Pomcor JavaScript Cryptographic Library (PJCL)

https://pomcor.com/pjcl/

Version 1.0.1

Contents

List of Tables

1 Preliminaries

1.1 License

The PJCL library is available under the terms of the MIT open source license, which can be found at `https://opensource.org/licenses/MIT`.

1.2 Entropy requirement

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 [1, § 8.8]. You are responsible for providing the random bits used for seeding or reseeding. Methods for obtaining entropy are discussed below in Section 21.2. `Math.random` does not provide entropy.

2 Data encodings

2.1 Small integers

JavaScript numbers are represented in IEEE 754 double-precision (64-bit) floating point format [2], which allows every nonnegative integer n in the range $0 \leq n < 2^{53}$ to be represented exactly. Floating point numbers are silently converted to 32-bit integers before applying bitwise boolean and shift operators, but there are no integer arithmetic operators in JavaScript.

This library uses JavaScript floating point numbers to encode bits, bytes, unsigned 32-bit integers and, as discussed below, 24-bit limbs of big integers. Hex digits, on the other hand, are always encoded as characters in JavaScript strings. Whenever a sequence of bits is the binary representation of a byte, 32-bit integer, or hex digit, the most significant bit goes first in the sequence; we refer to this a *big-endian bit ordering*.

We represent a sequence of bits as a *bit array*, i.e. an array of numbers where each element is 0 or 1, and a sequence of bytes (sometimes called an octet string) as a *byte array*, i.e. an array of numbers where each element is an 8-bit integer. Cryptographic hash functions and HMAC take bit arrays as inputs, while HKDF and PBKDF2 take byte arrays as inputs. Representing a bit sequence as a JavaScript array of 64-bit numbers is not space-efficient, but it is computationally efficient, and the space inefficiency does not matter for purposes such as authentication or key derivation. It would matter for hashing the contents of a very large file, since it might be difficult or impossible to represent the entire contents of the file as a bit array; but a future version of the library will provide incremental hashing for that purpose.

In function names "`UI32`" refers to an unsigned 32-bit integer represented as a JavaScript number, and "`UI32Array`" to an array of unsigned 32-bit integers. Notice that "`UI32Array`" refers to an ordinary JavaScript array, not to a *typed array*.

2.1.1 Typed arrays and Node.js buffers

The library does not construct any *typed arrays*, nor any *Node.js buffers*, and library functions do not return such constructs. However, when a function parameter is expected to be an array of bytes, or an array of unsigned 32-bit integers, a typed array or a Node.js buffer can be passed instead as an argument.

2.2 Big integers

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 constants of Section 4.1 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.

2.3 Unicode text

JavaScript uses the type *String* to encode Unicode text in UTF-16, a variable-length encoding where each *code point* is encoded by a 16-bit *code unit* or a so-called *surrogate pair* of code units, referred to as the *high-surrogate code unit* and the *low-surrogate code unit*. A JavaScript string is a sequence of code units. If `s` is a string and `n` a non-negative number less than `s.length`, then `s.charCodeAt(n)` is the code unit at position `n` in `s`. If a code point is encoded by a surrogate pair, the high code unit goes before the low code unit in the sequence.

The Unicode standard defines two *byte serializations*, which specify how a sequence of code units is mapped to a sequence of bytes: big endian (UTF-16BE), where the most significant byte of each code unit goes before the least significant byte, and little-endian (UTF-16LE), where the least significant byte goes first. In both serializations, the high-surrogate code unit, a.k.a. the *leading surrogate*, goes before the low-surrogate code unit, a.k.a. the *trailing surrogate*. The standard provides an optional *byte order mark*, `0xFEFF`, that can be prefixed to a byte serialization to indicate its byte order. This version of the library treats `0xFEFF` (which is the code point of an invisible character) and `0xFFFE` (which is an unassigned code unit) as ordinary UTF-16 code units.

The JavaScript language does not provide individual access to each byte of a code unit, and therefore does not specify a particular byte serialization. Unfortunately, this means that the concept of a *cryptographic hash* is ambiguous for a JavaScript string. The string must be converted to a bit sequence before it can be hashed, and there are different ways of doing that, resulting in different bit sequences. It also means that password-based key derivation, e.g. as specified by PBKDF2, is undefined if the password is encoded as a JavaScript string. The password must first be converted to a byte sequence before it can be passed as an argument to PBKDF2, and again there are different ways of doing that, which result in different byte sequences.

This version of the library provides four functions that convert strings to byte arrays that can be used as inputs to hashing (after further conversion to bit arrays) or key derivation:

- `pjclString2ByteArray_UTF16BE` implements big-endian serialization.
- `pjclString2ByteArray_UTF16LE` implements little-endian serialization.
- `pjclString2ByteArray_UTF8` converts the string to a byte array by converting each UTF-16 character in the string (represented by one code unit or a surrogate pair) to the one-to-four byte sequence of its UTF-8 encoding.
- `pjclString2ByteArray_ASCII` assumes that the string contains only ASCII characters and converts the string to a byte array whose elements are the ASCII code points of the characters.

The library also provides a function `pjclByteArray2BitArray` and, for convenience and performance, four functions that convert directly from strings to bit arrays:

`pjclString2BitArray_UTF16BE`,
`pjclString2BitArray_UTF16LE`,
`pjclString2BitArray_UTF8`, and
`pjclString2BitArray_ASCII`.

Version 0.9.0 had functions `pjclUTF16toBitArray` and `pjclASCII2BitArray`. These are now global variables that can be used as synonyms for the functions `pjclString2BitArray_UTF16BE` and `pjclString2BitArray_ASCII` respectively.

2.4 Hex strings

We use the term *hex string* to refer to 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 case hexadecimal digits depending on the JavaScript engine that interprets the function.

3 API generalities

Sections 4 through 29 describe the global variables and functions that comprise the API in the order in which they are declared in the code.

3.1 Argument checking

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.

3.2 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`.

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

4.1 `export const pjclBaseBitLength` `export const pjclBase` `export const pjclBaseMask` `export const pjclBaseMaskMinusOne` `export const pjclBaseInv` `export const pjclBaseAsBigInt` `export const 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` is $B-1$, 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.

4.2 `function pjclWellFormed(x)`

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

5 Conversion functions

5.1 `export function pjclBigInt_from_ES11BigInt(i)`

The parameter `x` is expected to be an ECMAScript 2020 (ES11) BigInt with mathematical value i. The function returns a returns a big integer with mathematical value i.

5.2 `export function pjclBigInt_to_ES11BigInt(x)`

The parameter `x` is expected to be a big integer with mathematical value x. The function returns an ECMAScript 2020 (ES11) BigInt with mathematical value x.

5.3 `export 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.

5.4 `export function pjclByteArray2BitArray(byteArray)`

Converts a byte array to a bit array, using big-endian bit ordering.

5.5 `export function pjclBitArray2ByteArray(bitArray)`

The parameter `bitArray` is expected to be a bit array whose length is a multiple of 8. The function returns the result of converting the bit array to a byte array, using big-endian bit ordering.

5.6 `export function pjclString2ByteArray_ASCII(s)`

The parameter `s` is expected to be an ASCII string, which the function converts to a byte array where the value of each byte is the code point of the corresponding ASCII character. Note that although each character is encoded as a 16-bit UTF-16 code unit in the JavaScript string `s`, it is mapped to a single byte in the resulting bit array.

When `s` is an ASCII string, `pjclString2ByteArray_ASCII(s)` produces the same result as `pjclString2ByteArray_UTF8(s)` but more efficiently. (Another reason to use `pjclString2ByteArray_ASCII`(s) instead of `pjclString2ByteArray_UTF8(s)` is that, with argument checking, it checks that `s` is an ASCII string.)

5.7 `export function pjclString2BitArray_ASCII(s)` `export const pjclASCII2BitArray = pjclString2BitArray_ASCII`

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-bit binary representation of the code point of the character in big-endian bit ordering. Since an ASCII code point is an integer in the range 0. . . 127, the first of the eight 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 eight bits in the resulting bit array.

As other string-to-bit-array conversion functions, `pjclString2BitArray_ASCII` could be implemented by a call to `pjclString2ByteArray_ASCII` followed by a call to `pjclByteArray2BitArray`, but this would reduce performance.

5.8 `export function pjclString2ByteArray_UTF8(s)`

The parameter `s` is expected to be a JavaScript string, which the function converts to a byte array by converting each UTF-16 character in the string (represented by one code unit or a surrogate pair) to the one-to-four byte sequence of its UTF-8 encoding. With argument checking, the function throws an exception if `s` ends at a high surrogate, or a high surrogate in `s` is not followed by a low surrogate.

5.9 `export function pjclString2BitArray_UTF8(s)`

This is a convenience function that applies `pjclString2ByteArray_UTF8` to `s`, then applies `pjclByteArray2BitArray` to the resulting byte array.

5.10 `export function pjclString2ByteArray_UTF16BE(s)`

The parameter `s` is expected to be a JavaScript string. The function returns the big-endian byte serialization of the sequence of UTF-16 code units of `s`. With argument checking, the function checks that `s` is a JavaScript string, but, contrary to `pjclString2ByteArray_UTF8` it does not check whether `s` ends in a high surrogate or contains a high surrogate not followed by a low surrogate.

5.11 `export function pjclString2BitArray_UTF16BE(s)` `export const pjclUTF16toBitArray = pjclString2BitArray_UTF16BE`

The parameter `s` is expected to be a string, which the function converts to a bit array by mapping each UTF-16 code unit in `s` to a sequence of 16 bits in big-endian bit order. Like `pjclString2ByteArray_UTF16BE`, with argument checking `pjclString2BitArray_UTF16BE` checks that `s` is a JavaScript string, but does not check whether `s` ends in a high surrogate or contains a high surrogate not followed by a low surrogate.

As other string-to-bit-array conversion functions, `pjclString2BitArray_UTF16BE` could be implemented by a call to `pjclString2ByteArray_UTF16BE` followed by a call to `pjclByteArray2BitArray`, but this would reduce performance.

5.12 `export function pjclString2ByteArray_UTF16LE(s)`

The parameter `s` is expected to be a JavaScript string. The function returns the little-endian serialization of the sequence of UTF-16 code units of `s`. With argument checking, the function checks that `s` is a JavaScript string, but does not check whether `s` ends in a high surrogate or contains a high surrogate not followed by a low surrogate.

5.13 `export function pjclString2BitArray_UTF16LE(s)`

This is a convenience function that applies `pjclString2ByteArray_UTF16LE` to `s`, then applies `pjclByteArray2BitArray` to the resulting byte array. Since the mapping from UTF-16 code units to bytes uses little-endian byte order, but the subsequent mapping from bytes to bits uses big-endian bit order, the order of the bits in the resulting bit array is peculiar: for example, the least significant bit of the first 16-bit code unit is at position 7 in the bit array, and is followed by the most significant bit at position 8.

5.14 `export function pjclUI32toBitArray(ui32)`

The parameter `ui32` is expected to be a JavaScript number whose value is 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.

5.15 `export 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 2, PJCL does not construct typed arrays, but an application may pass a `Uint32Array` as an argument to the function instead of an ordinary JavaScript array.

5.16 `export function pjclUI32Array2ByteArray(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 returns a byte array obtained by mapping each n to four bytes and pushing the bytes to the array, most significant byte frst.

5.17 `export function pjclBigInt2ByteArray(x)` `export function pjclBigInt2ByteArray(x,minByteLength)`

The parameter `x` is expected to be a big integer with mathematical value x, and the parameter `minByteLength`, if the function is called with two arguments, a JavaScript number whose value is a nonnegative integer n. The function returns a byte array whose elements comprise the big-endian representation of x in base 256. If the second argument is not omitted, sufficient leading zero bytes are added to the byte array to bring its length to n.

5.18 `export 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.

5.19 `export 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 $|x| \bmod 2^n$.

5.20 `export 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)`.

5.21 `export 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.

5.22 `export 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.

5.23 `export function pjclBitArray2Hex(bitArray)` `export function pjclBitArray2Hex(bitArray,minHexLength)`

The parameter `bitArray` is expected to be a bit array of any length. The optional parameter `minHexLength`, if present, is expected to be a JavaScript number whose value l is a nonnegative integer. The function returns the hex string that would be obtained by: (i) prepending leading zero bits to the bit array as needed to make the length of the array a multiple of four; (ii) mapping each group of four bits in the array to a hex digit, with big-endian bit ordering; and (iii) if the function is called with two arguments, prepending zero hex digits as needed to bring the value of the hex string up to l. If the bit array is empty and `minHexLength` is not supplied or has value 0, the function returns an empty string.

5.24 `export 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.

5.25 `export function pjclHex2ByteArray(s)`

The parameter `s` is expected to be a hex string of even length, which the function converts to a byte array by mapping each consecutive pair of hex digits in `s` to a byte.

5.26 `export function pjclByteArray2Hex(byteArray)`

The function expects its argument to be a byte array, which it converts to a hex string by concatenating the hexadecimal representations of the bytes.

5.27 `export 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.

5.28 `export function pjclBigInt2Hex(x)` `export function pjclBigInt2Hex(x,minHexLength)`

The parameter `x` is expected to be a big integer with mathematical value x, and the parameter `minHexLength`, if the function is called with two arguments, a JavaScript number whose value l is a nonnegative integer. The `negative` property of `x`, if present, is ignored. The function returns the hexadecimal representation of x as a hex string, with leading zero hex digits as needed to bring its length up to l if the function is called with two arguments. If $x = 0$ and the second argument is omitted or has value 0, the resulting hex string is empty.

5.29 `export 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 of length four.

5.30 `export function pjclUI32Array2Hex(x)`

The parameter `x` is expected to be an array of n unsigned 32-bit integers. The function converts `x` to a hex string by mapping each integer to its hexadecimal representation, of length $4n$.

6 Basic arithmetic functions

6.1 `export 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.

6.2 `export 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.

6.3 `export 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.

6.4 `export 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 `export 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.

6.6 `export 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.

6.7 `export 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 $x + y$.

6.8 `export 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.

6.9 `export 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 $|x| - |y|$.

6.10 `export 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.

6.11 `export let pjclMult` `export function pjclMult_Long(x,y)` `export 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 [3, § 15.1.2].

6.12 `export 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 mathematical value is the product xy.

6.13 `export 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 $|x| \cdot y$.

6.14 `export let pjclSqr` `export function pjclSqr_Long(x,y)` `export 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.

6.15 `export function pjclShortShiftLeft(x,k)`

As discussed above in Section 2.2, "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.

6.16 `export function pjclShiftLeft(x,k)` `export function pjclMultByPowerOf2(x,k)`

As discussed above in Section 2.2, "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.

6.17 `export 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.

6.18 `export function pjclShiftRight(x,k)` `export 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 $\lfloor x/2^k \rfloor$.

6.19 `export 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 [4] and returns an object with properties `quotient` and `remainder` whose values are big integer representations of the quotient and the remainder.

6.20 `export 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 $r = x - qy$.

6.21 `export 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 [5, §11.5.3].

6.22 `export 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 $x \bmod m$.

6.23 `export function pjclTruncate(x,t)` `export 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.

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.

6.24 `export 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 $x \bmod m$.

6.25 `export function pjclEGCD(a,b)` `export 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.

6.26 `export 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.

7 Montgomery reduction

Our implementation of Montgomery reduction is based on Section 14.3.2 of the Menezes et al. Handbook of Applied Cryptography [4]. More specifically, it is based on the optimized Algorithm 14.32, further optimized and adapted for use with our big integer representation.

In this Section 7 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 2.2.

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 $(m_{n-1} \dots m_1, m_0)_B$ 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 algorithm 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'.

7.1 `export 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 $m' = -m^{-1} \bmod B$.

7.2 `export 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 $TR^{-1} \bmod m$.

8 Generic sliding window exponentiation in a monoid

8.1 `export function pjclOptimalWindowSize(l)` `export function pjclPreExp(slidingWindowSize,context)` `export 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 [4]. 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 described below in Section 26.13, 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 advantage 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 $e = 0$, where the function would return

[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".

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 `pjclModExp` 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 bit length of the exponent. The function returns a JavaScript number whose mathematical value is the optimal window size.

9 Exponentiation in $\mathbb{N}$

9.1 `export 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.

10 Modular exponentiation with ordinary reduction

10.1 `export 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 `pjclPlainExp`, it is too slow to be used in most cryptographic applications.

11 Modular exponentiation with Montgomery reduction

11.1 `export 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 7 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.

12 Generic double exponentiation in a commutative monoid

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

These functions are like those of Section 8.1, 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.

13 Double exponentiation with Montgomery reduction

13.1 `export 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`.

14 Hash functions (SHA-2 family)

This version of the library provides two members of the SHA-2 family of hash functions: SHA-256 and SHA-384.

14.1 `export function pjclSHA256(bitArray)` `export function pjclSHA384(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 [6] to the input.

The functions `pjclSHA384` similarly implements SHA-354.

15 Message authentication codes (HMAC)

15.1 `export function pjclHMAC_SHA256(key,text)`

The function `pjclHMAC_SHA256` implements the HMAC algorithm of [7] instantiated with the hash function SHA-256 of [6]. The parameters `key` and `text` are expected to be bit arrays, and the result is a bit array.

15.2 `export function pjclHMAC_SHA384(key,text)`

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

15.3 `export function pjclHMAC_SHA256PreComputeKeyHashes(key)` `export 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`.

15.4 `export function pjclHMAC_SHA384PreComputeKeyHashes(key)` `export 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.

16 Extract-and-expand key derivation (HKDF)

The HMAC-based Extract-and-Expand Key Derivation Function (HKDF), specified in RFC 5869 [8], is used to derive an unlimited amount of pseudo-random output keying material from a limited amount of input keying material that contains entropy but may or may not be uniformly distributed. The input keying material may be, e.g., a shared secret established using a key establishment primitive such as Diffie-Hellman, and the output keying material may be used, e.g., to construct a symmetric encryption key plus a symmetric message authentication key that may be used to provide traffic confidentiality and integrity protection in a secure channel.

HKDF uses HMAC instantiated with a cryptographic hash function. This version of the library provides HKDF using HMAC instantiated with SHA-256, which is suitable for use in conjunction with all key establishment primitives contemplated by NIST, as seen in Tables 1-3 of SP 800-56C [9].[2]

As described in [8], HKDF has two steps. Step 1, the *Extract* step, takes as input an optional salt and the input keying material `IKM`, and produces a uniformly distributed pseudo-random key `PRK`. Step 2, the *Expand* step, produces the output keying material `OKM`, taking as input `PRK`, optional context-specific information `info`, and the desired length in bytes `L` of `OKM`. Step 1 is optional, because `IKM` itself can be used as the `PRK` input to Step 2 if it is a uniformly distributed pseudo-random key. So the library provides two functions, `pjclHKDF_SHA256_Expand`, which implements Step 2 by itself, and `pjclHKDF_SHA256`, which implements both steps.

16.1 `export function pjclHKDF_SHA256_Expand(PRK,info,L)`

The function `pjclHKDF_SHA256_Expand` implements Step 2 of HKDF as described in [8], using HMAC instantiated with SHA-256. The pseudo-random key parameter `PRK` and the optional context-specific information parameter `info` are expected to be byte arrays. To omit the context-specific information, pass an empty array `[]` as the second argument. The parameter `L` is expected to be a positive JavaScript number, specifying the length L in bytes of the output keying material to be derived. The function returns the output keying material as a byte array of length L.

16.2 `export function pjclHKDF_SHA256(IKM,L,salt,info)`

The function `pjclHKDF_SHA256` implements both steps of HKDF as described in [8], using HMAC instantiated with SHA-256. The input keying material parameter `IKM` is expected to be a byte array. The parameter `L` is expected to be a positive JavaScript number, specifying the length L in bytes of the output keying material to be derived. The parameters `salt` and `info` are optional, and are expected to be byte arrays if supplied. If `info` is omitted, the empty array `[]` is used as its default value. If `salt` is also omitted, the function behaves as if it was an array of 32 bytes. The function returns the output keying material as a byte array of length L.

[2]When used for purposes that require collision resistance, SHA-256 provides a security strength equal to only half the bit length of its output, i.e. 128 bits; but when used for other purposes it provides a security strength equal to the full bit length of its output, i.e. 256 bits, as seen in Table 3 of SP 800-57 [10]. HKDF does not require collision resistance.

17 Password-based key derivation (PBKDF2)

Password-Based Key Derivation Function 2 (PBKDF2) derives a key from a password and a salt using a method designed to be slow for the purpose of mitigating dictionary attacks against the password. Computing the derived key requires calling an underlying hash function c times, where c is an *iteration count* passed to the function as an argument.

PBKDF2 is specified in RFC 8018 [11], which is a republication of PKCS #5 and obsoletes RFC 2898 [12].

17.1 `export function pjclPBKDF2_SHA256(P,salt,count,dkLen)`

The function `pjclPBKDF2_SHA256` computes PBKDF2 using SHA-256 as the underlying hash function.

The parameter `P` is expected to be an encoding of the password as a byte array. A string encoding cannot be used because, as explained in Section 2.3, a JavaScript string cannot be unambiguously hashed. If the password is provided as a string, it must be converted to a byte array using one of the functions `pjclString2ByteArray_UTF16BE`, `pjclString2ByteArray_UTF16LE`, `pjclString2ByteArray_UTF8`, or `pjclString2ByteArray_ASCII`.

The parameter `salt` is expected to be a byte array. The parameter `count` is expected to be a JavaScript number whose value c is a positive integer, used as the iteration count. The parameter `dkLen` is expected to be a JavaScript number whose value n is a positive number specifying the desired length in bytes of the derived key. For the sake of strict adherence to the standard, with argument checking the function throws an exception if n is greater than $(2^{32} - 1)$ times the length in bytes of SHA-256, i.e. if $n > (2^{32} - 1) \times 32 =$ `0x1FFFFFFFE0`. Otherwise it returns the derived key as an array of n bytes.

18 Statistically random data vs. cryptographically random data

We make a distinction between statistically random data 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 [1, § 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

[5, 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.

19 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.

20 Generation of statistically random data

20.1 `export 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 $0 \leq n < B = 2^\beta = 2^{24}$.

20.2 `export 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 $0 \leq x < B^n$.

20.3 `export function pjclStRndHex(n)`

The parameter `n` is expected to be a JavaScript number whose mathematical value is a nonnegative 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.

20.4 `export 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 $a \leq x < b$.

21 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 [1, § 10.1.1], instantiated with the hash function SHA-256 for 128 bits of security strength or SHA-384 for 192 bits of security strength.

21.1 Storage of the internal state of a DRBG

In functions that use a DRBG, the parameter called `rbgStateStorage` is expected to be an object used to store the internal state of the DRBG. That object may be an ordinary JavaScript object or, in a JavaScript runtime environment that implements the *W3C Web Storage* specification [13], a *storage object*, either `localStorage` or `sessionStorage`.

The `localStorage` object persists across browser sessions but cannot be accessed by web workers. However a DRBG that uses `localStorage` can provide random bits that can be passed to a web worker and used by the web worker to initialize its own DRBG. Examples of how to do this can be found in `DSAPerfTesting` and `DHPerfTesting`.

If an ordinary JavaScript object is used in a browser environment, it can be persisted across browser sessions by saving it to a browser database using the IndexedDB API [14]. If an ordinary JavaScript object is used in Node.js running on a server, the DRBG may be initialized (*instantiated* in NIST terminology) each time Node.js is started or, if desired, the object may be persisted by saving it to a server-side database such as MongoDB. The parameter `rbgStateStorage` may be viewed as an implementation of the *state_handle* of [1].

A DRBG has a nominal security strength and can be used for purposes that require up to that strength. When random bits are needed for different purposes that require different security strengths, a DRBG supporting the highest strength can be used for all those purposes. However, it may be desirable to use DRBGs with different strengths for different purposes to take advantage of the higher performance provided by DRBGs with lower strength. Only one DRBG state can be stored in a given object. Multiple DRBGs can be implemented by storing their states in different objects. However only one DRBG can store its state in `localStorage`.

21.2 `export function pjclRBG128Instantiate(rbgStateStorage,entropy)` `export function pjclRBG128Instantiate(rbgStateStorage,entropy,nonce`

This function instantiates a DRBG with 128 bits of security strength as specified in Section 10.1.1.2 of [1]. No *personalization_string* is used. As discussed in Section 21.1, the parameter `rbgStateStorage` is expected to be a storage object or an ordinary JavaScript object where the function will create the internal state of the DRBG. To use `localStorage`, call the function as follows:

```
const myEntropy = ...;
const myNonce = ...; //optional
```

```
pjclRBG128Instantiate(localStorage,myEntropy,myNonce);
```

To use an ordinary object, call the function as follows:

```
const myEntropy = ...;
const myNonce = ...; //optional
const myRBGState = new Object();
pjclRBG128Instantiate(myRBGState,myEntropy,myNonce);
```

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 browser 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 use browser entropy are provided by two functions `pjclBrowserEntropy128Bits` and `pjclBrowserEntropy192Bits`, which can be found in the file `browser-entropy.js`. In a JavaScript runtime environment such as Node.js that provides access to an underlying Unix-like OS you may use `/dev/random`, which provides 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 [1].) 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`: `pjclRBG128_v`, `pjclRBG128_c` and `pjclRBG128_reseed_counter`. If these properties exist, they are overwritten. If corresponding properties for the 192 security strength exist (`pjclRBG192_v`, `pjclRBG192_c` and `pjclRBG192_reseed_counter`) the function throws an exception. To avoid the exception you may use a fresh ordinary object, or remove the offending properties from a storage object using its `removeItem` method. (Actually, strictly speaking, only the existence of `pjclRBG192_v` is checked and needs to be removed, but it is a best practice to remove them all.)

21.3 `export function pjclRBG128Reseed(rbgStateStorage,entropy)`

This function reseeds a DRBG based on the *Hash_DRBG* mechanism instantiated with SHA-256 as specified in Section 10.1.1.3 of [1]. No *additional_input* is used. The parameter `rbgStateStorage` is expected to be an ordinary object or a storage object containing the internal state of a DRBG with 128 bits of security strength, and an exception is thrown otherwise by the argument checking version of the library. 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.

21.4 `export function pjclRBG128InstantiateOrReseed( rbgStateStorage,entropy,nonce)`

The parameters `rbgStateStorage`, `entropy` and `nonce` are expected to be as in `pjclRBG128Instantiate`. The function `pjclRBG128InstantiateOrReseed` is a convenience function that calls `pjclRBG128Instantiate(rbgStateStorage,entropy,nonce)` to initialize a DRBG at `rbgStateStorage` unless one already exists there, in which case it calls `pjclRBG192Reseed` to reseed the existing DRBG using the concatenation of the entropy and the nonce as the entropy argument.

21.5 `export function pjclRBG192Instantiate( rbgStateStorage,entropy,nonce)` `export function pjclRBG192Reseed( rbgStateStorage,entropy)` `export function pjclRBG192InstantiateOrReseed( rbgStateStorage,entropy,nonce)`

The functions `pjclRBG192*` are like the corresponding functions `pjclRBG128*` except that they use SHA-384 as the hash function and 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.

21.6 `export function pjclRBGSecStrength(rbgStateStorage)`

The parameter `rbgStateStorage` is expected to be an ordinary object or a storage object. The function returns a JavaScript number whose value is the security strength of a DRBG whose internal state is stored in `rbgStateStorage`, or zero if no well-formed DRBG state can be found in `rbgStateStorage`.

21.7 `export function pjclRBGGen( rbgStateStorage,requestedSecStrength,bitLength)`

This function generates random bits from a DRBG as specified in Section 10.1.1.4 of [1]. The parameter `rbgStateStorage` is expected to be an ordinary object or a storage object containing the internal state of the DRBG as discussed in Section 21.1. The parameter `requestedSecStrength` is expected to be a JavaScript number specifying the security strength requested for the random bits. An exception is thrown if this is greater than the security strength of the DRBG whose state is found in `rbgStateStorage`. 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 [1]. The function throws an exception otherwise. The function returns a bit array with the specified number of bits.

21.8 `export function pjclCryptoRNG( rbgStateStorage,requestedSecStrength,a,b)`

This function generates a cryptographically random big integer in a specified range. The parameter `rbgStateStorage` is expected to be an ordinary object or a storage object containing the internal state of the DRBG as discussed in Section 21.1. The parameter `requestedSecStrength` is expected to be a JavaScript number specifying the security strength requested for the random number generation. An exception is thrown if this is greater than the security strength of the DRBG whose state is found in `rbgStateStorage`. 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 quasi-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 [15] for key pair generation and in Section B.2.1 for per-message secret number generation.

22 Primality testing

22.1 `export function pjclIsPrime(n,t)` `export 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 [4] 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 compositeness 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.

23 Finite Field Cryptography (FFC) for DSA and DH

NIST uses the term Finite Field Cryptography (FFC) to refer to public-key cryptographic primitives, including DSA and Diffie-Hellman (DH), that rely on the difficulty of computing discrete logarithms in the multiplicative group $(\mathbb{Z}_p^*, \times)$ of the field $\mathbb{Z}_p$ of integers modulo p, where p is a prime number such that $p - 1$ is divisible by a large prime q.

Such primitives use domain parameters (p, q, g), where g is a generator of the unique cyclic subgroup of order q of the group $(\mathbb{Z}_p^*, \times)$, and key pairs (x, y) where the public key y is such that $y = g^x \bmod p$. Section B.1.1 of [15] specifies that the private key x must be in the range $1 \leq x < q$, but $x = 1$ could be trivially detected from the public quantities y and g. While there is a negligible probability that a secure DRBG will generate $x = 1$ within the range $1 \leq x < q$, as a matter of defense in depth it is preferable to restrict x to be in the range $2 \leq x < q$. In this documentation we say that (x, y) is a *well-formed key pair* relative to the domain parameters (p, q, g) if $2 \leq x < q$ and $y = g^x \bmod p$. We also say that x is a *well-formed private key* and y a *well-formed public key* relative to (p, q, g) if (x, y) is a well-formed key pair.

FFC primitives have *nominal security strengths* that depend on the bit lengths L and N of p and q. The nominal security strength of a primitive, however, is only an upper limit on its *actual security strength*, which may also be limited by other factors, such as the security strength of the DRBG used to generate a key pair or the per-message secret used to compute a signature, or the security strength of the hash function used to hash a message to be signed. With argument checking, functions implementing FFC primitives verify that these other factors do not reduce the actual security strength of a primitive below its nominal strength.

This version of the library can generate domain parameters with lengths $(L, N) = (3072, 256)$ and $(L, N) = (2048, 256)$, which provide nominal security strengths of 128 and 112 bits respectively according to [10, Table 2], and can validate domain parameters of those lengths provided by untrusted parties. It can also make use of domain parameters of other lengths provided by trusted sources.

23.1 `export function pjclFFCSecStrength(p,q)`

The parameters `p` and `q` are expected to be non-negative big integers. The function observes the bit lengths L and N of `p` and `q` and returns the security strength assigned by [10, Table 2] to domain parameters with those bit lengths. The function does not otherwise validate `p` and `q`; domain parameter validation is performed by `pjclFFCValidatePQ`.

23.2 `export function pjclFFCGenPQ_3072_256()` `export function pjclFFCGenPQ_3072_256(domainParameterSeed)`

The function `pjclFFCGenPQ_3072_256` generates probable primes p and q of bit lengths $L = 3072$ and $N = 256$ respectively, with q dividing $p - 1$, to be used as FFC domain parameters. It is implemented as specified in Section A.1.1.2 of [15] 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 to validate prime numbers p and q if generated by a non-trusted party, as described below.

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` and `q`, whose values are big integers representing the domain parameters p and q, as well as properties `domainParameterSeed` and `counter` whose values are the domain parameter seed and counter of Algorithm A.1.1.2, encoded as a bit array and a JavaScript number respectively. The domain parameter seed and counter can be provided to a third party who wishes to validate the generation of p and q as specified in Algorithm A.1.1.3 of [15]. The domain parameters p and q produced by the function provide a nominal security strength of 128 bits.

23.3 `export function pjclFFCValidatePQ_3072_256(p,q,domainParameterSeed,counter)`

This function can be used to validate domain parameters p and q of bit lengths $L = 3072$ and $N = 256$ when they are provided by an untrusted third party, using a domain parameter seed and a counter provided by the third party, as specified by Algorithm A.1.1.3 of [15]. The parameters `p` and `q` are expected to be big integers whose values are p and q, while the last two parameters are expected to encode the domain parameter seed and the counter as a bit array and a JavaScript number respectively. The function returns `true` if validation succeeds, `false` otherwise.

23.4 `export function pjclFFCGenPQ_2048_256()` `export function pjclFFCGenPQ_2048_256(domainParameterSeed)`

This function and its optional parameter `domainParameterSeed` are like `pjclFFCGenPQ_3072_256`, except that the bit length of the generated prime p is $L = 2048$ instead of $L = 3072$. The domain parameters p and q generated by the function provide a nominal security strength of 112 bits.

23.5 `export function pjclFFCValidatePQ_2048_256(p,q,domainParameterSeed,counter)`

This function can be used to validate domain parameters p and q like `pjclFFCValidatePQ_3072_256`, when their bit lengths are $L = 2048$ and $N = 256$. It returns `true` if validation succeeds, `false` otherwise.

23.6 `export function pjclFFCGenG_256(p,q)` `export function pjclFFCGenG_256(p,q,domainParameterSeed,index)`

This function can be used to generate the component g of the FFC domain parameters (p, q, g) given p and q, i.e. to produce a generator g of the subgroup of order q of the multiplicative group of the field $\mathbb{Z}_p$. The parameters `p` and `q` are expected to be big integers representing p and q. If four arguments are supplied, the function performs verifiable generation of g as specified by Algorithm A.2.3 of [15], with SHA-256 as the hash function used by the algorithm. The parameter `domainParameterSeed` is then expected to be a bit array of length 256, encoding the domain parameter seed produced by Algorithm A.1.1.3 and used by Algorithm 1.1.4 for validation of p and q, and the parameter `index` is expected to be a bit array of length 8. If only two arguments are supplied, the function performs unverifiable generation of g as specified by Algorithm A.2.1. The function returns a big integer representing g.

23.7 `export function pjclFFCValidateG_256(g,p,q)` `export function pjclFFCValidateG_256(g,p,q,domainParameterSeed,index`

This function can be used to validate the component g of the FFC domain parameters (p, q, g) when it is provided by an untrusted third party. The parameters `p` and `q` are expected to be big integers whose values are p and q.

If five arguments are supplied, the function performs full validation as specified by Algorithm A.2.4 of [15], assuming that g was generated using Algorithm A.2.3, with SHA-256 as the hash function used by the algorithm. The parameter `domainParameterSeed` is then expected to be a bit array of length 256, encoding the domain parameter seed produced by Algorithm A.1.1.3 and used by Algorithm 1.1.4 for validation of p and q, while the parameter `index` is expected to be a bit array of length 8, encoding the index that Algorithm A.2.4 takes as input. The function returns `false` if the full validation fails, or the truthy value "`Valid`" if it succeeds.

If only three arguments are supplied, the function performs partial validation of g as specified by Algorithm A.2.2. The function returns `false` if partial validation fails, or the truthy value "`Partially valid`" if it succeeds.

Notice that the function will never return "`Partially valid`" if five arguments are supplied. It will only return `false` or "`Valid`" in that case.

23.8 `export function pjclFFCGenPQG_3072_256()` `export function pjclFFCGenPQG_3072_256(domainParameterSeed,index)`

This is a convenience function that generates FFC domain parameters (p, q, g) where the bit length of p is $L = 3072$ and the bit length of q is $N = 256$, by calling `pjclFFCGenPQ_3072_256` then `pjclFFCGenG_256`. It returns an object with the properties `p`, `q`, `domainParameterSeed` and `counter` produced by `pjclFFCGenPQ_3072_256`, and a property `g` whose value is the big integer returned by `pjclFFCGenG_256`. Both arguments may be omitted.

23.9 `export function pjclFFCGenPQG_2048_256()` `export function pjclFFCGenPQG_2048_256(domainParameterSeed,index)`

This function is like `pjclFFCGenPQG_3072_256`, with $(L, N) = (2048, 256)$.

23.10 `export function pjclFFCGenKeyPair(rbgStateStorage,p,q,g)`

The parameter `rbgStateStorage` is expected to be an ordinary object or a storage object containing the internal state of the DRBG as discussed in Section 21.1. The parameters `p`, `q` and `g` are expected to be big integers representing FFC domain parameters (p, q, g) generated by `pjclFFCGenPQG` or obtained from an external party. The function generates a well-formed FFC key pair (x, y) relative to the domain parameters (p, q, g), as defined above in the preamble of Section 23. With argument checking, an exception is thrown if the security strength of the RBG is less than the nominal security strength provided by the bit lengths (L, N) of (p, q). The function returns an object with properties `x` and `y`, whose values are the big integer representations of x and y.

23.11 `export function pjclFFCValidatePublicKey(p,q,g,y)`

The parameters `p`, `q`, `g` are expected to be big integers representing FFC domain parameters (p, q, g) generated by `pjclFFCGenPQG` or obtained from an external source. The parameter `y` is expected to be a big integer representing a well-formed public key relative to the domain parameters (p, q, g). The function validates the public key as specified in Algorithm 5.6.2.3.1 of [16], returning `true` if it is valid or `false` otherwise.

24 DSA

24.1 Synonyms: using DSA instead of FFC in function names

The library defines the following global variables as synonyms for the names of API functions that begin with `pjclFFC`, replacing `FFC` with `DSA` in each name:

```
export const pjclDSAGenPQ_3072_256 = pjclFFCGenPQ_3072_256
export const pjclDSAValidatePQ_3072_256 = pjclFFCValidatePQ_3072_256
export const pjclDSAGenPQ_2048_256 = pjclFFCGenPQ_2048_256
export const pjclDSAValidatePQ_2048_256 = pjclFFCValidatePQ_2048_256
export const pjclDSAGenG_256 = pjclFFCGenG_256
export const pjclDSAValidateG_256 = pjclFFCValidateG_256
export const pjclDSAGenPQG_3072_256 = pjclFFCGenPQG_3072_256
export const pjclDSAGenPQG_2048_256 = pjclFFCGenPQG_2048_256
export const pjclDSAGenKeyPair = pjclFFCGenKeyPair
export const pjclDSAValidatePublicKey = pjclFFCValidatePublicKey
```

You can use these synonyms to make DSA-related code more readable by people who may be unfamiliar with the FFC acronym and its use by NIST.

24.2 `export function pjclDSASignHash(rbgStateStorage,p,q,g,x,hash)`

The parameter `rbgStateStorage` is expected to be an ordinary object or a storage object containing the internal state of a DRBG as discussed in Section 21.1. The parameters `p`, `q` are `g` are expected to be big integers representing FFC domain parameters (p, q, g), generated by `pjclFFCGenPQG` or obtained from an external source. The parameter `x` is expected to be a big integer representing a well-formed FFC private key relative to the domain parameters (p, q, g). The parameter `hash` is expected to be the bit array encoding of the cryptographic hash of a message to be signed. The function generates a cryptographically random per-message secret k and its inverse k^{-1} mod q, then computes the signature (r, s) on the message, as described in [15, Section 4.6]. It returns an object with properties `r` and `s` whose values are the big integer representations of r and s.

With argument checking, the function verifies that the security strength of the DRBG is not less than the security strength S of the domain parameters and the bit length of the hash is not less than $2S$.

The generation of the per-message secret and the computationally expensive modular inverse operation for computing k^{-1} mod q can be performed ahead of time; then a function called `pjclDSASignHashK` can be used instead of `pjclDSASignHash`, passing the per-message secret and its inverse as the last two arguments. However this may facilitate a timing attack against DSA, as suggested in Section 8 of [17]. For that reason we do not recommend doing it and do not view `pjclDSASignHashK` as an API function.

24.3 `export function pjclDSASignMsg(rbgStateStorage,p,q,g,x,msg)`

The parameter `rbgStateStorage` is expected to be an ordinary object or a storage object containing the internal state of a DRBG as discussed in Section 21.1. The parameters `p`, `q` are `g` are expected to be big integers representing FFC domain parameters (p, q, g), generated by `pjclFFCGenPQG` or obtained from an external source. The parameter `x` is expected to be a big integer representing a well-formed private key relative to the domain parameters (p, q, g). The parameter `msg` is expected to be a bit array that encodes a message to be signed. The function computes a hash of the message using the hash function of the SHA-2 family that produces the shortest output of length greater than or equal to twice the nominal security strength of the domain parameters, throwing an exception if no such function is available. Then it calls `pjclDSASignHash(rbgStateStorage,p,q,g,x,hash)`, passing as the value of `hash` the bit array encoding of the computed hash, and returns its output. (With argument checking, the function `pjclDSASignHash` called by `pjclDSASignMsg` verifies that the security strength of the DRBG is not less than the security strength of the domain parameters.)

24.4 `export function pjclDSAVerifyHash(p,q,g,y,hash,r,s)`

The function `pjclDSAVerifyHash` verifies a DSA signature on a message as described in Section 4.7 of [15], taking the hash of the message as input. The parameters `p`, `q` and `g` are expected to be big integers representing FFC domain parameters (p, q, g). The parameter `y` is expected to be a big integer representing a well-formed public key relative to the domain parameters (p, q, g). The parameter `hash` is expected to be the bit array encoding of the hash of the message. The parameters `r` and `s` are expected to be big integers representing the components (r, s) of the signature. The function returns `true` if verification succeeds, `false` otherwise.

24.5 `export function pjclDSAVerifyMsg(p,q,g,y,msg,r,s)`

The function `pjclDSAVerify` verifies a signature as described in Section 4.7 of [15], taking the message itself, rather then its hash, as input. The parameters `p`, `q` and `g` are expected to be big integers representing FFC domain parameters (p, q, g). The parameter `y` is expected to be a big integer representing a well-formed public key relative to the domain parameters (p, q, g). The parameter `msg` is expected to be the bit array encoding of the message. The parameters `r` and `s` are expected to be big integers representing the components (r, s) of the signature. The function computes a hash of the message using the hash function of the SHA-2 family that produces the shortest output of length greater than or equal to twice the nominal security strength of the domain parameters, throwing an exception if no such function is available, then it calls `pjclDSAVerifyHash(p,q,g,y,hash,r,s)`, passing as the value of `hash` the bit array encoding of the computed hash, and returns its output.

24.6 How to achieve target security strengths with DSA

The present version of the library can be used as follows to compute DSA signatures with 112, 128 and 192 bits of security strength, based on the assignment of security strengths to FFC domain parameters in [10, Table 2, Column 3].

To achieve 192 bits of security strength:

1. Use domain parameters (p, q) with lengths (L, N) such that $L \geq 7680$ and $N \geq 384$. Such parameters cannot be generated by this version of the library, but could be obtained from a trusted source.

2. Set up a DRBG with 192 bits of security strength using `pjclRBG192Instantiate`, and use it for generation of the per-message secret by passing the object containing its internal state as the first argument when calling `pjclDSASignMsg` or `pjclDSASignHash`.

3. Call `pjclDSASignMsg`, which will choose SHA-384 to hash the message, or sign the message using SHA-384 and call `pjclDSASignHash`.

To achieve 128 bits of security strength:

1. Use domain parameters (p, q) with lengths (L, N) such that $L \geq 3072$ and $N \geq 256$, while either $L < 7680$ or $N < 384$. Such parameters can be generated using `pjclFFCGenPQ_3072_256`, obtained from a trusted source, or obtained from an untrusted source and validated using `pjclFFCValidatePQ_3072_256`.

2. Set up a DRBG with 128 bits of security strength using `pjclRBG128Instantiate`, and use it for generation of the per-message secret by passing the object containing its internal state as the first argument when calling `pjclDSASignMsg` or `pjclDSASignHash`.

3. Call `pjclDSASignMsg`, which will choose SHA-256 to hash the message, or sign the message using SHA-256 and call `pjclDSASignHash`.

To achieve 112 bits of security strength:

1. Use domain parameters (p, q) with lengths (L, N) such that $L \geq 2048$ and $N \geq 224$, while either $L < 3072$ or $N < 256$. Such parameters can be generated using `pjclFFCGenPQ_2048_256`, obtained from a trusted source, or obtained from an untrusted source and validated using `pjclFFCValidatePQ_2048_256`.

2. Set up a DRBG with 128 bits of security strength using `pjclRBG128Instantiate`, and use it for generation of the per-message secret by passing the object containing its internal state as the first argument when calling `pjclDSASignMsg` or `pjclDSASignHash`. (This version of the library does not provide a DRBG with only 112 bits of security strength.)

3. Call `pjclDSASignMsg`, which will choose SHA-256 to hash the message, or sign the message using SHA-256 and call `pjclDSASignHash`. (This version of the library does not provide SHA-224.)

25 Diffie-Hellman (DH)

25.1 Synonyms: using DH instead of FFC in function names

The library defines the following global variables as synonyms for the names of API functions that begin with `pjclFFC`, replacing `FFC` with `DH` in each name:

```
export const pjclDHGenPQ_3072_256 = pjclFFCGenPQ_3072_256
export const pjclDHValidatePQ_3072_256 = pjclFFCValidatePQ_3072_256
export const pjclDHGenPQ_2048_256 = pjclFFCGenPQ_2048_256
export const pjclDHValidatePQ_2048_256 = pjclFFCValidatePQ_2048_256
export const pjclDHGenG_256 = pjclFFCGenG_256
export const pjclDHValidateG_256 = pjclFFCValidateG_256
export const pjclDHGenPQG_3072_256 = pjclFFCGenPQG_3072_256
export const pjclDHGenPQG_2048_256 = pjclFFCGenPQG_2048_256
export const pjclDHGenKeyPair = pjclFFCGenKeyPair
export const pjclDHValidatePublicKey = pjclFFCValidatePublicKey
```

You can use these synonyms to make DH-related code more readable by people who may be unfamiliar with the FFC acronym and its use by NIST.

25.2 `export function pjclDH(p,x_A,y_B)`

The function `pjclDH` implements the Diffie-Hellman primitive as specified in Section 5.7.1.1 of [16]. It is used by a party A to compute a secret z shared with a party B.

The parameter `p` is expected to be a positive big integer representing the first component p of a triple of FFC domain parameters (p, q, g); domain parameters q and g are not used in the computation. The parameters `x_A` and `y_B` are expected to be positive big integers representing the private key x_A of A and the public key y_B of B respectively, with y_B expected to be in the range $2 \leq y_B \leq p - 2$.

The function computes $z = {y_B}^{x_A} \bmod p$ and throws an exception if $z = 1$, which cannot happen if x_A is the private key component of a well-formed key pair relative to the FFC domain parameters (p, q, g), and y_B is the public key component of a well-formed key pair relative to those same domain parameters. If $z \neq 1$, the function returns a byte array whose elements comprise the big-endian base-256 representation of z, prefixed with leading zero bytes as needed so that its length is equal to the length of the base-256 representation of p.[3]

[3]In Section 5.7.1.1 of [16], NIST specifies that the output of the DH primitive is to be constructed as specified by the integer-to-byte-string conversion routine of Appendix C.1, which refers to an intended length n of the byte string, without specifying what that intended length is. NIST should have referred instead to the field-element-to-byte-string conversion routine of Appendix C.2, which unambiguously specifies the length of the output when considering that z is an element of the field $\mathbb{Z}_p$.

26 Elliptic curves

26.1 NIST curves

NIST specifies five elliptic curves over prime fields [15, § D.2]: P-192, P-224, P-256, P-384 and P-521. Descriptions of these curves can also be found in [18, §10.2], [19], [20] and [21]. This version of the library implements ECDSA on curves P-256 and P-384. Other NIST and non-NIST curves will 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 [22, § 2.6.2]. This version of the library hardcodes the fact that $a = -3$.

The specification of a Weierstrass curve over a prime field $\mathbb{F}_p$ includes, 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. The fact that the cofactor is 1 is hardcoded in this version of the library. This will change in the future when the library supports other curves.

NIST [15, § 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 26.7) and reduction modulo a Pseudo-Mersenne prime can be performed using [4, Algorithm 14.47] (see also [23, Algorithm 3]). This is what the library does.

26.2 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 $y^2z = x^3 + axz^2 + bz^3$ 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 $y^2 = x^3$.

26.3 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.

26.4 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.

26.5 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.

26.6 `export function pjclModSpecial(x,t,xc,m)`

The function `pjclModSpecial` computes $x \bmod m$, where $m = 2^t - c$, using Algorithm 14.47 of [4], 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.

26.7 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`.

26.8 `export const pjclCurve_P256`

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

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

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

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

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

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

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 base point of the curve, which is also the order of the curve 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 26.4.)

26.9 `export const pjclCurve_P384`

The value of the global variable `pjclCurve_P384` is an object whose properties describe the 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`.

26.10 `export 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`. Recall that, in the library, an affine representation is a special case of a Jacobian representation, as explained in Section 26.4. If `P` is an affine representation, i.e. if the mathematical value of `P.z` is 1, the function returns its first argument with no other processing. Otherwise it computes and returns the affine representation of `P`.

26.11 `export 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 26.4.) This is useful for scalar multiplication, as explained below.

26.12 `export 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 $2P = P + P$.

26.13 `export 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`,[4] 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 26.5 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.

26.14 `export 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`.

[4]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".

27 Elliptic Curve Cryptography (ECC)

The term Elliptic Curve Cryptography (ECC) is used to refer to public-key cryptographic primitives, including Elliptic Curve DSA (ECDSA) and Elliptic Curve Diffie-Hellman (ECDH), that rely on the difficulty of computing discrete logarithms in the group of points of an elliptic curve. The specification of the curve and its chosen base point G play the role of domain parameters and determine the *nominal security strength* of the primitives.

As in FFC, the *nominal security strength* of a primitive is an upper limit on its *actual security strength*, which may also be limited by other factors, such as the security strength of the DRBG used to generate a key pair or the per-message secret used to compute a signature, or the security strength of the hash function used to hash a message being signed. With argument checking, functions implementing ECC primitives verify that these other factors do not reduce the actual security strength of a primitive below its nominal strength. The curves implemented by this version of the library, P-256 and P-384, provide security strengths of 128 and 192 bits respectively, according to [10, Table 2] and [16, Table 2].

An *ECC key pair* relative to an elliptic curve with a chosen base point G of order n is a pair (d, Q), where the private key d is an integer in the range $1 \leq d < n$ and the public key `Q` is a Jacobian representation of the point $Q = dG$.

27.1 `export function pjclCurveSecStrength(curve)`

The parameter `curve` is expected to be an object specifying one of the curves supported by the library. The function returns the nominal security strength of the curve.

27.2 `export function pjclECCGenKeyPair(rbgStateStorage,curve)`

The parameter `rbgStateStorage` is expected to be an ordinary object or a storage object containing the internal state of a DRBG as discussed in Section 21.1. The parameter `curve` is expected to be an object specifying one of the curves supported by the library; this version of the library includes `pjclCurve_P256` and `pjclCurve_P384`, which provide 128 and 192 bits of security strength respectively. With argument checking, the function throws an exception if the security strength of the DRBG is less than that of the curve.

The function returns an object containing two properties `d` and `Q` representing an ECC key pair (d, Q) relative to the curve and its chosen base point G, `d` being the big integer representation of d and `Q` an affine representation of $Q = dG$. (Recall that an affine representation is a special case of a Jacobian representation, as explained above in Section 26.4.)

27.3 `export function pjclECCValidatePublicKey(Q,curve)`

The function `pjclECCValidatePublicKey(Q,curve)` implements Algorithm 5.6.2.3.2 of [16] 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. The last step is unnecessary if the cofactor is 1, since in that case (with the notations of Algorithm 5.6.2.3.2) n is the order of the curve, and therefore $n\mathcal{Q} = \mathcal{O}$. This hardcodes the fact that the cofactor is 1 in NIST curves over prime fields, and will change in the future if the library supports curves with other cofactors.

28 ECDSA

28.1 Synonyms: using ECDSA instead of ECC in function names

The library defines the following global variables as synonyms for the names of API functions that begin with `pjclECC`, replacing `ECC` with `ECDSA` in each name:

```
export const pjclECDSAGenKeyPair = pjclECCGenKeyPair
export const pjclECDSAValidatePublicKey = pjclECCValidatePublicKey
```

28.2 `export function pjclECDSASignHash(rbgStateStorage,curve,d,hash)`

The parameter `rbgStateStorage` is expected to be an ordinary object or a storage object containing the internal state of a DRBG as discussed in Section 21.1. The parameter `curve` is expected to be an object specifying one of the curves supported by the library; this version of the library includes `pjclCurve_P256` and `pjclCurve_P384`, which provide 128 and 192 bits of security strength respectively. The parameter `d` is expected to be a nonnegative big integer, whose value is the private key to be used for signing the message. The parameter `hash` is expected to be the bit array encoding of the cryptographic hash of a message to be signed. The function generates a cryptographically random per-message secret k and its inverse $k^{-1} \bmod n$, where n is the order n of the base point of the curve (and of the curve, since the cofactor is 1), represented by the big integer `curve.n`. Then it computes the signature (r, s) on the message as described in [15, Section 6.4]. It returns an object with properties `r` and `s` whose values are the big integer representations of r and s.

With argument checking, the function verifies that the security strength of the DRBG is not less than the security strength S of the curve and the bit length of the hash is not less than $2S$.

As is the case for DSA, the generation of the per-message secret and the computationally expensive modular inverse operation for computing its inverse modulo n can be performed ahead of time; then a function called `pjclECDSASignHashK` can be used instead of `pjclECDSASignHash`, passing the per-message secret and its inverse as the last two arguments. However, for consistency with DSA, we do not view `pjclECDSASignHashK` as an API function in this version of the library.

The implementation of `pjclECDSASignHashK`, and of `pjclECDSAVerifyHash` below, hardcodes the fact that the order n of the generator has the same bit length as the prime p that defines the field, which is true for all the NIST curves over prime fields. This may change in the future as other curves are included in the library.

28.3 export function pjclECDSASignMsg(rbgStateStorage,curve,d,msg)

The parameters `rbgStateStorage`, `curve` and `d` are as those of `pjclECDSASignHash`. The parameter `msg` is expected to be a bit array that encodes a message to be signed. The function computes a hash of the message using the hash function of the SHA-2 family that produces the shortest output of length greater than or equal to twice the security strength of the curve, throwing an exception if no such function is available, then it calls `pjclECDSASignHash(rbgStateStorage,curve,d,hash)`, where the value of `hash` is the bit array encoding the computed hash, and returns its output. (With argument checking, `pjclECDSASignHash` verifies that the security strength of the DRBG is not less than the security strength of the curve.)

28.4 export function pjclECDSAVerifyHash(curve,Q,hash,r,s)

The function `pjclECDSAVerifyHash` verifies an ECDSA signature on a message taking the hash of the message as input. The parameter `curve` is expected to be an object specifying one of the curves supported by the library. The parameter `Q` is expected to be a Jacobian representation of a point, to be used as the public key. The parameter `hash` is expected to be the bit array encoding of the hash of the message. The parameters `r` and `s` are expected to be big integers representing the components (r, s) of the signature. The function returns `true` if verification succeeds, `false` otherwise.

28.5 export function pjclECDSAVerifyMsg(curve,Q,msg,r,s)

The function `pjclECDSAVerifyMsg` verifies a signature taking the message itself, rather than its hash, as input. The parameter `curve` is expected to be an object specifying one of the curves supported by the library. The parameter `Q` is expected to be the Jacobian representation of a point, to be used as the public key. The parameter `msg` is expected to be the bit array encoding of the message. The parameters `r` and `s` are expected to be big integers representing the components (r, s) of the signature. The function computes a hash of the message using the hash function of the SHA-2 family that produces the shortest output of length greater than or equal to twice the security strength of the curve, throwing an exception if no such function is available, then it calls `pjclECDSAVerifyHash(curve,Q,hash,r,s)`, where the value of `hash` is the bit array encoding the computed hash, and returns its output.

29 Elliptic Curve Diffie-Hellman (ECDH)

29.1 Synonyms: using ECDH instead of ECC in function names

The library defines the following global variables as synonyms for the names of API functions that begin with `pjclECC`, replacing `ECC` with `ECDH` in each name:

```
export const pjclECDHGenKeyPair = pjclECCGenKeyPair
export const pjclECDHValidatePublicKey = pjclECCValidatePublicKey
```

29.2 `export function pjclECDH(curve,d_A,Q_B)`

The function `pjclECDH` implements the Elliptic-Curve Diffie-Hellman (ECDH) primitive as specified in Section 5.7.1.2 of [16], except that the cofactor is not used, because the curves supported by this version of the library are NIST curves over prime fields, where the cofactor is 1. The function is used by a party A to compute a secret z shared with a party B.

The parameter `curve` is expected to be an object specifying one of the curves supported by the library, either `pjclCurve_P256` or `pjclCurve_P384`. The parameter `d_A` is expected to be a big integer representing the private key d_A of party A, and the parameter `Q_B` a Jacobian representation of the public key Q_B of party B.

The function computes the point $P = d_A Q_B$ and throws an exception if P is the point at infinity, which cannot happen if d_A and Q_B are the private and public key components of two ECC key pairs relative to the curve and its chosen base point. If P is not the point at infinity, the function returns the base-256 representation of the x coordinate of the affine representation of P as a byte array.

References

[1] 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.

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

[3] 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.

[4] 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/.

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

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

[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] H. Krawczyk and P. Eronen. HMAC-based Extract-and-Expand Key Derivation Function (HKDF). RFC 5869, May 2010. http://tools.ietf.org/html/rfc5869.

[9] Lily Chen. Recommendation for Key Derivation through Extraction-then-Expansion. NIST Special Publication 800-56C, November 2011. http://nvlpubs.nist.gov/nistpubs/Legacy/SP/nistspecialpublication800-56c.pdf.

[10] 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.

[11] K. Moriarty (Ed.), B. Kaliski and A. Rusch. PKCS #5: Password-Based Encryption Standard Version 1.5, January 2017. Informational RFC 8018. https://tools.ietf.org/html/rfc8018.

[12] B. Kaliski. PKCS #5: Password-Based Cryptography Specification Version 2.0, September 2000. http://tools.ietf.org/html/rfc2898.

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

[14] W3C. Indexed Database API. https://www.w3.org/TR/IndexedDB/.

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

[16] Elaine Barker, Lily Chen, Allen Roginsky, and Miles Smid. Recommendation for Pair-Wise Key Establishment Schemes Using Discrete Logarithm Cryptography. NIST SP 800-56A Rev. 2. http://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-56Ar2.pdf.

[17] Paul C. Kocher. *Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems*. Springer Berlin Heidelberg.

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

[19] National Security Agency. Mathematical routines for the NIST prime elliptic curves. April 05, 2010. Available with a browser security warning at https://www.iad.gov/iad/library/ia-guidance/ia-solutions-for-classified/algorithm-guidance/mathematical-routines-for-the-nist-prime-elliptic-curves.cfm and without browser warning at http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.204.9073&rep=rep1&type=pdf.

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

[21] 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.

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

[23] 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.