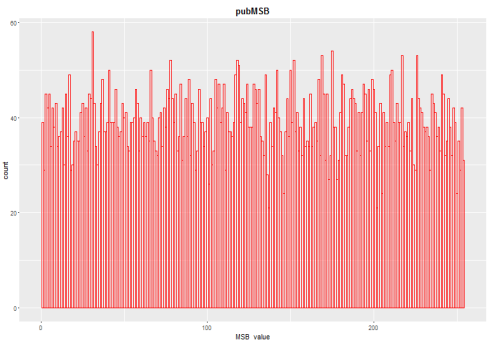


Figure 5.6: *P-192*

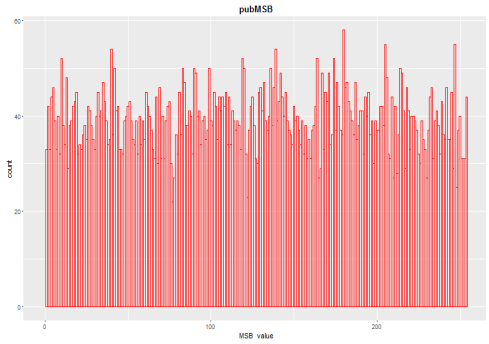


Figure 5.7: *secp192r1*

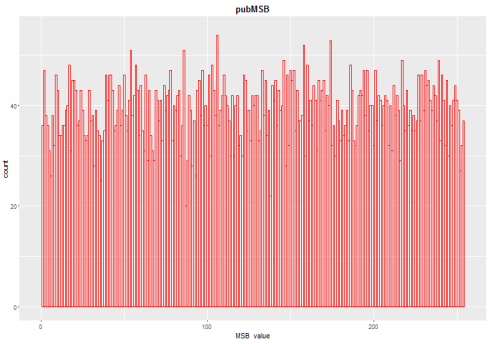


Figure 5.8: *prime192r1*

As seen from the histogram plots, the MSB value for the curves from NIST, SECG and ANSI X9.62 distributed over 0 to 0xFF (256) because the prime p value is 0xFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF-FEFFFFFFFFFFFFFFFF, in case of Brianpool curve the prime p value is; 0xC302F41D932A36CDA7A3463093D18DB78FCE476DE1A86297 so the MSB value distributed over 0 to 0xC3 (195).

From the MSB analysis of EC keys generated from cards (Appendix-D), it is observed that the histogram distribution of public key MSB value spread over 0-256.

In comparison, it is evident that the elliptic curve parameters used by the cards are from one of the NIST, SECG, ANSI X9.62 standards but not from the Brainpool standards.

6 Conclusions

In this work, we provided statistical analysis performed on EC key pairs generated and extracted from eight software cryptographic libraries and six cryptographic smart cards.

Our analysis results show that the most of the selected versions of cryptographic libraries didn't exhibit any significant variance in their key pair generation except the source library with curve25519. The MSB value of extracted public keys from a source library for a given field uniformly distributes across the MSB value of the modulus of the field but the MSB value of a source library with domain parameters as Curve25519 slightly biased and distinguishable. This difference allowed us to identify a source library that uses Curve25519 domain parameters for generating the given public key.

The Analysis results of keys extracted from smart cards show that the most of the cards didn't exhibit any significant variance in their key pair generation; however, from the card *NXPJCOP31* we observed a slight bias in the frequency distribution of ones and zeros of private keys. We hypothesized that there may be the weakness in TRNG implementation.

We also detected a small variance in key pair generation of the card *G&D Sm@rtcafe 6.0*. This might be due to a bug in TRNG implementation or this card might not have taken care of EC key validation. However this variance may not be significant to classify the keys generated from this card but the smaller size of private keys may have the impact on solving the ECDLP.

We also demonstrated that the MSB analysis can also help in guessing probable standard elliptic curve parameters from a black-box implementation.

Bibliography

[1] *Comparison of TLS implementations*. URL: https://en.wikipedia.org/wiki/%20Comparison_of_TLS_implementations#Overview (visited on 11/28/2016).

[2] *Recommended Elliptic Curve Domain Parameters(SEC2)*. Tech. rep. Standards for Efficient Cryptography Group, 2000.

[3] *ANSI X9.62 Public Key Cryptography For The Financial Services Industry:The Elliptic Curve Digital Signature Algorithm*. Tech. rep. Internet Engineering Task Force, 1998.

[4] "ECC Brainpool Standard Curves and Curve Generation". In: ed. by Dr.Manfred Lochter. 2005.

[5] *RECOMMENDED ELLIPTIC CURVES FOR FEDERAL GOVERNMENT USE*. Tech. rep. National Institute of Standards and Technology, 1999.

[6] *Nacl*. URL: http://nacl.cr.yp.to/ (visited on 08/04/2016).

[7] *libsodium 1.0.11*. URL: https://download.libsodium.org%20/%20libsodium/releases/libsodium-1.0.11.tar.gz (visited on 08/04/2016).

[8] Christof Paar and Jan Pelzl. *Understanding cryptography: a textbook for students and practitioners*. Springer Science & Business Media, 2009. ISBN: 9783642041006.

[9] Julio López and Ricardo Dahab. *An Overview of Elliptic Curve Cryptography*. Tech. rep. 2000.

[10] S.Vanstone. "Response to NIST proposal". In: (1992).

[11] Michel Abdalla, Mihir Bellare, and Phillip Rogaway. "DHAES: An Encryption Scheme Based on the Diffie-Hellman Problem." In: *IACR Cryptology ePrint Archive* 1999 (1999), p. 7.

[12] *SEC 1 : Elliptic Curve Cryptography*. Tech. rep. Standards for Efficient Cryptography Group, 2009.

[13] Chae Hoon Lim and Pil Joong Lee. "A key recovery attack on discrete log-based schemes using a prime order subgroup". In: *Annual International Cryptology Conference*. Springer. 1997, pp. 249–263.

[14] Adrian Antipa et al. "Validation of elliptic curve public keys". In: *International Workshop on Public Key Cryptography*. Springer. 2003, pp. 211–223.

[15] Tim Güneysu, Christof Paar, and Jan Pelzl. "On the security of elliptic curve cryptosystems against attacks with special-purpose hardware". In: *Special-Purpose Hardware for Attacking Cryptographic Systems–SHARCS'06* (2006), pp. 03–04.

[16] Stephen Pohlig and Martin Hellman. "An improved algorithm for computing logarithms over GF(p) and its cryptographic significance (Corresp.)" In: *IEEE Transactions on information Theory* 24.1 (1978), pp. 106–110.

[17] John M Pollard. "Monte Carlo methods for index computation (mod \ p)". In: *Mathematics of computation* 32.143 (1978), pp. 918–924.

[18] Robert Gallant, Robert Lambert, and Scott Vanstone. "Improving the parallelized Pollard lambda search on anomalous binary curves". In: *Mathematics of Computation of the American Mathematical Society* 69.232 (2000), pp. 1699–1705.

[19] Michael J Wiener and Robert J Zuccherato. "Faster attacks on elliptic curve cryptosystems". In: *International Workshop on Selected Areas in Cryptography*. Springer. 1998, pp. 190–200.

[20] Paul C Van Oorschot and Michael J Wiener. "Parallel collision search with cryptanalytic applications". In: *Journal of cryptology* 12.1 (1999), pp. 1–28.

[21] *Debdeep Mukhopadhyay, Elliptic curve cryptography [Power Point slides]*. URL: `http://cse.iitkgp.ac.in/~debdeep/pres/TI/ecc.pdf` (visited on 07/04/2016).

[22] *Elliptic curve performance: NIST vs Brainpool*. URL: `https://tls.mbed.org/kb/cryptography/elliptic-curve-performance-nist-vs-brainpool` (visited on 12/01/2016).

[23] Huseyin Hisil et al. "Twisted Edwards curves revisited". In: *International Conference on the Theory and Application of Cryptology and Information Security*. Springer. 2008, pp. 326–343.

[24] Daniel Bernstein et al. "ECM using Edwards curves". In: *Mathematics of Computation* 82.282 (2013), pp. 1139–1179.

[25] Daniel J Bernstein et al. "High-speed high-security signatures". In: *Journal of Cryptographic Engineering* 2.2 (2012), pp. 77–89.

[26] Daniel J Bernstein et al. "Twisted edwards curves". In: *International Conference on Cryptology in Africa*. Springer. 2008, pp. 389–405.

[27] Daniel J Bernstein. "Curve25519: new Diffie-Hellman speed records". In: *International Workshop on Public Key Cryptography*. Springer. 2006, pp. 207–228.

[28] Petr Švenda et al. "The Million-Key Question–Investigating the Origins of RSA Public Keys". In: *25th USENIX Security Symposium. Proceedings*. 2016.

[29] Solomon W Golomb et al. *Shift register sequences*. Aegean Park Press, 1982. ISBN: 0894120484.

[30] Andrew Rukhin et al. *A statistical test suite for random and pseudorandom number generators for cryptographic applications*. Tech. rep. DTIC Document, 2001.

[31] *OpenSSL 1.0.1f*. URL: `https://www.openssl.org/source/openssl-1.0.1f.tar.gz` (visited on 03/16/2016).

[32] *Libgcrypt 1.7.2*. URL: `https://www.gnupg.org/ftp/gcrypt/libgcrypt/libgcrypt-1.7.2.tar.bz2` (visited on 07/16/2016).

[33] Peter Gutmann. "Software Generation of Practically Strong Random Numbers." In: *Usenix Security*. 1998.

[34] Sharon S Keller. "NIST-recommended random number generator based on ANSI X9. 31 appendix A. 2.4 using the 3-key triple DES and AES algorithms". In: *NIST Information Technology Laboratory-Computer Security Division, National Institute of Standards and Technology* (2005).

[35] *DIGITAL SIGNATURE STANDARD (DSS)*. Tech. rep. National Institute of Standards and Technology, 2000.

[36] Daniel J Bernstein and Tanja Lange. "Faster addition and doubling on elliptic curves". In: *International Conference on the Theory and Application of Cryptology and Information Security*. Springer. 2007, pp. 29–50.

[37] Peter L Montgomery. "Speeding the Pollard and elliptic curve methods of factorization". In: *Mathematics of computation* 48.177 (1987), pp. 243–264.

[38] Simon Josefsson and Ilari Liusvaara. "Edwards-curve Digital Signature Algorithm (EdDSA)". In: *draft-irtf-cfrg-eddsa-02 (work in progress)* (2016).

[39] *Crypto++ 5.6.3*. URL: `https://www.cryptopp.com/cryptopp563.zip` (visited on 08/08/2016).

[40] Elaine B Barker and John Michael Kelsey. *Recommendation for random number generation using deterministic random bit generators (revised)*. US Department of Commerce, Technology Administration, National Institute of Standards and Technology, Computer Security Division, Information Technology Laboratory, 2007.

[41] *Microsoft Cryptography API: Next Generation (CNG*. URL: https://%20msdn.microsoft.com%20/en-us%20/%20library%20/%20windows%20/desktop%20/%20aa376210(v=vs.85).aspx (visited on 09/18/2016).

[42] *Bouncy Castle 1.55*. URL: https://www.bouncycastle.org/download/bcprov-ext-jdk15on-155.jar (visited on 08/23/2016).

[43] Kai Michaelis, Christopher Meyer, and Jörg Schwenk. "Randomly failed! The state of randomness in current Java implementations". In: *Cryptographers' Track at the RSA Conference*. Springer. 2013, pp. 129–144.

[44] *CVE Details*. URL: https://www.cvedetails.com/cve/CVE-2015-7940/ (visited on 09/20/2016).

[45] *mbed TLS 2.3.0*. URL: https://tls.mbed.org/download/start/mbedtls-2.3.0-apache.tgz (visited on 09/01/2016).

[46] *Nettle 3.2*. URL: https://ftp.gnu.org/gnu/nettle/nettle-3.2.tar.gz (visited on 09/08/2016).

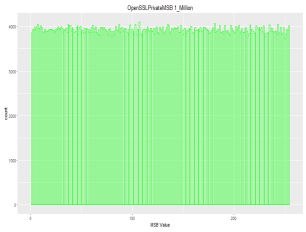
[47] John Kelsey, Bruce Schneier, and Niels Ferguson. "Yarrow-160: Notes on the design and analysis of the yarrow cryptographic pseudorandom number generator". In: *International Workshop on Selected Areas in Cryptography*. Springer. 1999, pp. 13–33.

[48] *Randomness Requirements for Security*. Tech. rep. Internet Engineering Task Force, 2005.

[49] Vasily Dolmatov and Alexey Degtyarev. "GOST R 34.10-2012: Digital Signature Algorithm". In: (2013).

[50] *DRBG Validation List*. URL: http://csrc.nist.gov/groups/STM/cavp/documents/drbg/drbgnewval.html (visited on 11/28/2016).

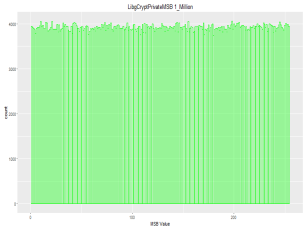
A Histogram distribution of most significant bytes of private and public keys (software libraries)



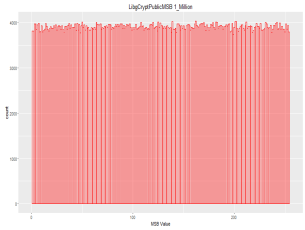
Private key

Public key

OpenSSL

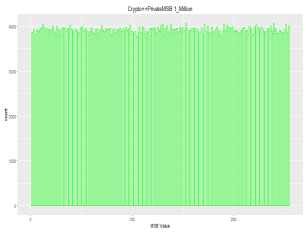


Private key

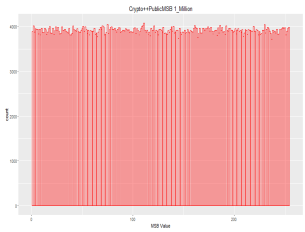


Public key

Libgcrypt

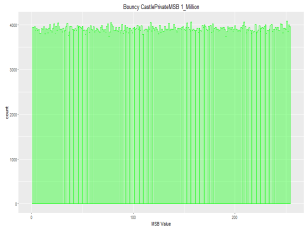


Private key

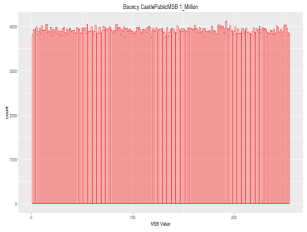


Public key

Crypto++

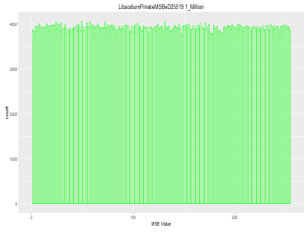


Private key

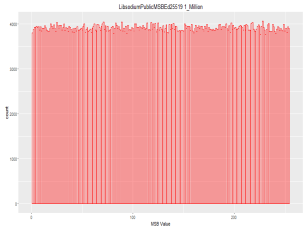


Public key

Bouncy Castle

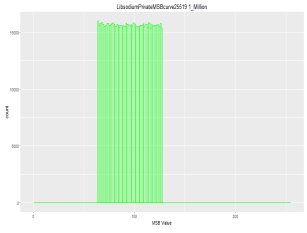


Private key

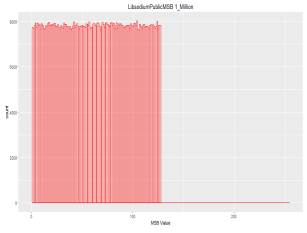


Public key

libsodium with Twisted Edwards curve

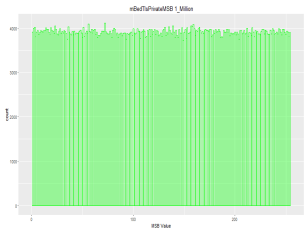


Private key

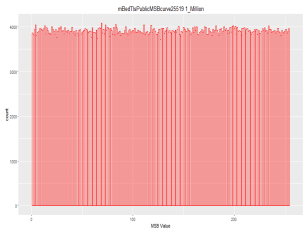


Public key

libsodium with Curve25519

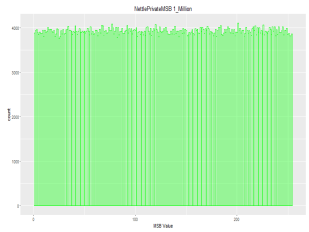


Private key

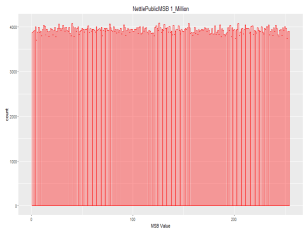


Public key

mbed TLS

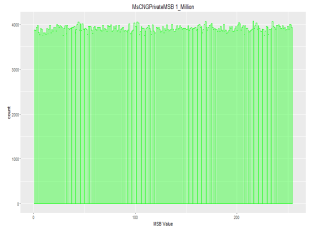


Private key

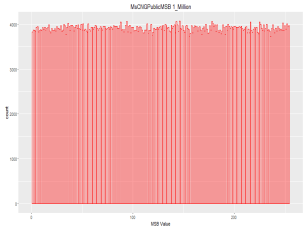


Public key

Nettle



Private key



Public key

MS CNG

B Histogram distribution of frequency of ones and zeros of public keys

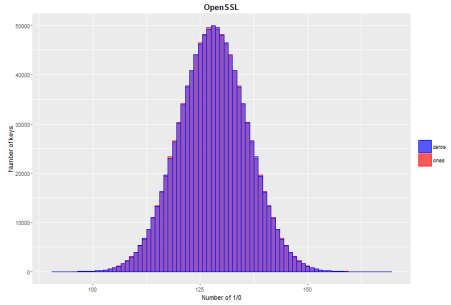


Figure B.1: *OpenSSL*

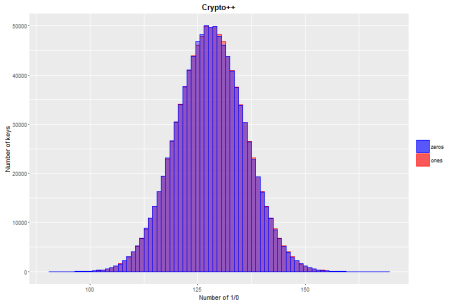


Figure B.2: *Crypto++*

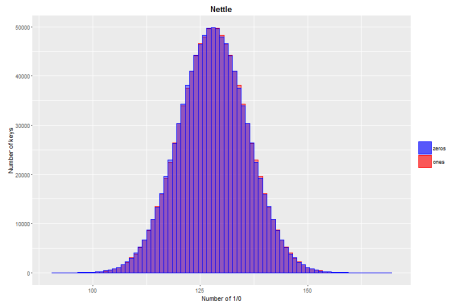


Figure B.3: *Nettle*

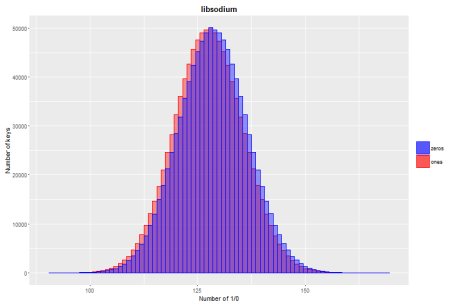


Figure B.4: *libsodium*

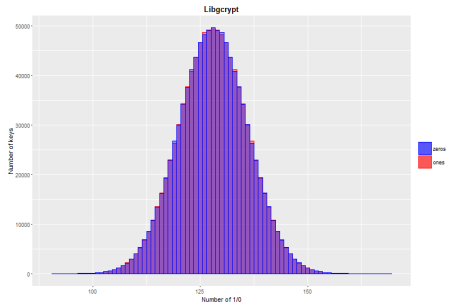


Figure B.5: *Libgcrypt*

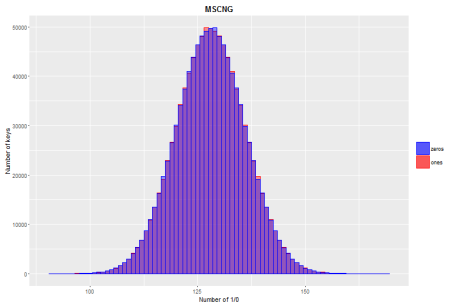


Figure B.6: *MS CNG*

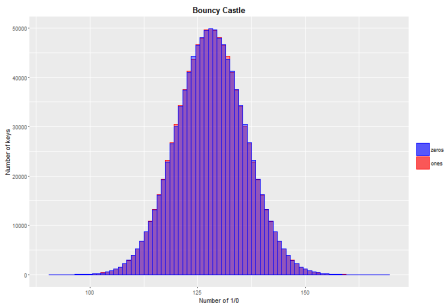


Figure B.7: *Bouncy Castle*

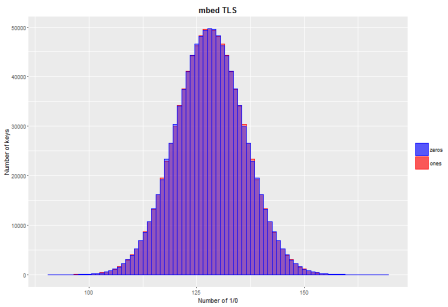


Figure B.8: *mbed TLS*

C Histogram distribution of frequency of ones and zeros of private keys

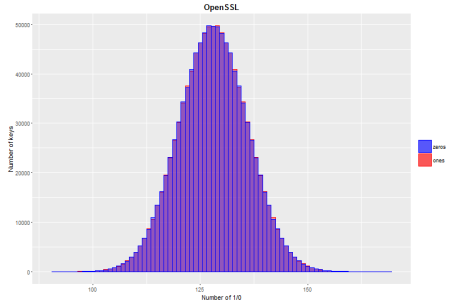


Figure C.1: *OpenSSL*

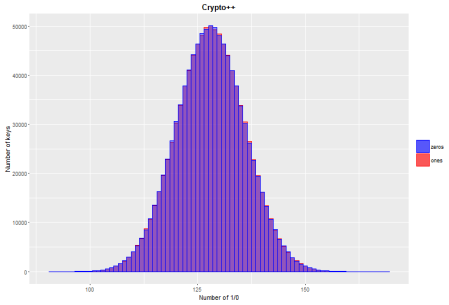


Figure C.2: *Crypto++*

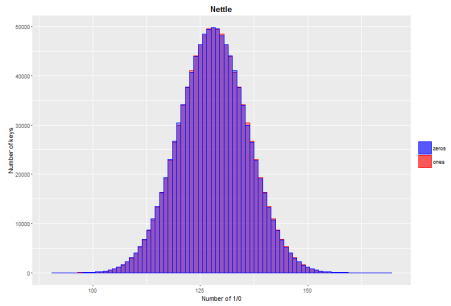


Figure C.3: *Nettle*

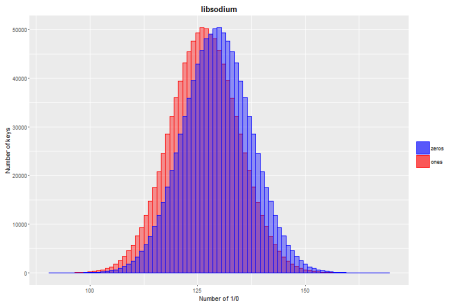


Figure C.4: *libsodium*

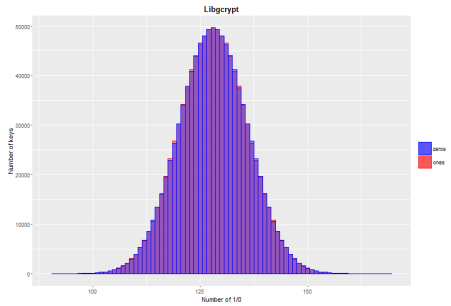


Figure C.5: *Libgcrypt*

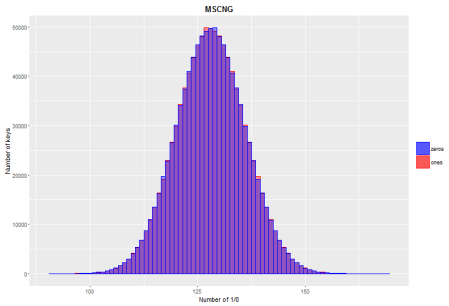


Figure C.6: *MS CNG*

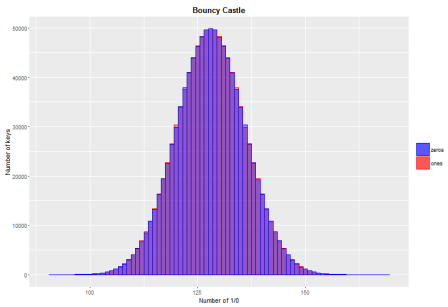


Figure C.7: *Bouncy Castle*

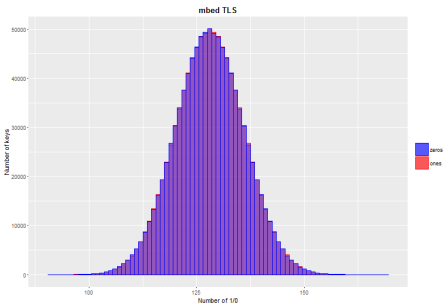
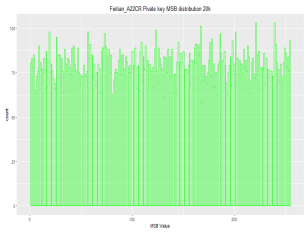
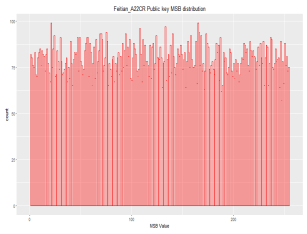


Figure C.8: *mbed TLS*

D Histogram distribution of most significant bytes of private and public keys (Smart cards)

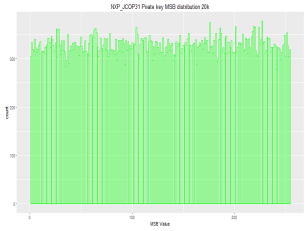


Private key

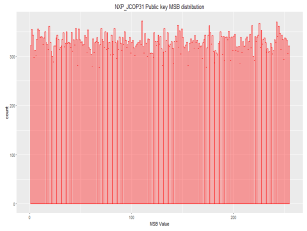


Public key

Feitian

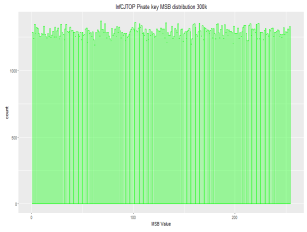


Private key

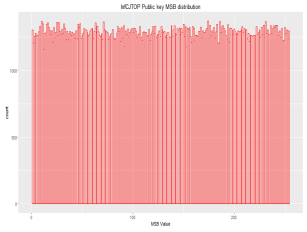


Public key

NXPJCOP31

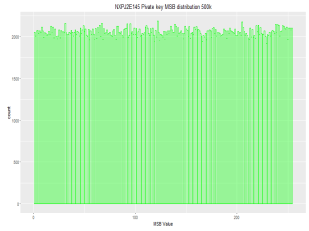


Private key

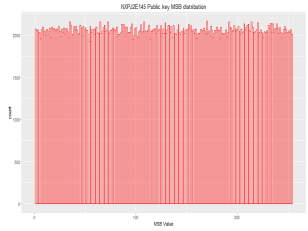


Public key

InfCJTOP

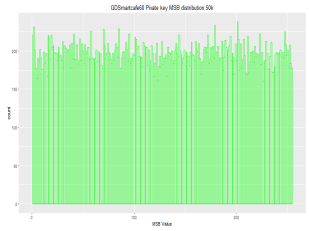


Private key

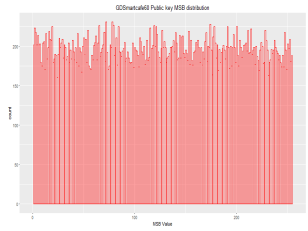


Public key

NXPJ2E145

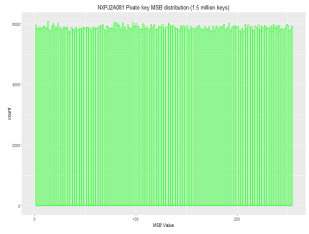


Private key

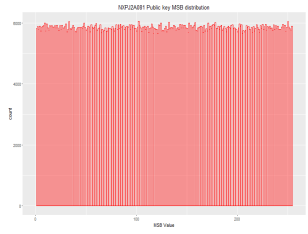


Public key

G&D Sm@rtcafe 6.0.



Private key



Public key

NXPJ2A081

E Histogram distribution of frequency of ones and zeros of private keys (Smart cards)

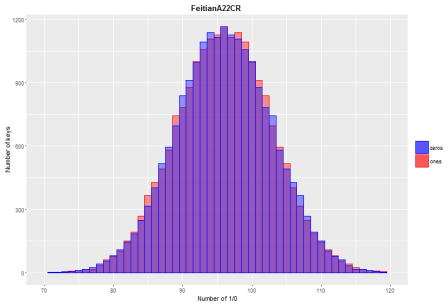


Figure E.1: *FeitianA22CR*

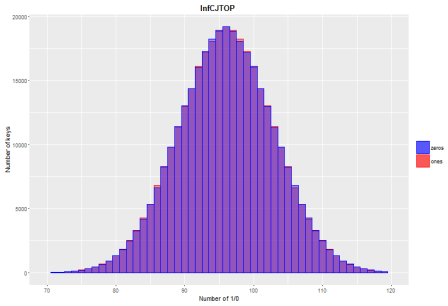


Figure E.2: *InfCJTOP*

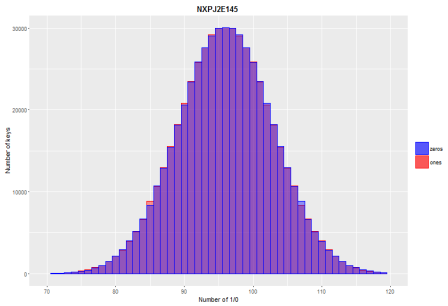


Figure E.3: *NXPJ2E145*

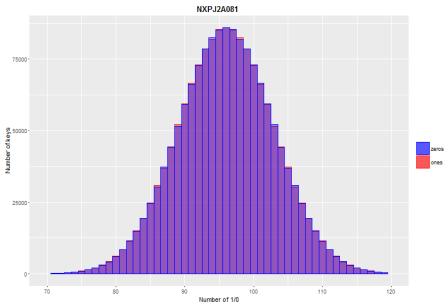


Figure E.4: *NXPJ2A081*

F Histogram distribution of frequency of ones and zeros of public keys (Smart cards)

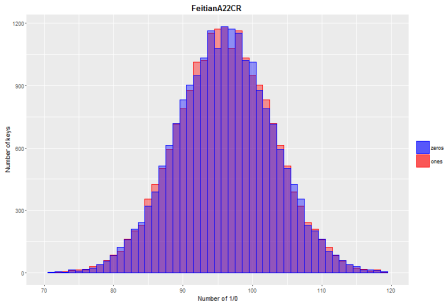


Figure F.1: *FeitianA22CR*

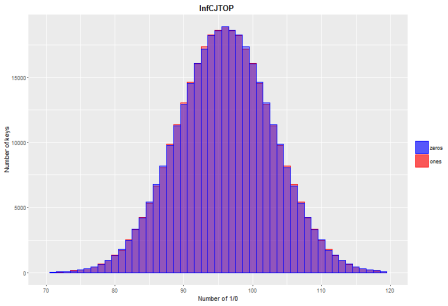


Figure F.2: *InfCJTOP*

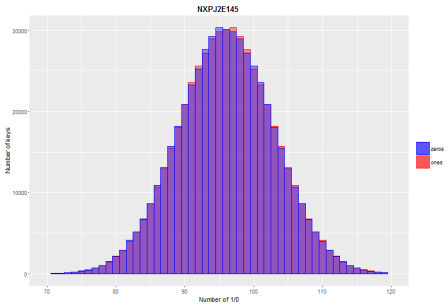


Figure F.3: *NXPJ2E145*

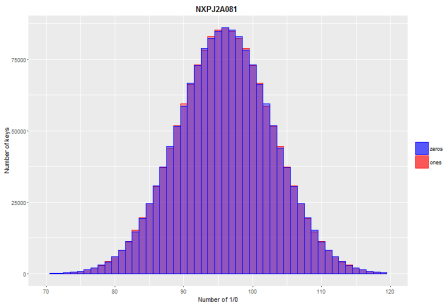


Figure F.4: *NXPJ2A081*

G Histogram distribution of key generation time (smart cards)

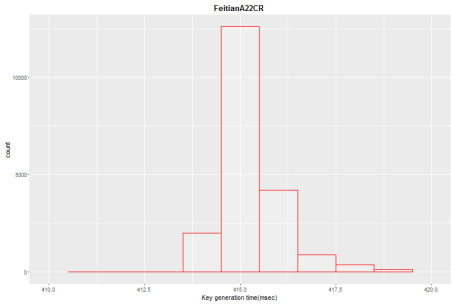


Figure G.1: *FeitianA22CR*

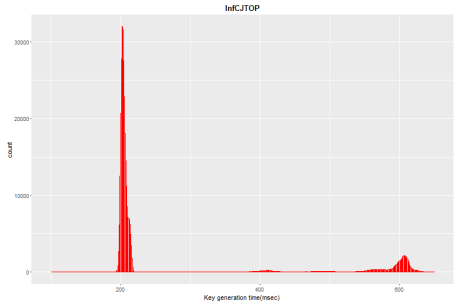


Figure G.2: *InfCJTOP*

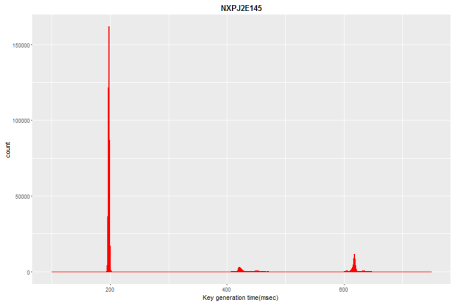


Figure G.3: *NXPJ2E145*

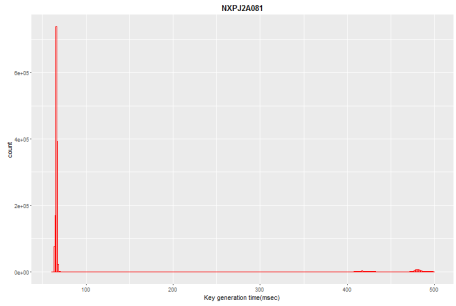


Figure G.4: *NXPJ2A081*

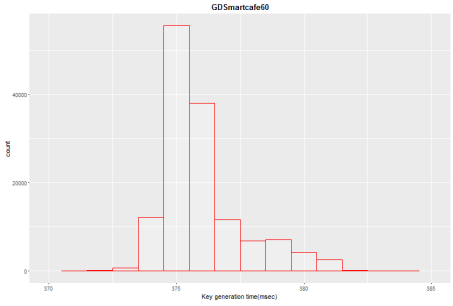


Figure G.5: *G&D Sm@rtcafe 6.0.*

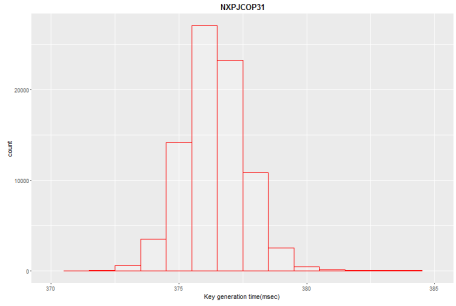


Figure G.6: *NXPJCOP31*

H List of source files/class and methods responsible for key pair generation

S.NO	Library	Version	Source file/Class	Method
1	OpenSSL	1.0.1f	crypto/ec/ec_key.c	EC_KEY_generate_key()
2	Libgcrypt	1.7.2	/cipher/ecc.c	nist_generate_key()
3	Crypto++	5.6.3	eccrypto.cpp	privateKey.Initialize(prng, params) privateKey.MakePublicKey()
4	Bouncy Castle	1.55	KeyPairGenerator	generateKeyPair()
5	libsodium	1.0.11	src/libsodium /crypto_sign /ed25519/ref10 /keypair.c	crypto_sign_ed25519_keypair()
6	mbed TLS	2.3.0	library/ecp.c	mbedtls_ecp_gen_key()
7	Nettle	3.2	ecdsa-keygen.c	ecdsa_generate_keypair()
8	Microsoft CNG	Windows 10	CNG Cryptographic Primitive Functions	BCryptGenerateKeyPair()

Table H.1: The source files and methods responsible for key pair generation

I List of source files/class and methods responsible for key validation

S.NO	Library	Version	Source file/class	Method
1	OpenSSL	1.0.1f	crypto/ec /ec_key.c	EC_KEY_check_key()
2	Libgcrypt	1.7.2	/cipher/ecc.c	check_secret_key()
3	Crypto++	5.6.3	ecp.cpp	ValidateParameters() VerifyPoint()
4	Bouncy Castle	1.55	ECPoint	isValid()
5	libsodium	1.0.11	---	---
6	mbed TLS	2.3.0	library/ecp.c	ecp_check_pubkey_sw()
7	Nettle	3.2	ecdsa-keygen-test.c	ecc_valid_p()
8	Microsoft CNG	Windows 10	BCryptImportKeyPair	BCRYPT_NO_KEY_VALIDATION If this flag is enable, no validation is performed.

Table I.1: The source files and methods responsible for key validation

J Data attachment

`Linux_ECC`

C and C++ source codes for generating EC keys from the cryptographic libraries (Linux).

`Windows_ECC`

Source codes for generating EC keys from Microsoft CNG and Bouncy Castle (Windows).

`R_scripts`

R scripts for generating figures :- MSB histogram distribution, histogram distribution of frequency of number of ones and zeros, histogram distribution of key generation time for smart cards.

`Classification_tool`

Source code of a tool that classifies a probable library based on the given public key.