Pomcor JavaScript Cryptographic Library (PJCL)

Version 0.9 (beta test version)

Contents

1 Functionality provided in Version 0.9

The primary goal of this version of the library is to support the implementation of cryptographic authentication and remote identity proofing. To that purpose it provides:

- Big integer arithmetic, including:
  - Long multiplication and Karatsuba multiplication [1, § 15.1.2].
  - Montgomery reduction [2, § 14.3.2].
  - Sliding window exponentiation [2, Algorithm 14.85] in a generic monoid, with specializations including modular exponentiation with Montgomery reduction and scalar multiplication of a point of an elliptic curve. (The latter may be implemented differently in a future version.) Our implementation of modular exponentiation with Montgomery reduction is several times faster than the one in the Stanford JavaScript Cryptographic Library (SJCL) [3] according to the performance testing described below in Section 8.1.
- Elliptic curve group operations, in the NIST curves P-256 [4, § D.2.3] and P-384 [4, § D.2.4]. Other curves may be supported in the future.
- The following hash functions and message authentication codes:
  - SHA-256 and SHA-384 [5].
  - HMAC-SHA256 and HMAC-SHA384 [6, 7].

- Pseudo-random number generation based on the NIST Hash-Based Deterministic Random Bit Generator (Hash_DRBG) [8] with hash functions SHA-256 and SHA-384.
- Prime number generation using the Miller-Rabin algorithm.
- DSA [4] with 128 bits of security strength, which requires a 256-bit private key and a 3072-bit public key [9, Table 2], including:
  - Generation of domain parameters.
  - Key pair generation.
  - Signature generation.
  - Signature verification.
- ECDSA [10], [11, § 2.6.2] with NIST curves P-256 and P-384, which provide 128 and 196 bits of security strength respectively [9, Table 2], including:
  - Key pair generation.
  - Public key validation [10, § 6.2].
  - Signature generation.
  - Signature verification.

Future versions of the PJCL may support secure messaging and data protection at rest, and to that purpose may provide encryption and key agreement functionality.

2 Requirements

PJCL does not require any recent features of JavaScript, nor any particular JavaScript engine, runtime environment or framework. The PJCL API is a collection of global functions and variables whose names are all prefixed by `pjcl` to avoid name conflicts. (The PJCL acronym and the `pjcl` prefix are trademarks of Pomcor.) Therefore PJCL can be used wherever JavaScript is used. It can be used in a browser, in a native app (e.g. using React Native [12]), or in a server (e.g. using node.js [13]).

The PJCL pseudo-random bit generator must be seeded with random bits with sufficient entropy obtained from a true random source. It may be reseeded before generating random bits for the sake of prediction resistance [8, § 8.8]. You are responsible for providing the random bits used for seeding or reseeding. Methods for obtaining entropy are discussed below in Section 6.17.1. `Math.random` does not provide entropy.

3 License

The PJCL library can be used subject to the terms of the PJCL license, which can be found at `https://pomcor.com/pjcl/pjcl-license.txt`.

4 Downloadable zip archive

The current version of the PJCL library can be downloaded as a zip archive that can be found at `https://pomcor.com/pjcl/pjcl-090.zip`. The archive contains a `pjcl` directory, which itself contains the following files:

- The file `pjcl.js` contains the PJCL library.
- The file `pjcl-withArgChecking.js` differs from `pjcl.js` in that most of the API functions include code that checks the validity of their arguments. This may be useful for debugging applications that use the library, and as a precise specification of the properties of the arguments expected by the functions.
- The directory `KaratsubaThresholds` contains files that let you estimate optimal thresholds for Karatsuba multiplication and Karatsuba squaring in a particular JavaScript environment, as described below in Section 7.
- The file `browserEntropy.js` contains examples of how to generate random bits in browsers that support the Web Crypto API.
- The directory `ModExpPerfTesting` contains files that allow you to test the performance of modular exponentiation on a browser using long multiplication or Karatsuba multiplication and compare it to the performance of SJCL, as described below in Section 8.1.

- The directory `DSAPerfTesting` contains files that allow you to test the performance of DSA on a browser using long multiplication or Karatsuba multiplication, as described below in Section 8.2. ECDSA performance testing may be provided in the future.

5 Data encodings

5.1 Numbers in JavaScript

With some exceptions, JavaScript numbers are represented in IEEE 754 double-precision (64-bit) floating point format [14], which allows every nonnegative integer n in the range $0 \leq n < 2^{53}$ to be represented exactly.

5.2 Big integers in PJCL

PJCL represents nonnegative integers of arbitrary size in base $B = 2^\beta$, with $\beta = 24$. Following tradition, we refer to the digits of the base-B representation as *limbs*. A limb is thus a 24-bit quantity. It is unlikely but not impossible that the number of bits per limb will change in the future. Your own code should use the variables of Section 6.3.1 to avoid hardcoding the number of bits per limb.

The limbs are stored in an array. For performance reasons, the least significant limb is the first element of the array, i.e. the element with index 0. Thus, the index of each limb is its weight in the base-B representation: limb λ_i of the nonnegative integer $N = \sum_{0 \leq i < n} \lambda_i B^i$ is stored at position i in the n-limb array that represents N.

The order in which the limbs are stored in the array only matters for understanding the implementation of the library; it should not matter to developers who use the API, and it does not affect the API-level metaphors. For example, "shifting left by one limb" shall mean shifting by one limb towards the most significant end of the array, i.e. multiplying by B, even though the most significant limb is the array element with the highest index, which is the rightmost element in an array literal; and the "leading limb" shall mean the most significant limb.

JavaScript arrays are not objects, but can have properties like objects. A negative integer is represented by encoding its absolute value as an array of limbs, and giving the array a property `negative` with value `true`. A nonnegative integer does not have a `negative` property.

We use the term *big integer* to refer to an integer represented in base B as an array of limbs with an optional `negative property`. A big integer has a unique representation. Leading zero limbs are not allowed, i.e. the most significant limb must not be zero. The big integer zero is represented as an empty array without a `negative` property.

For the sake of performance and code footprint minimization, some functions ignore the `negative` property of big integer arguments and thus operate on the absolute values of those arguments, while other functions take the `negative` property into account and thus operate on their relative values. The latter functions are distinguished by the suffix `Rel` in their names. For example, `pjclAdd(x,y)` adds the absolute values of the parameters `x` and `y`, while `pjclAddRel` adds their relative values.

5.3 Other data types

A *bit array* is a JavaScript array whose elements are JavaScript floating point numbers with value 0 or 1. Bit arrays are used for encoding the inputs and outputs of hash functions and random bit generators, and sometimes for encoding a nonnegative integer or a sequence of fixed-length nonnegative integers, each integer being mapped to a sequence of bits comprising the *binary representation* of the integer, with the most significant bit being first, i.e. having the lowest array index.

A *hex string* is a JavaScript string whose characters are hexadecimal digits: 0...9, A...F or a...f. Functions that take a hex string as input accept both upper and lower case hexadecimal digits. Functions that produce a hexadecimal string as output use the JavaScript method `toString(16)`, which may produce upper or lower hexadecimal digits depending on the JavaScript engine that interprets the function. JavaScript encodes strings in UTF-16, so each hex digit in a hex string is encoded as a 16-bit UTF-16 character.

An *ASCII string* is a JavaScript string whose characters are ASCII characters. Although ASCII characters can be encoded in 7 bits, they are stored as 16-bit UTF-16 characters in a JavaScript ASCII string.

In a function name such as, for example, `pjclUTF16toBitArray`, the substring `UTF16` is used to refer to a JavaScript string where all 16 bits of the UTF-16 encoding of each character matter, even if all the characters in the string happen to be within the ASCII range of UTF-16 and have UTF-16 encodings whose most significant 9 bits are 0.

An *unsigned 32-bit integer* is a nonnegative integer n in the range $0 \leq n < 2^{32}$, represented in JavaScript as an IEEE double precision floating point number. In function names the substring `UI32` refers to an unsigned 32-bit integer and the substring `UI32Array` to an array of unsigned 32-bit integers. Notice that `UI32Array` refers to an ordinary JavaScript array, not to the *typed array* `UInt32Array`. Typed arrays are not used in PJCL. However, an application that uses typed arrays may pass an argument of type `UInt32Array` to a PJCL function that expects an ordinary JavaScript array of 32-bit unsigned integers.

6 API

This section describes the global variables and functions that comprise the API in the order in which they are declared in `pjcl.js`.

6.1 Argument expectations

When a description of a function states that a parameter is *expected* to have some property, it is an error if the expectation is not met. In `pjcl-withArgChecking.js`, most of the API functions have argument checking code that throws an exception if such expectations are not met.

6.2 Side effects

Functions have no side effects unless otherwise indicated in their documentation. The following functions have side effects in the current version of the library: `pjclShortShiftLeft`, `pjclShiftLeft`, `pjclShortShiftRight`, `pjclShiftRight`, `pjclPreExp` and `pjclPreExp2`.

6.3 Global variables and functions related to the representation of big integers

6.3.1 `var pjclBaseBitLength` `var pjclBase` `var pjclBaseMask` `var pjclBaseMaskMinusOne` `var pjclBaseInv` `var pjclBaseAsBigInt` `var pjclHalfBase`

These global variables encapsulate most of the dependencies on the fact that a limb has 24 bits. Your code should not hardcode the fact that a limb has 24 bits.

- The value of `pjclBaseBitLength` is β, i.e. 24.
- The value of `pjclBase` is B, i.e. 2^{24}, encoded as a JavaScript number.
- The value of `pjclBaseMask` if $B - 1$, encoded as a JavaScript number, which is viewed as

  $$\underbrace{00000000}_{8}\underbrace{111111111111111111111111}_{24}$$

  by JavaScript bitwise operators.
- The value of `pjclBaseMaskMinusOne` is $B - 2$, encoded as a JavaScript number, which is viewed as

  $$\underbrace{00000000}_{8}\underbrace{111111111111111111111110}_{24}$$

  by JavaScript bitwise operators.
- The value of `pjclBaseInv` is $1/B$ encoded as a JavaScript (floating point) number.
- The value of `pjclBaseAsBigInt` is B, encoded as a big integer.

- The value of `pjclHalfBase` is $B/2$, encoded as a JavaScript number, which is viewed as

  $$\underbrace{00000000}_{8}\underbrace{100000000000000000000000}_{24}$$

  by JavaScript bitwise operators.

6.3.2 `function pjclWellFormed(x)`

Returns `true` if the parameter `x` is a well-formed big integer, or `false` otherwise. It is used for argument checking.

6.4 Conversion functions

6.4.1 `function pjclHex2BitArray(s)`

The parameter `s` is expected to be a hex string, which the function converts to a bit array by mapping each hex digit in `s` to the four bits comprising the binary representation of the digit.

6.4.2 `function pjclASCII2BitArray(s)`

The parameter `s` is expected to be an ASCII string, which the function converts to a bit array by mapping each ASCII character in `s` to the eight bits that comprise the 8-bit binary representation of the *ASCII code* of the character. Since an ASCII code is an integer in the range 0. . . 127, the first of the 8 bits is 0. Note that although each character is encoded as a 16-bit UTF character in the JavaScript string `s`, it is mapped to only 8 bits in the resulting bit array.

6.4.3 `function pjclUTF16toBitArray(s)`

The parameter `s` is expected to be a JavaScript string, which the function converts to a bit array by mapping each character to the 16-bit binary representation of its UTF-16 code.

6.4.4 `function pjclByte2BitArray(byte)`

The parameter `byte` is expected to be a JavaScript floating point number whose value is an integer n in the range $0 \leq n < 2^8$, which is converted to a bit array whose elements are the 8 bits of the binary representation of n.

6.4.5 `function pjclUI32toBitArray(ui32)`

The parameter `ui32` is expected to be an unsigned 32-bit integer, i.e. an integer n in the range $0 \leq n < 2^{32}$, which is converted to a bit array whose elements are the 32 bits of the binary representation of n.

6.4.6 `function pjclUI32Array2BitArray(x)`

The parameter `x` is expected to be an array where each element is a JavaScript number whose value is an integer n in the range $0 \leq n < 2^{32}$. The function converts `x` to a bit array by mapping each integer n to the 32 bits of its binary representation. As discussed above in Section 5, PJCL does not use typed arrays, but an application may pass a `UInt32Array` as an argument to the function instead of an ordinary JavaScript array.

6.4.7 `function pjclBigInt2BitArray(x)`

The parameter `x` is expected to be a big integer with mathematical value x. The `negative` property of `x`, if present, is ignored. The function returns a bit array representing the binary encoding of $|x|$ without leading zeros. If $x = 0$ the bit array is empty.

6.4.8 `function pjclBigInt2SizedBitArray(x,size)`

The parameter `x` is expected to be a big integer with value x. The `negative` property of `x`, if present, is ignored. The parameter `size` is expected to be a JavaScript number whose value is a nonnegative integer n. The function returns a bit array of length n. If $|x| < 2^n$, the bit array is the n-bit binary representation of $|x|$ (with leading zero bits as needed). If $|x| >= 2^n$, the bit array is the n-bit binary representation of $|x| \bmod 2^n$.

6.4.9 `function pjclBitLengthOfBigInt(x)`

The parameter `x` is expected to be a big integer with value x. The `negative` property of `x`, if present, is ignored. The function returns the length of the binary representation of $|x|$, i.e. the length of the bit array that would be returned by `pjclBigInt2BitArray(x)`.

6.4.10 `function pjclBitArray2UI32Array(bitArray)`

The parameter `bitArray` is expected to be a bit array of length $32n$. The function returns an array of n 32-bit unsigned integers obtained by partitioning the bit array into groups of 32 bits and viewing each group as the binary representation of a nonnegative integer.

6.4.11 `function pjclBitArray2BigInt(bitArray)`

The parameter `bitArray` is expected to be a bit array of any length. The function returns the nonnegative big integer whose binary representation is the bit array.

6.4.12 `function pjclBitArray2Hex(bitArray)`

The parameter `bitArray` is expected to be a bit array of any length. The function returns a hex string obtained by: (i) prepending leading zero bits to the bit array as needed to make the length of the array a multiple of 4 (notice the difference with `pjclBitArray2UI32Array`, which expects the length of its argument to be a multiple of 32 and does not prepend any leading zeros); and (ii) partitioning the resulting bit array into groups of four bits and viewing each group as the binary representation of a hexadecimal digit; if the hexadecimal digit is greater than nine, whether it is encoded as an upper or lower case letter depends on the implementation of the `toString(16)` method by the JavaScript engine.

6.4.13 `function pjclHex2BigInt(s)`

The parameter `s` is expected to be a hex string. The function returns the big integer having `s` as its hexadecimal representation.

6.4.14 `function pjclBigInt2Hex(x)` `function pjclBigInt2Hex(x,minHexLength)`

The parameter `x` is expected to be a big integer, and the parameter `minHexLength`, if the function is called with two arguments, a JavaScript number whose value is a nonnegative integer. The function returns a hex string consisting of the hexadecimal representation of the big integer, prepended with hexadecimal zero digits as needed to bring its length to the value of `minHexLength`.

6.4.15 `function pjclUI32toHex(x)`

The parameter `x` is expected to be an unsigned 32-bit integer, which is converted to its hexadecimal representation encoded as a hex string.

6.4.16 `function pjclUI32Array2Hex(x)`

The parameter `x` is expected to be an array of unsigned 32-bit integers. The function converts `x` to a hex string by mapping each integer to its hexadecimal representation.

6.5 Basic arithmetic functions

6.5.1 `function pjclGreaterThan(x,y)`

The parameters `x` and `y` are expected to be big integers with mathematical values x and y. Their `negative` properties, if present, are ignored. The function returns `true` if $|x| > |y|$, `false` otherwise.

6.5.2 `function pjclGreaterThanRel(x,y)`

The parameters `x` and `y` are expected to be big integers with mathematical values x and y. The function returns `true` if $x > y$, `false` otherwise.

6.5.3 `function pjclGreaterThanOrEqual(x,y)`

The parameters `x` and `y` are expected to be big integers with mathematical values x and y. Their `negative` properties, if present, are ignored. The function returns `true` if $|x| \geq |y|$, `false` otherwise.

6.5.4 `function pjclGreaterThanOrEqualRel(x,y)`

The parameters `x` and `y` are expected to be big integers with mathematical values x and y. The function returns `true` if $x \geq y$, `false` otherwise.

6.5.5 `function pjclEqual(x,y)`

The parameters `x` and `y` are expected to be big integers with mathematical values x and y. Their `negative` properties, if present, are ignored. The function returns `true` if $|x| = |y|$, `false` otherwise.

6.5.6 `function pjclEqualRel(x,y)`

The parameters `x` and `y` are expected to be big integers with mathematical values x and y. The function returns `true` if $x = y$, `false` otherwise.

6.5.7 `function pjclAdd(x,y)`

The parameters `x` and `y` are expected to be big integers with mathematical values x and y. Their `negative` properties, if present, are ignored. The function returns the nonnegative big integer representing $|x| + |y|$. Thus if $x, y \geq 0$, it simply returns the big integer representing $x + y$.

6.5.8 `function pjclAddRel(x,y)`

The parameters `x` and `y` are expected to be big integers with mathematical values x and y. The function returns the big integer representing $x+y$, which may be negative.

6.5.9 `function pjclSub(x,y)`

The parameters `x` and `y` are expected to be big integers with mathematical values x and y. Their `negative` properties, if present, are ignored. *The function expects that* $|x| \geq |y|$, and returns the nonnegative big integer representing $|x| - |y|$.

6.5.10 `function pjclSubRel(x,y)`

The parameters `x` and `y` are expected to be big integers with mathematical values x and y. The function returns the big integer representing $x-y$, which may be negative.

6.5.11 `var pjclMult` `function pjclMult_Long(x,y)` `function pjclMult_Karatsuba(x,y)`

Big integer multiplication is performed by calling the function `pjclMult(x,y)`. However, there is no definition of that function. Instead, `pjclMult` is a global variable which must be assigned either the function `pjclMult_Long`, which implements long multiplication, or the function `pjclMult_Karatsuba`, which implements Karatsuba multiplication. Both implementations may be used within the same application by assigning different implementations to `pjclMult` at different times.

Both implementations expect the parameters `x` and `y` to be big integers with mathematical values x and y, ignore the `negative` properties of the parameters if present, and return the big integer representing the product of the absolute values of x and y, $|x| \cdot |y|$.

Long multiplication uses an optimized version of the same algorithm that is used for multiplication by hand. Karatsuba multiplication uses the recursive algorithm described in [1, § 15.1.2], carefully implemented for good performance on JavaScript.

The Karatsuba algorithm is asymptotically faster than long multiplication, so it is faster for larger operands but slower for smaller operands. During an execution of the algorithm, recursive calls fall back on long multiplication when the size of the operands becomes less than a *Karatsuba multiplication threshold*. The optimal threshold depends on the platform (machine and JavaScript engine) being used, and can be estimated for a particular machine and engine combination using the tool described below in Section 7. The estimated threshold, expressed as a number of limbs, should be assigned to the global variable `pjclKaratsubaThresholdMult` before using `pjclMult_Karatsuba`. An exception is thrown if `pjclMult_Karatsuba` is called when `pjclKaratsubaThresholdMult` is undefined, but a default is provided to avoid the exception.

6.5.12 `function MultRel(x,y)`

The parameters `x` and `y` are expected to be big integers, with mathematical values x and y. Returns a big integer whose mathematial value is the product

xy.

6.5.13 `function pjclShortMult(x,y)`

The parameter `x` is expected to be a big integer, with mathematical value x, whose `negative` property, if present, is ignored. The parameter `y` is expected to be a JavaScript number whose mathematical value y is an integer in the range $0 \leq y < B = 2^{24}$. The function returns the big integer representing the product $|x| \cdot y$.

6.5.14 `var pjclSqr` `function pjclSqr_Long(x,y)` `function pjclSqr_Karatsuba(x,y)`

Big integer squaring is performed by calling the function `pjclSqr(x)`. Computing `pjclSqr(x)` is faster than computing `pjclMult(x,x)`.

As is the case for big integer multiplication, two implementations of the algorithm are available, which can be selected by assigning either `pjclSqr_Long` or `pjclSqr_Karatsuba` to the global variable `pjclSqr`.

Both implementations expect the parameter `x` to be a big integer with mathematical value x and return the big integer representing x^2. There is a *Karatsuba squaring threshold* analogous to the Karatsuba multiplication threshold. An optimal value of this threshold should be estimated using the tool described below in Section 7 and assigned to `pjclKaratsubaThresholdSqr`, replacing the default, before using `pjclSqr_Karatsuba`.

6.5.15 `function pjclShortShiftLeft(x,k)`

As discussed above in Section 5.2, "shifting left" a big integer means shifting it towards its most significant end, i.e. multiplying it by a power of 2. For performance reasons, `pjclShortShiftLeft` operates by side-effect, modifying its first argument and returning no result; see `pjclMultByPowerOf2` for an alternative without side-effect.

The parameter `x` is expected to be a big integer, possibly negative, with mathematical value x. The parameter `k` is expected to be a JavaScript number whose mathematical value k is a nonnegative integer in the range $0 \leq k < \beta = 24$. The function operates by side-effect, computing the big integer representing $x \cdot 2^k$ and assigning it to `x`.

Although at the API level the parameter `x` is expected to be a big integer, which must not have leading zero limbs, internally, in `pjclDiv`, the function `pjclShortShiftLeft` is used with a first argument that may have leading zero limbs. In `pjcl-withArgChecking` the argument checking code of `pjclShortShiftLeft` throws an exception if `x` has leading zero limbs, which `pjclDiv` catches and cancels.

6.5.16 `function pjclShiftLeft(x,k)` `function pjclMultByPowerOf2(x,k)`

As discussed above in Section 5.2, "shifting left" a big integer means shifting it towards its most significant end, i.e. multiplying it by a power of 2. For performance reasons, `pjclShiftLeft` operates by side-effect, modifying its first argument and returning no result; on the other hand `pjclMultByPowerOf2` is a wrapper that avoids the side-effect, at the cost of a small performance penalty, by making a copy of its first argument before modifying it and returning the result.

The parameter `x` is expected to be a big integer, possibly negative. The parameter `k` is expected to be a JavaScript number whose mathematical value k is a nonnegative integer. The function returns the big integer representing $x \cdot 2^k$. The functions compute the big integer representing $x \cdot 2^k$; `pjclShiftLeft` assigns this big integer to its first argument, while `pjclMultByPowerOf2` returns the result without modifying its arguments.

6.5.17 `function pjclShortShiftRight(x,k)`

This function is analogous to `pjclShortShiftLeft`, shifting towards the least significant rather than the most significant end. It differs from `pjclShortShiftLeft` in that `x` is expected to be nonnegative. Without argument checking, the `negative` property is ignored and `x` may become ill-formed if its negative property is set and it becomes the empty array as a result of the shift.

6.5.18 `function pjclShiftRight(x,k)` `function pjclDivByPowerOf2(x,k)`

These functions are analogous to `pjclShiftLeft` and `pjclMultByPowerOf2`, but like `pjclShortShiftRight` they expect `x` to be nonnegative. They shift towards the least significant end, thus dividing by a power of 2, i.e. computing $\lfloor x/2^k \rfloor$.

6.5.19 `function pjclDiv(x,y)`

The parameter `x` and `y` are expected to be big integers, with mathematical values x and y, whose `negative` properties, if present, are ignored; y must not be zero. The function divides $|x|$ by $|y|$ using Algorithm 14.20 of [2] and returns an object with properties `quotient` and `remainder` whose values are big integer representations of the quotient and the remainder.

6.5.20 `function pjclDivRel(x,y)`

The parameter `x` is expected to be a (relative) big integer with mathematical value x, the parameter `y` a positive big integer with mathematical value y. The function returns an object with properties `quotient` and `remainder` whose values are big integer representations of the quotient and remainder of the division of x by y. The mathematical values q and r of the `quotient` and `remainder` properties are defined as follows: q is the largest (relative) integer such that $qy <= x$, and $r = x - qy$.

6.5.21 `function pjclShortDiv(x,y)`

The parameter `x` is expected to be a big integer, with mathematical value x, whose `negative` property, if any, is ignored. The parameter `y` is expected to be a nonzero limb, i.e. a JavaScript number whose mathematical value y is an integer in the range $0 < y < B = 2^{24}$. Returns an object with a property `quotient` whose value is the big integer representation of the quotient of the division of $|x|$ by y, and a property `remainder` whose value is a JavaScript number representing the remainder.

This function relies on the fact that the JavaScript floating-point % operator is not the same as the "remainder" operation defined by IEEE 754, as explained in [15, §11.5.3].

6.5.22 `function pjclMod(x,m)`

The parameter `x` is expected to be a big integer with mathematical value x, the parameter `m` a positive big integer with mathematical value m. The function returns the big integer representing $x \bmod m$.

6.5.23 `function pjclTruncate(x,t)` `function pjclModPowerOf2(x,t)`

The parameter `x` is expected to be a nonnegative big integer, with mathematical value x and the parameter `t` a JavaScript number whose mathematical value t is a positive integer. Both functions compute the big integer representing $x \bmod 2^t$. For performance reasons, `pjclTruncate` operates by side-effect, modifying its first argument and returning no result; on the other hand `pjclModPowerOf2` is a wrapper that avoids the side-effect, at the cost of a small performance penalty, by making a copy of its first argument before modifying it and returning the result.

Please note that `pjclModPowerOf2` can only be used to reduce a nonnegative integer. You may use `pjclMod` to reduce relative integers, at a much higher computational cost.

6.5.24 `function pjclModLimb(x,m)`

The parameter `x` is expected to be a nonnegative big integer with mathematical value x, the parameter `m` a JavaScript number whose mathematical value m is a positive integer less than B, i.e. less than 2^{24}. Returns a JavaScript number whose mathematical value is $x \bmod m$.

6.5.25 `function pjclEGCD(a,b)` `function pjclEGCD(a,b,computeBothBezoutCoeffs)`

The parameters `a` and `b` are expected to be nonnegative big integers with mathematical values a and b. If the function is called with three arguments and `computeBothBezoutCoeffs` is or type-converts to `true`, the function implements the Extended Euclidean Algorithm and returns an object with properties `gcd`, `x` and `y` whose mathematical values are d, x and y, where d is the greatest common divisor of a and b, and (x, y) is a pair of integers, called Bézout coefficients, that satisfy $d = ax + by$. If only two arguments are passed to the function, y is not computed and the object returned by the function does not have `y` property.

6.5.26 `function pjclModInv(x,m)`

The parameter `x` is expected to be a big integer with mathematical value x, the parameter `m` a positive big integer with mathematical value m. The function returns `undefined` if x and m are not coprime. Otherwise it returns a big integer whose mathematical value is the inverse of x modulo m.

6.6 Montgomery reduction

Our implementation of Montgomery reduction is based on Section 14.3.2 of the Menezes et al. Handbook of Applied Cryptography [2]. More specifically, it is based on the optimzed Algorithm 14.32, further optimized and adapted for use with our big integer representation.

In this Section 6.6 we use the same mathematical variables as in algorithm 14.32, except that we write B instead of b, since $B = 2^\beta = 2^{24}$ is the base of our representation of big integers, as defined in Section 5.2.

Thus m is the modulus, which must be coprime with B, i.e. odd; n is the number of limbs of the big integer representation $(m_{n-1} \ldots m_1, m_0)_B$ of m; $R = B^n$; $m' = -m^{-1} \bmod B$; and T is the nonnegative integer to be reduced, which must be less than mR and therefore have a big integer representation with no more than $2n$ limbs.

In our implementation, the big integer representation of m must have at least two limbs. This is not required by algorithm 14.32, but it it is required by our further optimization of the algorithm. For one-limb moduli you may use ordinary modular reduction as provided by `pjclModLimb`.

Montgomery reduction is much faster than ordinary modular reduction, but instead of computing $T \bmod m$, it computes $TR^{-1} \bmod m$. It is intended to be used in an algorithm that requires many multiplications (and/or squarings), such as modular exponentiation. All quantities in the algorithm are modified to incorporate the factor R. Instead of multiplying x by y to obtain $z = xy$ and then reducing z modulo m, the modified algorighm multiplies xR by yR to obtain $(xR)(yR)$, then uses Montgomery reduction to compute $(xR)(yR)R^{-1} = xyR = zR$. zR can then be further multiplied by uR and Montgomery-reduced to produce vR with $v = zu$, and so on.

Our implementation includes a function `pjclPreMontRed` that precomputes m' and a function `pjclMontRed` that computes the Montgomery reduction of T modulo m using m'.

6.6.1 `function pjclPreMontRed(m)`

The parameter `m` is expected to be an odd positive big integer with at least two limbs, whose mathematical value is the modulus m. The function returns a JavaScript number whose mathematical value is $m' = -m^{-1} \bmod B$.

6.6.2 `function pjclMontRed(t,m,m1)`

The parameter `t` is expected to be a nonnegative big integer, the parameter `m` an odd big integer having at least two limbs, and `m1` a JavaScript number whose mathematical value is $m' = -m^{-1} \bmod B$, as returned by `pjclPreMontRed(m)`. The mathematical values T of `t` and m of `m` must satisfy $T < mR$ with $R = B^n$, where n is the number of limbs of the modulus. The function returns a big integer with mathematical value $TR^{-1} \bmod m$.

6.7 Generic sliding window exponentiation in a monoid

6.7.1 `function pjclOptimalWindowSize(l)` `function pjclPreExp(slidingWindowSize,context)` `function pjclExp(exponent,context)`

The function `pjclExp(exponent,context)` implements generic sliding window exponentiation in some monoid M using a slightly optimized version of Algorithm 14.85 of [2]. In this section we refer to the monoid operation as multiplication, but `pjclExp` can be used, and we do use it in this version of the library, to implement scalar multiplication in monoids where the operation is usually written as addition;[1] `pjclExp` is used by `pjclPlainExp` to implement exponentiation in $\mathbb{N}$, by `pjclModExp` to implement modular exponentiation with ordinary reduction, by `pjclMontExp` to implement modular exponentiation with Montgomery reduction, and, as desribed below in Section 6.20.13, by `pjclScalarMult` to implement scalar multiplication in the group of points of an en elliptic curve. (In a future version of the library we plan to implement a sliding window exponentiation function further optimized for groups by using nonadjacent form (NAF) to represent the exponent, and use it to implement `pjclScalarMult`, taking adantage of the fact that the points of an elliptic curve form a group and point subtraction can be implemented efficiently.)

The parameter `exponent` of `pcjlExp` is expected to be a big integer whose mathematical value is a positive integer e. (We exclude the case

[1] "Scalar multiplication" and "exponentiation" are alternative names given to the same external operation in a monoid, the term "scalar multiplication" being used when the operation is called "addition" while the term "exponentiation is used when the operation is called "multiplication".

$e = 0$, where the function would return the unit of the monoid, but the functions that call `pjclExp`, i.e. `pjclPlainExp`, `pjclModExp`, `pjclMontExp` and `pjclScalarMult`, take care of this special case). The parameter `context` is expected to be an object with a property `context.g` specifying the base $g \in M$ of the exponentiation, whose encoding depends on the nature of M. The function returns an encoding of the element g^e of M.

The parameter `context` must also have a method `context.mult` implementing the monoid operation, a method `context.sqr` such that `context.sqr(x)` produces the same result as `context.mult(x,x)`, and a property `context.preComputed` whose value must be an array providing the results of the precomputation that takes place at step 1 of Algorithm 14.85. It may also have additional properties specific to a particular monoid, such as `context.m`, whose value is the modulus m, when `pjclExp` is called by p̃jclModExp or `pjclMontExp`, and `context.m1`, whose value is $m' = -m^{-1} \bmod B$ where $B = 2^\beta = 2^{24}$ when it is called by `pjclMontExp`.

The function `pjclPreExp(slidingWindowSize,context)` is a side-effect function that performs the precomputation of step 1 of Algorithm 14.85. The parameter `slidingWindowSize` is expected to be a Javascript number whose mathematical value is a positive integer, called k in the algorithm, to be used as the window size. The parameter `context` is expected to be an object with the above-mentioned properties and methods `context.g`, `context.mult` and `context.sqr`. The function creates and fills the array `context.preComputed`. It does not return a result.

The function `pjclOptimalWindowSize(l)` gives the optimal window size for a given exponent size. The parameter `l` is expected to be a JavaScript number whose mathematical value is a positive integer that should be the approximate bitlength of the exponent. The function retuns a JavaScript number whose mathematical value is the optimal window size.

6.8 Exponentiation in $\mathbb{N}$

6.8.1 `function pjclPlainExp(g,x)`

The function `pjclPlainExp(g,x)` performs exponentiation in the monoid $(\mathbb{N}, +)$. The parameters `g` and `x` are expected to be nonnegative big integers with mathematical values g and x. The function returns the big integer representation of g^x.

Notice that the result of this function will be unmanageable if the exponent has more than one limb: if `g` and `x` have big integer representations [2] and [0,1], with mathematical values $g = 2$ and $x = 2^{24}$, then the result of the function should have the mathematical value $2^{2^{24}}$, whose big integer representation has 3,659,183 limbs.

6.9 Modular exponentiation with ordinary reduction

6.9.1 function pjclModExp(g,x,m)

The function `pjclModExp(g,x,m)` performs exponentiation in the monoid $(\mathbb{Z}_m, \times)$, where m is the mathematical value of the parameter `m`, and $\mathbb{Z}$ is the set of integers modulo m. The parameters `g`, `x` and `m` are expected to be big integers with mathematical values $g \geq 0$, $x \geq 0$ and $m \geq 1$. The function returns the big integer representation of $g^x \bmod m$.

Although `pjclModExp` does not produce unmanageable results like `pjclModExp`, it is too slow to be used in most cryptographic applications.

6.10 Modular exponentiation with Montgomery reduction

6.10.1 function pjclMontExp(g,x,m)

The function `pjclMontExp(g,x,m)` produces the same result as `pjclModExp( g,x,m)`, but using Montgomery reduction rather than ordinary reduction, which makes it fast enough to be used in cryptographic applications.

The parameters `g` and `x` are expected to be nonnegative big integers, with mathematical values g and x. The parameter `m` is expected to be a nonnegative big integer with $n \geq 2$ limbs whose mathematical value m is odd.

Recall that $B = 2^\beta = 2^{24}$ was defined in Section 6.6 as the base of the big integer representation. Let $R = B^n$. Using Montgomery reduction amounts to performing the exponentiation in the isomorphic image of the monoid $(\mathbb{Z}_m, \times)$ by the function ϕ_R that maps $u \in \mathbb{Z}_m$ to uR. If we call $*_R$ the operator of the image monoid, the product $uR *_R vR$ of two elements of $\phi_R(\mathbb{Z}_m)$ is $uvR \bmod m$, which is computed in two steps by first multiplying uR and vR to obtain uvR^2 then performing a Montgomery reduction to obtain $(uvR^2)R^{-1} \bmod m = uvR \bmod m$.

`pjclMontExp` assigns the big integer representation of gR to `context.g` and uses `pjclExp` to raise gR to x in the image monoid by performing multiplications followed by Montgomery reduction using `pjclContextualMontMult` and squarings followed by Montgomery reduction using `pjclContextualMontSqr`. The result $g^xR \bmod m$ is converted to $g^x \bmod m$ by one final Montgomery reduction.

6.11 Generic double exponentiation in a commutative monoid

6.11.1 `function pjclOptimalWindowSize2(l)` `function pjclPreExp2(slidingWindowSize,context)` `function pjclExp2(exponentG,exponentY,context)`

These functions are like those of Section 6.7.1, with the difference that `pjclExp2` computes the product of two exponentials, with exponents `exponentG` and `exponentY` and corresponding bases `context.g` and `context.y`, using "Shamir's trick" of combining the squarings of the two exponentiations. Either exponent, but not both, may be (the big integer) zero. In this version of the library, `pjclExp2` is used by `pjclMontExp2` and `pjclScalarMult2`. The array `context.preComputed` computed by `pjclPreExp2` as a side-effect is doubly indexed, and `pjclOptimalWindowSize2` computes the optimal window size for double exponentiation, taking as input the bit length of the longest of the two exponents.

6.12 Double exponentiation with Montgomery reduction

6.12.1 `function pjclMontExp2(g,y,exponentG,exponentY,m)`

The function pjclMontExp2(g,y,exponentG,exponentY,m) produces the same result as

```
pjclMod(pjclMult(pjclMontExp(g,exponentG,m),pjclMontExp(y,exponentY,m)),m)
```

but substantially faster, using `pjclExp2`.

6.13 Hash functions and message authentication codes (SHA, HMAC)

6.13.1 `function pjclSHA256(bitArray)`

The function `pjclSHA256` takes as input a sequence of bits encoded as a bit array and returns a bit array that encodes the result of applying the function SHA-256 of [16] to the input.

6.13.2 `function pjclSHA384(bitArray)`

The function `pjclSHA384` takes as input a sequence of bits encoded as a bit array and returns a bit array that encodes the result of applying the function SHA-384 of [16] to the input.

6.13.3 `function pjclHMAC_SHA256(key,text)`

The function `pjclHMAC_SHA256` implements the HMAC algorithm of [7] in conjunction with the hash function SHA-256 of [16]. The parameters `key` and `text` are expected to be bit arrays, and the result is a bit array.

6.13.4 `function pjclHMAC_SHA384(key,text)`

The function `pjclHMAC_SHA384` performs an HMAC computation using the hash function SHA-384 instead of SHA-256.

6.13.5 `function pjclHMAC_SHA256PreComputeKeyHashes(key)` `function pjclHMAC_SHA256WithPreCompute(iKeyHash,oKeyHash,text)`

An HMAC computation consists of two hash computations, and the first block of each computation does not depend on the text. When you need to perform many HMAC computations with the same key, you can use `pjclHMAC_SHA256PreComputeKeyHashes(key)` to precompute the hashes of those two blocks. The result is an object with properties `iKeyHash` and `oKeyHash`, whose values you can pass as arguments to `pjclHMAC_SHA256WithPreCompute(iKeyHash,oKeyHash,text)` to obtain the value of the HMAC computation for each `text`.

6.13.6 `function pjclHMAC_SHA384PreComputeKeyHashes(key)` `function pjclHMAC_SHA384WithPreCompute(iKeyHash,oKeyHash,text)`

The functions `pjclHMAC_SHA384PreComputeKeyHashes` and `pjclHMAC_SHA384WithPreCompute` perform a split HMAC precomputation like `pjclHMAC_SHA256PreComputeKeyHashes` and `pjclHMAC_SHA256WithPreCompute` using the hash function SHA-384 instead of SHA-256.

6.14 Statistically random data vs. cryptographically random data

We make a distinction between statistically random d[ata and cryptographically random data. We say that data produced by a data source is *statistically random* if it is uniformly distributed over a given range but may be predictable from data previously generated by the source. By contrast we say that data produced by a data source is *cryptographically random* if it is uniformly distributed and unpredictable from data previously generated by the source.

We use the built-in JavaScript function `Math.random` to generate statistically random data, and a pseudo-random bit generator implemented as specified in [8, § 10.1.1] to generate cryptographically random data. `Math.random` is well suited for generating statistically random data because its output is specified as having an approximately uniform distribution [15, 15.8.2.14]. It must not be used to generate cryptographically random data, or to seed or reseed the random bit generator, because its output may be predictable.

6.15 Random bit generation (RBG) vs. random number generation (RNG)

We make a distinction between random bit generation and random number generation. Generating l random bits is equivalent to generating a random number n in the range $0 \leq n < 2^l$. We use the term *random bit generation (RBG)* to refer to the generation of random bits or to the generation of a number in such a range. On the other hand we use the term *random number generation (RNG)* to refer to the generation of a random number n in a range $a \leq n < b$, where a may not be zero and $b - a$ may not be a power of two.

6.16 Generation of statistically random data

6.16.1 `function pjclStRndLimb()`

The function `pjclStRndLimb` takes no arguments and returns a statistically random JavaScript number that can serve as big integer limb, i.e. whose mathematical value n is an integer in the range $0 \leq n < B = 2^\beta = 2^{24}$.

6.16.2 `function pjclStRndBigInt(n)`

The parameter `n` is expected to be a JavaScript number whose mathematical value is a nonnegative number n. The function returns a statistically random big integer with up to n limbs, i.e. whose mathematical value x is uniformly distributed in the range $0 \leq x < B^n$.

6.16.3 `function pjclStRndHex(n)`

The parameter `n` is expected to be a JavaScript number whose mathematical value is a nonegative number n. The function returns a hex string consisting of n statistically random hex digits. Whether hex digits greater than 9 are in upper or lower case depends on the implementation of the `toString(16)` method by the JavaScript engine.

6.16.4 `function pjclStatisticalRNG(a,b)`

The parameters `a` and `b` are expected to be big integers with mathematical values a and b such that $0 \leq a < b$. The function returns a statistically random big integer whose mathematical value x is uniformly distributed in the range $a \leq x < b$.

6.17 Cryptographic random number generation

The functions in this section implement a deterministic random bit generator (DRBG) based on hash functions. More specifically, they implement the *Hash_DRBG mechanism* of of [8, § 10.1.1], using the hash functions SHA-256 for 128 bits of security strength and SHA-384 for 192 bits of security strength.

6.17.1 `function pjclRBG128Instantiate(rbgStateStorage,entropy)` `function pjclRBG128Instantiate(rbgStateStorage,entropy,nonce)`

This function instantiates the DRBG based on SHA-256 specified in Section 10.1.1.2 of [8]. No `personalization_string` is used.

The parameter `rbgStateStorage` may be a JavaScript object or, in a JavaScript runtime environment that implements the *W3C Web Storage* specification [17], a *storage object*, i.e. either `localStorage` or `sessionStorage`.

The parameter `entropy` is expected to be an array of at least 128 bits. An exception is thrown otherwise by both the argument checking and the production versions of the library. However this is only a sanity check, since there is no way for the function to know if the value of the parameter has *full entropy*. (A bit string is said to have full entropy if its entropy is equal to its length.)

Do not use `Math.random` to generate the value of the `entropy` parameter. In a JavaScript runtime environment that implements the *Web Cryptography API* you may use `crypto.getRandomValues()` to generate entropy; notice, however, that the Web Cryptography API does not explicitly guarantee that the output of `crypto.getRandomValue()` has full entropy. Examples of how to do this are provided by two functions `pjclBrowserEntropy128Bits` and `pjclBrowserEntropy192Bits`, which can be found in the file `browserEntropy.js`. The first of them is used for DSA performance testing. Notice, however, that the Web Cryptography API does not explicitly guarantee that the output of `crypto.getRandomValue()` has full entropy. In a JavaScript runtime environment that provides access to an underlying Unix-like OS you may use `/dev/random`, which is supposed to provide full entropy but may block if not enough entropy is available, or `/dev/urandom`, which does not block but is not guaranteed to provide full entropy. A web application may want to download entropy from the back-end to the front-end if a source of full entropy is available on the back-end.

The parameter `nonce` is also expected to be a bit array, but it is optional. (The use of this input is motivated in Section 8.6.7 of [8].) If no value is supplied, the function uses a value derived from `Data.getTime()`.

The function instantiates the DRBG by storing its initial internal state in three properties of `rbgStateStorage`, `rbgStateStorage.pjclRBG128_v`, `rbgStateStorage.pjclRBG128_c` and `rbgStateStorage.pjclRBG128_reseed_counter`. The function does not return a value.

6.17.2 `function pjclRBG128Reseed( rbgStateStorage, entropy)`

This function reseeds a DRBG based on SHA-256 as specified in Section 10.1.1.3 of [8]. No `additional_input` is used.

The parameter `rbgStateStorage` is expected to be a JavaScript object or storage object (`localStorage` or `sessionStorage`) containing the internal state of a DRBG in three properties `rbgStateStorage.pjclRBG128_v`, `rbgStateStorage.pjclRBG128_c` and `rbgStateStorage.pjclRBG128_reseed_counter`.

As in `pjclRBG128Instantiate`, the parameter `entropy` is expected to be an array of at least 128 bits, and an exception is thrown otherwise by both the argument checking and the production versions of the library. The function updates the internal state of the DRBG at `rbgStateStorage` and returns no value.

6.17.3 `function pjclRBG128InstantiateOrReseed( rbgStateStorage,entropy,nonce)`

This is a convenience function that uses `entropy` and `nonce` to initialize a DRBG at `rbgStateStorage` unless one already exists there, in which case the existing DRBG is reseeded using both `entropy` and `nonce` as entropy inputs. The parameters entropy and nonce are expected to be bit arrays.

6.17.4 function pjclRBG128Gen(rbgStateStorage, bitLength)

This function generates bits from a DRBG based on SHA-256 as specicified in Section 10.1.1.4 of [8].

The parameter `rbgStateStorage` is expected to be a JavaScript object or storage object (`localStorage` or `sessionStorage`) containing the internal state of a DRBG in three properties `rbgStateStorage.pjclRBG128_v`, `rbgStateStorage.pjclRBG128_c` and `rbgStateStorage.pjclRBG128_reseed_counter`.

The `bitLength` parameter is expected to be a JavaScript number specifying the number of bits to be returned, whose mathematical value must be a positive integer no greater than 2^{19} according to Table 2 of [8]. The function throws an exception otherwise.

The function returns a bit array with the specified number of bits.

6.17.5 `function pjclCryptoRNG128(rbgStateStorage,a,b)`

The parameter `rbgStateStorage` is expected to be a JavaScript object or storage object (`localStorage` or `sessionStorage`) containing the internal state of a DRBG in three properties `rbgStateStorage.pjclRBG128_v`, `rbgStateStorage.pjclRBG128_c` and `rbgStateStorage.pjclRBG128_reseed_counter`. The parameters `a` and `b` are expected to be big integers with mathematical values a and b such that $0 \leq a < b$. The function returns a cryptographically random big integer whose mathematical value x is uniformly distributed in the range $a \leq x < b$. To ensure a quasi-uniform distribution, the function uses the "extra random bits" method used in Section B.1.1 of [4] for key pair generation and in Section B.2.1 for per-message secret number generation.

6.17.6 `function pjclRBG192Instantiate(rbgStateStorage,entropy,nonce)` `function pjclRBG192Reseed(rbgStateStorage,entropy)` `function pjclRBG192InstantiateOrReseed(rbgStateStorage,entropy,nonce)` `function pjclRBG192Gen(rbgStateStorage,bitLength)` `function pjclCryptoRNG192(rbgStateStorage,a,b)`

The functions `pjclRBG192*` and `pjclCryptoRNG192` are like the corresponding functions `pjclRBG128*` and `pjclCryptoRNG128` except that they use SHA-384 as the hash function and accordingly provide 192 bits of security strength. The value of the `entropy` parameter in `pjclRBG192Instantiate`, `pjclRBG192Reseed` and `pjclRBG192InstantiateOrReseed` must be a bit array of length at least 192.

6.18 Primality testing

6.18.1 `function pjclIsPrime(n,t)` `function pjclMillerRabin(n,t)`

The function `pjclIsPrime` performs a probabilistic primality test on a big integer `n`, using the Miller-Rabin test if the big integer has more than one limb, and checking for divisibility by a 12-bit prime if it has only one limb. This is one place in the library where the number of bits per limb is hardwired.

The function `pjclMillerRabin`, which is called by `pjclIsPrime`, implements the Miller-Rabin probabilistic primality test as described in Algorithm 4.42 of [2] with a number of repetitions specified by the parameter `t`. In cryptographic applications the number to be tested is usually cryptographically random, but the potential witnesses to compositness only need to be statistically random, so the function `pjclIsPrime` uses the function `pjclStatisticalRNG` to generate witnesses.

In both functions the parameter `n` is expected to be a nonnegative integer and the parameter `t` a JavaScript number whose mathematical value is a positive integer. In `pjclMillerRabin` the parameter `n` must have two limbs and be odd.

6.19 DSA

This version of the library provides an implementation of DSA [4] with only one security strength, 128 bits, which requires a 256-bit private key and a 3072-bit public key according to Table 2 of [9]. Higher security strengths are deemed impractical by NIST according to Table 2.

6.19.1 `function pjclDSAGenPQ(domainParameterSeed)` `function pjclDSAGenPQ()`

The function `pjclDSAGenPQ` generates probable primes p and q of bit lengths $L = 3072$ and $N = 256$ respectively, with q dividing $p - 1$, as specified in Section A.1.1.2 of [4] using SHA-256 as the hash function, Miller-Rabin with 64 repetitions as the probabilistic primality test, and a seed length of 256 bits.

The algorithm of A.1.1.2 is non-deterministic: a domain parameter seed with the specified seed length is chosen at step 5, then a deterministic attempt at generating a probable prime q is made, going back to step 5 if the attempt fails. Once an attempt at generating q succeeds, a deterministic attempt at generating a probable prime p such that q divides $p - 1$ is made, going back to step 5 if the attempt fails. The algorithm returns p, q, the last domain parameter seed chosen at step 5 and a counter. The returned values can be used in algorithm A.1.1.3 to validate prime numbers p and q if generated by a non-trusted party. We may provide an implementation of algorithm A.1.1.3 in a future verion of the library.

The optional parameter `domainParameterSeed` is expected to be a bit array, which can be chosen arbitrarily and is used as the initial domain parameter seed of step 5 of the NIST algorithm. If not supplied, a bit array with 256 statistically random bits is used.

The function returns an object with properties `p`, `q`, `domainParameterSeed` and `counter` corresponding to the values returned by Algorithm A.1.1.2, `p` and `q` being big integers, `domainParameterSeed` being a bit array, and `counter` being a JavaScript number.

6.19.2 `function pjclDSAGenG(p,q,domainParameterSeed,index)`

The function `pjclDSAGenG` produces a big integer whose value is the generator g as specified in Section A.2.3 of [4] using SHA-256 as the hash function. The values of the parameters `p`, `q`, `domainParameterSeed` must be the corresponding properties of the object returned by `pjclDSAGenPQ`, and `index` must be a bit array of length 8, which may be chosen arbitrarily. With argument checking, the function verifies that `p` and `q` are well-formed nonnegative big integers with bit lengths are 3072 and 256 and `domainParameterSeed` and `index` are bit arrays of lengths 256 and 8 respectively.

6.19.3 `function pjclDSAGenPQG(domainParameterSeed,index)`

Calls `pjclDSAGenPQ` then `pjclDSAGenG`. Returns an object with the properties `p`, `q`, `domainParameterSeed` and `counter` returned by `pjclDSAGenPQ`, and a property `g` whose value is the big integer returned by `pjclDSAGenG`.

6.19.4 `function pjclDSAGenKeyPair(rbgStateStorage,p,q,g)` `function pjclDSAGenKeyPair(rbgStateStorage)`

Generates a DSA key pair (x, y) with 128 bits of security strength, i.e. with a 256-bit private key x and a 3072-bit public key y, as specified in Section B.1.1 of [4].

The parameter `rbgStateStorage` is expected to be a JavaScript object or storage object (`localStorage` or `sessionStorage`) containing the internal state of a DRBG in three properties `rbgStateStorage.pjclRBG128_v`, `rbgStateStorage.pjclRBG128_c` and `rbgStateStorage.pjclRBG128_reseed_counter`.

The optional parameters `p`, `q` and `g` are expected to be domain parameters such as returned by `pjclDSAGenPQG`. With argument checking, besides verifying that `p`, `q` and `g` are well-formed nonnegative big integers, the function verifies that the bit lengths of `p` and `q` are 3072 and 256 respectively.

The function returns an object with properties `x` and `y`, whose values are the big integer representations of x and y.

6.19.5 `function pjclDSASign(rbgStateStorage,p,q,g,x,msg)`

The function `pjclDSASign` computes a DSA signature with 128 bits of security strength on the parameter `msg` as described in Section 4.6 of [4].

The internal variable `k` is the per-message secret, or nonce, whose value k is generated cryptographically at random using `pjclCryptoRNG128`, which provides 128-bits of security strength assuming that it is seeded with 128 bits of entropy. The function should not be modified to take k as an argument; k should not be precomputed, because that may facilitate a timing attack, as discusssed in Section **??**; and k must be treated as a secret, because the private key can be computed from k and the signature.

The parameter `rbgStateStorage` is expected to be a JavaScript object or storage object (`localStorage` or `sessionStorage`) containing the internal state of a DRBG in three properties `rbgStateStorage.pjclRBG128_v`, `rbgStateStorage.pjclRBG128_c` and `rbgStateStorage.pjclRBG128_reseed_counter`. The parameters `p`, `q` and `g` are expected to be nonnegative big integers, representing the domain parameters as generated by `pjclDSAGenPQG`. The parameter `x` is expected to be a big integer, representing the private key, as generated by `pjclDSAGenKeyPair`. The parameter `msg` is expected to be a bit array, which is hashed using

SHA-256.

The function returns an object with properties `r` and `s` whose values are big integer representations of the components of the signature (r, s).

6.19.6 `function pjclDSAVerify(p,q,g,y,msg,r,s)`

The function `pjclDSAVerify` verifies a signature as described in Section 4.7 of [4]. The parameters `p`, `q` and `g` are expected to be big integers, representing the domain parameters. The parameter `y` is expected to be a big integer, representing the public key. The parameter `msg` is expected to be a bit array representing the message. The parameters `r` and `s` are expected to be big integers, representing the components of the signature. The function returns `true` if verification succeeds, `false otherwise`.

6.20 Elliptic curves

6.20.1 NIST curves

The Digital Signature Standard of NIST specifies five elliptic curves over prime fields [4, § D.2]: P-192, P-224, P-256, P-384 and P-521. Descriptions of these curves can also be found in [10, §10.2], [18] and [19]. This version of the library implements ECDSA on curves P-256 and P-384. Other NIST and non-NIST curves may be supported in future versions.

The term "Weierstrass equation" is defined with various degrees of generality. Here we shall use the term to refer to an equation of the form $y^2 = x^3 + ax + b$ over a field F, where $a, b \in F$ are constants such that $4a^3 + 27b^2 \neq 0$. We shall refer to a curve with a Weierstrass equation as a Weierstrass curve. Here we shall only be concerned with Weierstrass curves over a prime field $F = \mathbb{F}_p$.

NIST curves over prime fields have Weierstrass equations where the coefficient a is -3. An explanation of the motivation for choosing $a = -3$ can be found in [11, § 2.6.2]. This version of the library hardcodes the fact that $a = -3$.

The domain parameters for ECDSA on a Weierstrass curve over a prime field $\mathbb{F}_p$ include, in addition to p, a, and b, the choice of a base point G. The base point is a point of prime order n, i.e. a point that generates a subgroup of order n of the group $E(\mathbb{F}_p)$ of points of the curve. By Lagrange's theorem, n divides the order $\#E(\mathbb{F}_p)$ of (the group of points of) the curve. The quotient

$h = \#E(\mathbb{F}_p)/n$, called the cofactor, is another domain parameter. In all the NIST curves over prime fields the order of the curve is a prime number, and therefore the cofactor is 1.

NIST [4, § D.2] suggests taking advantage of the fact that the primes p in the five curves over prime fields are Generalized Mersenne Primes whose exponents are multiples of 32 in order to improve the performance of reduction modulo p. However the suggested method is only suitable for big integer representations with 32-bit limbs. But those primes are also Pseudo-Mersenne Primes (see Section 6.20.7) and reduction modulo a Pseudo-Mersenne prime can be performed using Algorithm 14.47 of [2] with about the same performance improvement [20]. This is what the library does.

6.20.2 Affine vs. projective vs. Jacobian coordinates

(This section can be skipped without loss of continuity.)

An elliptic curve has a "point at infinity" that cannot be represented in affine coordinates, but can be represented in projective coordinates or, preferably for performance reasons, in Jacobian coordinates.

A point with affine coordinates (X, Y) in a two-dimensional space over a field F has projective coordinates (x, y, z) such that $z \neq 0$, $x = Xz$ and $y = Yz$, which are the coordinates in the three-dimensional space of the points of the line containing the origin and the point $(X, Y, 1)$, excluding the origin. On the other hand the projective coordinates of a point at infinity are the coordinates of the points of a line that goes through the origin and lies in the plane $z = 0$, again not including the origin itself, i.e. there are the triples (x, y, z) such that $z \neq 0$ and $ax + by = 0$ for some $a, b \in F$ not both equal to zero.

A line with equation $aX + bY + c = 0$ in affine coordinates has equation $a\frac{x}{z} + b\frac{y}{z} + c = 0$, $z \neq 0$ in projective coordinates, which becomes $ax+by+cz = 0$ when the point at infinity of the line is included. The projective coordinates of the point at infinity are obtained by making $z = 0$ but $x, y \neq 0$ in the equation, i.e. they are the triples $(x, y, 0)$ other than the origin $(0, 0, 0)$ such that $ax + by = 0$.

An elliptic curve with affine equation $Y^2 = X^3 + aX + b$ has a projective equation $\frac{y^2}{z^2} = \frac{x^3}{z^3} + a\frac{x}{z} + b$, $z \neq 0$, which becomes $y^2 z = x^3 + axz^2 + bz^3$ when completed with the point at infinity. The projective coordinates of the point at infinity of the ellipical curve are obtained by making $z = 0$ but $x, y \neq 0$ in the equation, i.e. they are the triples $(x, y, 0)$ other than $(0, 0, 0)$ such that

$x^3 = 0$, which implies $x = 0$.

A point with affine coordinates (X, Y) has Jacobian coordinates (x, y, z) such that $z \neq 0$, $x = Xz^2$ and $y = Yz^3$, while a point at infinity in Jacobian space has the set of coordinates (x, y, z) such that $z \neq 0$ and $ax^3 + by^2 = 0$ for some $a, b \in F$ not both equal to zero.

An elliptic curve with affine equation $Y^2 = X^3 + aX + b$ has a projective equation $\frac{y^2}{z^6} = \frac{x^3}{z^6} + a\frac{x}{z^2} + b$, $z \neq 0$, which becomes $y^2 = x^3 + axz^4 + bz^6$ when completed with the point at infinity. The Jacobian coordinates of the point at infinity of the elliptical curve are obtained by making $z = 0$ but $x, y \neq 0$ in the equation, i.e. they are the triples $(x, y, 0)$ other than $(0, 0, 0)$ such that $y^2 = x^3$.

6.20.3 Jacobian representation of a point

In the library, a point of an elliptic curve is represented in Jacobian coordinates, as a JavaScript object with three properties `x`, `y` and `z` whose values are big integers representing the Jacobian coordinates x, y and z of the point. We shall refer to such an object as *a Jacobian representation* of the point.

6.20.4 Affine representation as a special case of Jacobian representation

If $(x, y, 1)$ are Jacobian coordinates of a point P, then (x, y) are its affine coordinates. In the library, the *affine representation* of a finite point is a special case of a Jacobian representation where the value of the `z` property is the big integer representation of 1, i.e. [1]. The function `pjclJacobian2Affine` produces that affine representation.

6.20.5 Jacobian-affine optimization of point addition

The function `pjclPointAdd` takes as arguments two Jacobian representations, but checks if the second one is an affine representation and optimizes that special case.

6.20.6 `function pjclModSpecial(x,t,xc,m)`

The function `pjclModSpecial` computes $x \bmod m$, where $m = 2^t - c$, using Algorithm 14.47 of [2], which is applicable when $0 < c < 2^{t-1}$ and efficient when c is "small" compared to 2^{t-1}, which we shall write $c \ll 2^{t-1}$.

The parameter `x` is expected to be a nonnegative big integer representing the integer x to be reduced. The parameter `t` is expected to be a JavaScript number representing the exponent t, which must be a positive integer. The parameter `xc`, read "times c", is expected to be a function that takes as its only argument a positive big integer and returns a big integer representing its product by c; different such functions can be written and optimized for different values of c. The parameter `m` is expected to be a positive big integer representing the modulus $m = 2^t - c$. The function returns a big integer representing $x \bmod m$.

In this version of the library, the function `pjclModSpecial` is used to compute reductions modulo Pseudo-Mersenne primes. Note, however, that `pjclModSpecial` can also be used in cases where m is not a prime.

6.20.7 Pseudo-Mersenne representation of a prime

A Pseudo-Mersenne Prime is a prime of the form $p = 2^t - c$ with $0 < c \ll 2^{t-1}$. Modular reduction by such a prime p can thus be sped up by using `pjclModSpecial` instead of `pjclMod`. A *Pseudo-Mersenne representation* of p is a triple of JavaScript values consisting of the JavaScript number representing t, a function that multiplies a big integer by c, and the big integer representation of p suitable to be passed as second, third and fourth arguments to `pjclModSpecial`.

6.20.8 `var pjclCurve_P256`

The value of the global variable `pjclCurve_P256` is an object whose properties describe the ECDSA domain parameters for NIST curve P-256, which is the curve with equation

$$y^2 = x^3 - 3x^2 + b$$

over prime field $\mathbb{F}_p$, where[2]

$$p = 2^{256} - 2^{224} + 2^{192} + 2^{96} - 1$$

and b has the big integer representation shown in the code as the value of the property `b`. The prime p can be written $p = 2^t - c$ with

$$c = 2^{224} - 2^{192} - 2^{96} + 1$$

[2]A typo in the value of p has been fixed after release. The error was only in the documentation. The code was correct.

The object has the following properties and methods:

- Three properties `t`, `xc` and `p` comprising the Pseudo-Mersenne representation of the prime p.
- A property `b` whose value is a big integer representing the coefficient b of the curve.
- A property `n` whose value is a big integer representing the order n of the curve, which is also the order of the base point, since the cofactor is 1.
- A property `G` whose value is the affine representation of the base point of the curve. (Recall that, in the library, an affine representation is a special case of a Jacobian representation, as explained in Section 6.20.4.)

6.20.9 `var pjclCurve_P384`

The value of the global variable `pjclCurve_P384` is an object whose properties describe the ECDSA domain parameters for NIST curve P-384, which is the curve with equation

$$y^2 = x^3 - 3x^2 + b$$

over prime field $\mathbb{F}_p$, where

$$p = 2^{384} - 2^{128} - 2^{96} + 2^{32} - 1$$

and b has the big integer presentation shown in the code as the value of the property `b`. The prime p can be written $p = 2^t - c$ with

$$c = 2^{128} + 2^{96} - 2^{32} + 1$$

The object has properties and methods like those of `pjclCurve_P256`.

6.20.10 `function pjclJacobian2Affine(P,curve)`

The parameter `P` is expected to be a Jacobian representation of a finite point P over a prime field $\mathbb{F}_p$. The parameter `curve` is expected to be an object with properties `t`, `xc` and `p` that comprise a Pseudo-Mersenne representation of the prime number p, such as one of the curve objects `pjclCurve_P256` or `pjclCurve_P384`. The function returns the affine representation of P. (Recall that, in the library, an affine representation is a special case of a Jacobian representation, as explained in Section 6.20.4.)

6.20.11 function pjclPointAdd(P1,P2,curve)

The parameters `P1` and `P2` are expected to be Jacobian representations of two points P_1 and P_2 of a Weierstrass curve over a prime field $\mathbb{F}_p$, and the parameter `curve` is expected to be an object representing the curve. There are two objects representing curves in the current version of the library: `pjclCurve_P256` and `pjclCurve_P384`.

If one of the points P_1, P_2 is the point at infinity of the curve, the function represents the value of the parameter representing the other point. Otherwise, if $P_1 \neq P_2$, the function returns a Jacobian representation of the sum $P_1 + P_2$ and if $P_1 = P_2$ the function calls `pjclPointDouble(P1,curve)` and returns the result.

The function optimizes the case where P_2 is given by an affine representation. (Recall that, in the library, an affine representation is a special case of a Jacobian representation, as explained in Section 6.20.4.) This is useful for scalar multiplication, as explained below.

6.20.12 function pjclPointDouble(P,curve)

The parameter `P` is expected to be the Jacobian representation of a point P of a Weierstrass curve with coefficient $a = -3$, and the parameter `curve` is expected to be an object representing the curve. There are two objects representing curves in the current version of the library, `pjclCurve_P256` and `pjclCurve_P384`, both representing Weierstrass curves with coefficient $a = -3$. The function returns a Jacobian representation of the point $2P = P + P$.

6.20.13 function pjclScalarMult(P,k,curve)

The parameter `P` is expected to be a Jacobian representation of a point P of a Weierstrass curve with coefficient $a = -3$, the parameter `k` is expected to be a big integer whose mathematical value is a nonnegative integer k, and the parameter `curve` is expected to be an object representing the curve. There are two objects representing curves in the current version of the library, `pjclCurve_P256` and `pjclCurve_P384`, both representing Weierstrass curves with coefficient $a = -3$.

The function returns a Jacobian representation of the point $kP = \underbrace{P + \cdots + P}_{k}$, calculated using the sliding window algorithm implemented by `pjclExp`,[3] after calling `pjclPreExp` to perform the precomputation. The call to `pjclPreExp` is followed by a loop that calls `pjclJacobian2Affine` on all the precomputed values, so that `pjclPointAdd` can take advantage of the Jacobian-affine optimization mentioned above in Section 6.20.5 when used in `pjclExp`.

In a future version of the library we plan to use NAF to further optimize scalar multiplication. Different code will then be used for modular exponentiation and scalar multiplication.

6.20.14 function pjclScalarMult2(P1,P2,u1,u2,curve)

The function pjclScalarMult2(P1,P2,u1,u2,curve) produces the same result as

```
pjclPointAdd(pjclScalarMult(P1,u1,curve),pjclScalarMult(P2,u2,curve))
```

but substantially faster, by combining the point doublings of the two exponentiations. It calls `pjclPreExp2` and `pjclExp2`, and, like `pjclScalarMult`, calls `pjclJacobian2Affine` on the values precomputed by `pjclPreExp2` before using them in `pjclExp2`.

6.21 ECDSA

6.21.1 function pjclECDSA128GenKeyPair(rbgStateStorage,curve)

This function generates an ECDSA key pair with 128 bits of security strength. It uses the DRBG based on SHA-256 described at the beginning of Section 6.17, which provides that strength. The parameter `rbgStateStorage` is expected to be an object containing the state of the DRBG. The parameter `curve` is expected to be an object representing a Weierstrass curve with coefficient $a = 3$ providing that security strength, i.e., in this version of the library, `pjclCurve_P256`. The function returns an object containing two properties `d` and `Q` representing the private and public keys respectively. The value of the property `d` is a big integer whose mathematical value d is in the range $1 \leq d < n$, where n is the order of the curve, i.e. the mathematical

[3]Recall that "scalar multiplication" and "exponentiation" are alternative names given to the same external operation in a monoid, the term "scalar multiplication" being used when the operation is called "addition" while the term "exponentiation is used when the operation is called "multiplication".

value of `curve.n`. The value of the property `Q` is a Jacobian representation of the point $Q = dG$ where G is the base point of the curve, i.e. the value of `curve.G`.

6.21.2 function pjclECDSA192GenKeyPair(rbgStateStorage,curve)

This function generates an ECDSA key pair with 192 bits of security strength, using the DRBG based on SHA-384 described at the beginning of Section 6.17. It should be passed a curve that provides the same security strength, i.e., `pjclCurve_P384`. It is otherwise like `pjclECDSA128GenKeyPair`.

6.21.3 function pjclECDSAValidatePublicKey(Q,curve)

The function `pjclECDSAValidatePublicKey(Q,curve)` implements Algorithm 6 of [10] for ECDSA public key validation after verifying that `Q` is finite and converting it to its affine representation, except that it omits the last step of the algorithm, which is unnecessary if the cofactor is 1, since in that case (with the notations of Algorithm 6) $n = \#E(\mathbb{F}_p)$, and therefore $nQ = \mathcal{O}$. This hardcodes the fact that the cofactor is 1 in NIST curves, and will change in the future if the library supports curves with other cofactors.

6.21.4 function pjclECDSASign(curve,d,h,k)

THIS IS NOT AN API FUNCTION. DO NOT USE IT IN YOUR CODE.

This function computes an ECDSA signature on the hash `h` of a message, using a nonce `k` passed as an argument. It is used by pjclECDSA128Sign and pjclECDSA192Sign. Use one of those functions instead. Using this function with a value of `h` that is not a cryptographic hash of a message would not be secure, because an ECDSA signature is not secure against existential forgery if the message is not hashed. Using this function with a precomputed value of k might facilitate a timing attack: see Section **??**.

6.21.5 function pjclECDSA128Sign(rbgStateStorage,curve,d,msg)

This function computes an ECDSA signature with 128 bits of security strength, using a DRBG of that strength to generate the nonce. The parameter `rbgStateStorage` is expected to be an object containing the internal state of the DRBG. The parameter `curve` is expected to be an object representing a Weiesrstrass curve with coefficient $a = -3$ that provides 128 bits of security strength, i.e., in this version of the library, `pjclCurve_P256`. The parameter `d` is expected to be a nonnegative big integer whose value is the private key. The parameter `msg` is the message to be signed encoded as a bit array. The function returns the signature as an object with two properties `r` and `s` whose values are big integers.

6.21.6 `function pjclECDSA192Sign(rbgStateStorage,curve,d,msg)`

This function computes an ECDSA signature with 192 bits of security strength. It must be passed a curve of that strength. The other parameters and the result are as in `pjclECDSA128Sign`.

6.21.7 `function pjclECDSAVerify(curve,Q,msg,r,s)`

THIS IS NOT AN API FUNCTION. DO NOT USE IN YOUR OWN CODE. Use `pjclECDSA128Verify` or `pjclECDSA192Verify` instead.

6.21.8 `function pjclECDSA128Verify(curve,Q,msg,r,s)`

This function verifies an ECDSA signature with 128 bits of security strength computed using the curve referenced by the first parameter, returning `true` if successful or `false` otherwise. In the current version of the library there is one object representing a curve with 128 bits of security strength, `pjclCurve_P256`, which you may use as the first argument. The parameter `Q` is the public key. The parameter `msg` is the signed message, encoded as a bit array. The parameters `r` and `s` are the properties of the signature object, as returned by `pjclECDSA128Sign` or `pjclECDSA192Sign`.

6.21.9 `function pjclECDSA192Verify(curve,Q,msg,r,s)`

This function verifies an ECDSA signature with 192 bits of security strength computed using the curve referenced by the first parameter, returning `true` if successful or `false` otherwise. In the current version of the library there is one object representing a curve with 192 bits of security strength, `pjclCurve_P384`, which you may use as the first argument. The parameter `Q` is the public key. The parameter `msg` is the signed message, encoded as a bit array. The parameters `r` and `s` are the properties of the signature object, as returned by `pjclECDSA192Sign` or `pjclECDSA192Sign`.

7 Estimation of the Karatsuba thresholds

The directory `KaratsubaThresholds` contains a facility for estimating the optimal Karatsuba thresholds for multiplication and squaring on a target browser in a particular machine. JavaScript does not provide a means of measuring the number of clock cycles used in a computation, so the estimates are computed by measuring elapsed time, using the `performance.now()` method of the User Timing API. Results may be highly inaccurate if there is other activity on the machine where the browser is running.

To compute the optimal thresholds, simply visit the file `KaratsubaThresholds.html` found in the `KaratsubaThresholds` directory with the target browser. You may place the `KaratsubaThresholds` directory in a server and access the file using an `http` or `https` URL, or in the same machine where the browser is running and access the file using a `file` URL or open the file with the browser. However the facility cannot be used with Chrome if the file is local, and it cannot be used at all with Safari or Internet Explorer because those browsers do not support the User Timing API in web workers. There are no problems with Firefox or Edge. You must place the file `pjcl.js` containing the PJCL library in the parent directory of the `KaratsubaThresholds` directory.

The computation of the optimal thresholds is performed in the background by a web worker, which is launched automatically as soon as you visit the file, It takes a couple of minutes and may be monitored on the browser console. You may want to repeat the computation several times, discard outliers that might be caused by other activity on the machine, and average the retained results.

Once computed, the optimal thresholds should be assigned to the global variables `pjclKaratsubaThresholdMult` and `pjclKaratsubaThresholdSqr`, overriding the defaults. The default thresholds should be adequate for ordinary laptops. They may be too high for some smartphones, and too low for machines with very fast floating-point multiplication. Karatsuba is unlikely to be useful for elliptic curve computations.

8 Performance testing

8.1 Testing the performance of modular exponentiation

The directory `ModExpPerfTesting` contains files that allow you to test the performance of modular exponentiation on a browser using long multiplication or Karatsuba multiplication, and to compare it to the performance of SJCL.

For the performance test, simply visit the file `ModExpPerfTest.html` found in the `pjcl/ModExpPerfTesting` directory with a browser, and follow the instructions in the file. As when measuring the optimal Karatsuba thresholds, you may place the directory in a server and access the file using an `http` or `https` URL, or in the same machine where the browser is running and access the file using a `file` URL or open the file with the browser; but the facility cannot be used with Chrome if the file is local, and it cannot be used at all with Safari or Internet Explorer because those browsers do not support the User Timing API in web workers. There are no problems with Firefox or Edge. You must place the file `pjcl.js` containing the PJCL library in the parent directory of the `ModExpPerfTesting` directory.

For the performance comparison, place in the same directory the `pjcl` directory found in the PJCL zip archive downloaded from `https://pomcor.com/pjcl/pjcl-090.zip` and the `sjcl` directory found in the SJCL zip archive downloaded from `https://github.com/bitwiseshiftleft/sjcl`. (If you are using Windows, be sure to move the `pjcl` and `sjcl` directories out of the wrapping directories that Windows creates when unzipping the archives.) The SJCL code used in the comparison is in the files `sjcl/sjcl.js` and `sjcl/bn/core.js`.

Tables 1, 2 and 3 provide some measurements that we have made ourselves, using the default Karatsuba thresholds, 150 limbs = 3600 bits for multiplication and 175 limbs = 4200 for squaring. The SJCL performance figures have been obtained with a version of the SJCL library downloaded on November 15, 2017.

8.2 Testing the performance of DSA

The directory `DSAPerfTesting` contains files that allow you to test the performance of DSA key pair generation, signature and verification.

Machine: Surface with Intel Core i5-6300U CPU @ 2.40 GHz 2.50 GHz Browser: Firefox 56.0.2 (64-bit)				
Modulus and base	Exponent	Long multiplication and squaring	Karatsuba multiplication and squaring	SJCL
3072 bits	256 bits	22 ms	22 ms	186 ms
15360 bits	512 bits	819 ms	845 ms	7559 ms

Table 1: Modular exponentiation performance in Firefox

Machine: Surface with Intel Core i5-6300U CPU @ 2.40 GHz 2.50 GHz Browser: Chrome 62.0.3202.89 (64-bit)				
Modulus and base	Exponent	Long multiplication and squaring	Karatsuba multiplication and squaring	SJCL
3072 bits	256 bits	25 ms	27 ms	170 ms
15360 bits	512 bits	1098 ms	1005 ms	6756 ms

Table 2: Modular exponentiation performance in Chrome

Machine: Surface with Intel Core i5-6300U CPU @ 2.40 GHz 2.50 GHz Browser: Edge 41.16299.15.0, EdgeHTML 16.16299				
Modulus and base	Exponent	Long multiplication and squaring	Karatsuba multiplication and squaring	SJCL
3072 bits	256 bits	21 ms	22 ms	233 ms
15360 bits	512 bits	811 ms	763 ms	7009 ms

Table 3: Modular exponentiation performance in Edge

	Key pair generation	Signature	Verification
Firefox	21 ms	21 ms	26 ms
Chrome	22 ms	26 ms	38 ms

Table 4: DSA performance

To use this facility, visit the file `DSAPerfTest.html` found in the `DSAPerfTesting` directory with a browser, and follow the instructions in the file. As when measuring the optimal Karatsuba thresholds or the performance of modular exponentiation, you may place the directory in a server and access the file using an `http` or `https` URL, or in the same machine where the browser is running and access the file using a `file` URL or open the file with the browser. You must place the file `pjcl.js` containing the PJCL library in the parent directory of the `DSAPerfTesting` directory.

The facility can be used in Firefox, and in Chrome if the directory `DSAPerfTesting` is installed on a remote server. It cannot be used in Chrome if the directory is local because Chrome does not assign a common web origin to local files. It cannot be used in Edge because `crypto.getRandomValues()` is not available in worker context. It cannot be used in Safari or Internet Explorer because those browsers do not support the User Timing API in web workers.

Table 4 shows some measurements that we have made ourselves using Firefox and Chrome on a Windows machine with a processor identified as Intel Core i5-6300U CPU @ 2.40 GHz 2.50 GHz. The version of Firefox is 56.0.2 (64-bit) and the version of Chrome is 62.0.3202.89 (64-bit).

References

[1] Torbjörn Granlund and the GMP development Team. The GNU Multiple Precision Arithmetic Library. Edition 6.1.2. 16 December 2016. https://gmplib.org/gmp-man-6.1.2.pdf.

[2] Alfred J. Menezes and Paul C. Van Oorschot and Scott A. Vanstone and R. L. Rivest. Handbook of Applied Cryptography, 1997. Chapters available online at http://cacr.uwaterloo.ca/hac/.

[3] Emily Stark, Mike Hamburg, and Dan Boneh. Stanford JavaScript Crypto Library. https://github.com/bitwiseshiftleft/sjcl.

[4] NIST. Digital Signature Standard (DSS), July 2013. FIPS PUB 186-4, http://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.186-4.pdf.

[5] NIST. Secure Hash Standard (SHS), August 2015. FIPS PUB 180-4, http://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.180-4.pdf.

[6] H. Krawczyk, M. Bellare, and R. Canetti. HMAC: Keyed-Hashing for Message Authentication, February 1997. http://tools.ietf.org/html/rfc2104.

[7] NIST. The Keyed-Hash Message Authentication Code (HMAC), July 2008. FIPS PUB 198-1, http://csrc.nist.gov/publications/fips/fips198-1/FIPS-198-1_final.pdf.

[8] Elaine Barker and John Kelsey. Recommendation for Random Number Generation Using Deterministic Random Bit Generators, June 2015. NIST Special Publication 800-90A Revision 1. http://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-90Ar1.pdf.

[9] Elaine Barker. Recommendation for Key Management. NIST Special Publication 800-57 Part 1 Revision 4. http://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-57pt1r4.pdf.

[10] Don Johnson, Alfred Menezes, and Scott Vanstone. The elliptic curve digital signature algorithm (ecdsa). *Int. J. Inf. Secur.*, 1(1):36–63, August 2001.

[11] Lawrence C. Washington. *Elliptic Curves: Number Theory and Cryptography*. CRC Press, 2003.

[12] Facebook. React Native. https://facebook.github.io/react-native/.

[13] Node.js Foundation. node.js. https://node.js.

[14] Wikipedia. Double precision floating point format. https://en.wikipedia.org/wiki/Double-precision_floating-point_format.

[15] ECMA. ECMAScript Language Specification. ECMA-262 5.1 Edition, June 2011. http://www.ecma-international.org/ecma-262/5.1/Ecma-262.pdf.

[16] NIST. Secure Hash Standard (SHS), March 2012. FIPS PUB 180-4, http://csrc.nist.gov/publications/fips/fips180-4/fips-180-4.pdf.

[17] W3C. Web Storage (Second Edition)—W3C Recommendation 19 April 2016. https://www.w3.org/TR/webstorage/.

[18] NIST. Recommended Elliptic Curves for Federal Government Use. July 1999. http://csrc.nist.gov/groups/ST/toolkit/documents/NISTReCur.doc.

[19] Accredited Standards Committee X9. American National Standard X9.62-2005, Public Key Cryptography for the Financial Services Industry, The Elliptic Curve Digital Signature Algorithm (ECDSA), November 16, 2005.

[20] Mario Taschwer. Modular multiplication using special prime moduli. In Patrick Horster, editor, *Kommunikationssicherheit im Zeichen des Internet: Grundlagen, Strategien, Realisierungen, Anwendungen*, pages 346–371. Vieweg+Teubner Verlag, Wiesbaden, 2001.