Improvement of a JavaScript cryptographic library performance with big numbers

Julien von Felten

School of Computer and Communication Sciences

Decentralized and Distributed Systems lab

Master Project Report

June 2019

Responsible
Prof. Bryan Ford
EPFL / DEDIS

Supervisor
Gaylor Bosson
EPFL / DEDIS

Contents

1 Introduction and background

The goal of this project is to optimize the library Kyber[1]. Kyber is a package which provides a toolbox of advanced cryptographic primitives, implemented in Go. It is used in the Cothority project[2]. This project is being developed by the DEDIS team which is working on projects related to large-scale collective authorities (cothorities), which distribute trust among a number of independent parties to allow scalable self-organizing communities[3]. A part of Cothority, named Cothority client library, was implemented in JavaScript to offer a socket interface and the status of the SkipChain Explorer[4]. Kyber was also adapted in JavaScript to have link verifications in this status. The problem with this adaptation is its slowness. Indeed, as a cryptography library needs to use big integers and JavaScript did not have any big integer type, Kyber is using bn.js[5] for the big integers as the native type number supports only 53 bits. This turns out to be very slow especially when it comes to elliptic curve operations. Therefore, the link verifications are turned off on the SkipChain Explorer.

This project aims to improve the performance of the pairing cryptography such that it can be applied to the SkipChain Explorer so that the links could be turned on. The general idea is to find different possible optimizations and implement them. A benchmark is also implemented in order to compare the different optimizations and the actual library. The first optimization chosen is the use of the BigInt type instead of the bn.js library. The second one is to decrease the number of modulo operations. The last is to set up memory pools.

2 Design

2.1 Benchmark

To be consistent with all the optimizations, a benchmark has to be created which will serve as reference. To select the correct API, there are two constraints to satisfy: precision and overhead. Knowing that a verification of a key is currently between 150 and 200ms, the precision should be smaller than milliseconds. There are different options for the benchmarking that can be considered:

- Date object from JavaScript library
- Console API with the console.time function

[1]`https://github.com/dedis/kyber`
[2]`https://github.com/dedis/cothority`
[3]`https://www.epfl.ch/labs/dedis/`
[4]`http://status.dedis.ch/#/`
[5]`https://www.npmjs.com/package/bn.js`

- Performance MDN API
- Process API with the process.hrtime function

It is expected that the more precision the API has, the more overhead it will produce. The tests will compute the time for one signature and for one verification. It will also compute the overall time of n signatures, where $n = [2, 10, 100, 500, 1000]$, the objective being to avoid too much overhead. Since minimum overhead and maximum precision is wanted, the decision is not to take the Date object and the Console API because it is not precise enough (milliseconds). The Process API is also not taken as it has a precision of a nanosecond which will produce too much overhead[6]. Hence, the API chosen is Performance MDN API[7]. This API is straightforward to use and only the timing metrics is required. This API is already in JavaScript, there is no need to import anything. A performance.mark is used to create a timestamp. Then performance.measure can be used with two marks as argument to produce a measure of time in milliseconds. The measures can be found using the method getEntriesByName. The last function used is the clearMeasures which deletes the measures.

To keep a consistency of the results, the same benchmark will be used for every optimization. It will compute not only many signatures, the number of aggregations needed for the verifications and many verifications but also the average, the minimum, the maximum, the standard deviation and the median for signatures and verifications. These values will be useful in different optimizations.

2.2 From bn.js to BigInt

It was decided to change the bn.js to BigInt for the first optimization. The problem with the kyber library is that it is using huge numbers which can not be represented with a standard number in JavaScript which uses 53 bits. To thwart this problem, the bn.js is currently used but this library uses arrays to represent big numbers which is not performant in memory and computation time. Hence, it is a priority to find another method than bn.js to represent big integers. There are two different options:

- The new BigInt type
- Concatenate two numbers to create a big integer

Fortunately, a new type came in early 2019 which is the BigInt type[8]. It was a perfect timing to try this new feature from JavaScript. It is useful for representing numbers bigger than 2^{53}.

[6]https://www.stefanjudis.com/today-i-learned/measuring-execution-time-more-precisely-in-the-browser-and-node-js/

[7]https://developer.mozilla.org/fr/docs/Web/API/Performance

[8]https://v8.dev/features/bigint#bigint

The BigInt type is at the stage 4 of proposal[9]. When the stage 4 is accepted, it means that it will be added in the ECMAScript (ES)[10] which is the standard for JavaScript. Hence, it is expected to have the BigInt in the release of June 2020[11]. Currently, ESNEXT version is used as BigInt type is not in the standard ES18 or 19. Since the kyber library uses numbers, changing them to BigInt is not a big issue. The BigInt library does not have all the methods implemented as bn.js but it is sufficient to implement the basic functions that are needed in this project as for example: signum, modulus, power invmod with the EGCD algorithm, comparteTo, toBytes, testn. The implementations of these functions will be detailed in the implementation section. One problem with the BigInt type is that the supported operations on BigInt are not constant in time, therefore the BigInt type is not suitable for use in cryptography[12]. This may lead to possible timing based attack problem on the library. It was decided to use it anyway since on the website, there is no check any more because it takes too much time to verify it. Having these checks would be better than having no checks.

The package bigint-buffer[13] to convert BigInt to and from buffers was chosen. It is a simple package to use and seems to work well.

2.3 Optimization of modulus

After the first optimization, the tool used is the browser with his performance method found in the Inspect element". To use it in a clear way, the line "mode: "development"" must be added in webpack.config.js in module.exports. This will make clear which function is used with the time spent in each function.

Then, when running the performance against the benchmark, many things can be obtained: memory sketch, diagram of all the functions used at any given time of the execution, Bottom-Up and Call-Tree breakdown of the benchmark. A picture of this tool can be seen in Figure 1.

Bottom-Up and Call-Tree are really useful to see where the program spends the most time which is usually the area where the optimization can be found and a lot of gain can be made. Bottom-Up is the total time spent of the program in each function. As the name suggest, it is a bottom-up tree which starts with the lowest level functions. This is why it can be seen on Figure 1 that the functions mod, mul, sub, add from gfp.ts are the most used. In the Call-Tree, it is the opposite, it starts from the biggest function (which is the Event:click on Figure 1) and this function can be broken down

[9]https://github.com/tc39/proposal-bigint

[10]https://en.wikipedia.org/wiki/ECMAScript

[11]https://github.com/tc39/proposals/blob/master/finished-proposals.md

[12]https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/BigInt#Usage_recommendations

[13]https://www.npmjs.com/package/bigint-buffer

into smaller pieces. Self-Time represents the amount of time that was spent in the function at the current level of the call tree. It is the time needed to execute the code in the function itself. Total-Time is the self-time plus the amount of time it took to execute the code in functions that the current level calls. For the optimization of modulus, the Bottom-Up diagram is used. After analysing it, the mod function uses 55% of the self time. The mod function in the gfp.ts was basic, nothing can be changed in this function. However, one of the properties of the modular arithmetics can be used in a way to reduce the number of calls of the function. The property is as follows:

$$(A \mod n * B \mod n) \mod n \equiv (A * B) \mod n$$

The symbole * stands for addition, multiplication, substraction, modulo, power. Therefore, this property can be used to decrease the number of modulo call. Since the modulo used in most of the program is the constant p in the file constants.ts, the goal of this optimization is to remove all the modulo that are not needed. Hence, in curve-point.ts, gfp2.ts, gfp6.ts, gfp12.ts there are moduli that can be removed. Then, the idea is to use these moduli in higher level functions such as in twist-point.ts or opt-ate.ts files.

After the first try of the reduction of the number of moduli calls, a check was performed to analyse whether the self time was lower and whether it could find where the most time was spent for the moduli in order to try to find different implementation for this optimization. This optimization was the fastest to implement among the optimizations.

2.4 Pool memory

The next optimization is to use a memory pool with mutable classes instead of having immutable classes with functions returning new object. A memory pool is a group of objects that has been created but will not be destroyed by the garbage collector. When an object is needed, a call to the pool will provide one instead of creating it. If the pool is empty, it will double the side of the number of objects and return one.

The number of created gfp and gfp2 are the order of hundreds of millions. Indeed, at each function call in gfp.ts and gfp2.ts, it creates a new object. So the next optimization is to use memory pools and removing the immutability of each class of the gfps in order to decrease the number of created objects. One drawback of memory pool is that it increases the code complexity with the management of the pool and temporary variables with the recycle function.

A package that implements a memory pool was found: deePool[14]. As it says, "There are no configuration options to tweak behavior, but it does

[14]`https://www.npmjs.com/package/deepool`

one thing and one thing well". This package must be used with caution since there is no error handling. It is a basic memory pool package to use: after creating the pool, the function pool.use is called to get an object of the pool, pool.recycle to return the object to the pool. This library was chosen because of the simplicity of its code.

A small optimization with the pool memory was the number of temporary variables used in each function. To reach a minimum number of the temporary variables, an algorithm was used which will be described later in the implementation section.

3 Implementation

3.1 Benchmark

The idea of the implementation for the benchmark is to measure the time of one signature and of one verification. In addition, the time of all signatures and all verifications is measured regardless of the overhead. Since the overhead should be constant, the improvement will be obvious to notice. Using the performance.mark("name of mark") before and after the line of code of the signature and verification, the time needed for these functions is computed. Then a measure is created with two marks using performance.measure("name measure", "name mark one", "name mark two"). This measure has different attributes, only the duration attribute is used in the benchmark which is a DOMHighResTimeStamp which represents the duration of the measure.

At the end of both functions calculating signatures and verifications (function names are sign and verify in the monitoring.js), two measures are produced: one is a measure of the total time of each function, the second is all measures of each signature or each verification. For the latter, statistics are done such as total time, average, minimum, maximum, standard deviation and median. The ratio between the total time for the verifications and for the signatures is also computed. This constant could show whether there is a better improvement in the verify function or in the sign function.

Before the verifications, an aggregation is done with a mask for a single key "0b1" which is used in the aggregateSignatures function with one signed key. A check is also performed to know whether the verify function is valid. To perform the verification, a mask, the signed keys and the public keys are required. This is given by the signature function as return arrays.

For this benchmark, two buttons are added in a css file: the first button is the initial test of the repository, the second button is the benchmark which computes signatures and verifications. It displays in the console all information related to the benchmark. It performs the operations with 2, 10, 100, 500 and 1000 keys which is represented by the maxjs array. The signed message is ”abc”. The length of the signed message does not influence the obtained values.

3.2 From bn.js to BigInt

To avoid waiting a long time to be sure that the benchmark is working well, a third button was created in the css file which performs only two signatures and verifications. This button is not supposed to be used for the tests in order to compare values from one optimization to another, it is only meant to be used for correctness of verification after changes made.

The first modification to do is the tsconfig.json file. The target was ”es6” which does not support the BigInt yet, therefore it has to be changed to ”ESNext”. This will allow to compile the BigInt type.

The idea of this implementation is to replace the BNType by the BigInt type in the pairing package. All imports from bn.js were removed initially which produced errors to solve. The first file changed was gfp.ts, which is the lowest abstraction level, since everything is built on this file. The BigInt must not be declared with the ”new” word; it is used the same way as other primitive types. There is no method such as cmpn, pow, umod, invm, cmp, toArrayLike in the library. They all have to be implemented. To determine the sign, it is obvious that the BigInt has to be compared with 0: if it is greater, smaller or equal than 0. The function pow is implemented using an accumulator initialized to 1 and one for loop of k iterations: for a^k, the accumulator is multiplied k times with a. The mod function is $c = a \mod b$, where c is the positive value. In the BigInt library, c keeps the same sign as a, for that reason, a check is needed for the sign before returning the value. To compare two gfps, it is the same idea as the sign function: the current object has to be compared with the argument. As said in the design section, the package bigint-buffer is used to convert a BigInt to a buffer: in the function toBytes(), the function toBufferBE from this package is used. To implement the egcd which is needed for the inverse, an example was found[15] and adapted to the BigInt. The inverse is the first return value of the egcd function.

The gfp2.ts file needs minor adaptations. The gfp12.ts has a testn(n) method which returns the n^{th} bit. To implement this function, a mask, set to one, is shifted to the left n times then ”and bitwise” with the bit string. This will return zero if the n^{th} bit is set to 0, else it will return something different than 0. To get the bit length of the BigInt, the BigInt must be converted to binary string using the function BigInt.toString(2) which has a length attribute.

In the constructors of bn.ts and curve-point.ts, the type of k, x, y, z and t must be checked whether it is undefined or a BigInt because BigInt can not

[15] `https://github.com/lpcsmath/egcd/blob/master/JavaScript/egcd.js`

be mixed with other types. The unmarshal functions have to be adapted using the toBigIntBE from the package bigint-buffer. The file point.ts does also need the package bigint-buffer for the bytes.

The scalar.ts file has more code to change. First, in the constructor, as the original code uses the umod from the bn.js library, the value must be positive. Hence, if the value is negative, p must be added to the result of the function. This is the case for other methods such as add, sub, neg, div, mul. The inverse is computed again with the egcd using the first return value. In the function marshalBinary(), the function toBufferBE from bigint-buffer replaces the function toArrayLike from the bn.js package. The function unmarshalBinary needs a check on the type since the function toBigIntBE accepts only buffers as argument, therefore if the argument is of the type "string", it has to be converted to buffer type using Buffer.from function. Finally, the buffer can be converted to BigInt.

The last file to adapt is the tonnelli-shanks.ts in the utils package. This was the hardest part of this implementation. First, all function operators were changed: from add, sub, div, mul, mod, eq, and, shrn to +, -, /, *, %, ===, &, >> respectively. Then the function cmp(a, b) was implemented which compares two BigInts. The last part to adapt in this file was the powMod(a, e, p) function. Three methods were found for this implementation which presents similar results. The first method is powMod0 which uses a while loop, divides the exponents by two, multiplies 'a' by 'a' and reduces to modulo p at each iteration[16]. The second method is powMod1 which is a recursive function that divides the exponent by two[17]. The last method is powMod2 which uses the bit string of the exponent. It iterates on this bit string, multiplies the accumulator, initialized to the value a, by itself at each iteration and reduces to modulo p. Then if the bit is 1, it multiplies the accumulator by a. In the implementation, the powMod2 was chosen because it seems to be the faster among the three implementations. These methods will not be compared since they have similar results and it can not be determined which method is exactly the fastest among these three.

When instantiating a BigInt with a big number greater than 2^{53}, one must be careful to use BigInt("big number"). If the call is made without quotes, it will be seen as a number, therefore the BigInt will be truncated to the 53 first bits of the number.

After all these modifications, the final change to make is to adapt all tests to the BigInt implementation.

3.3 Optimization of modulus

Since the first optimization was a great success, it was decided to keep it and find another optimization to implement in addition to the first one.

[16]It is the method number 3 from `https://helloacm.com/compute-powermod-abn/`

[17]It is the method number 4 from `https://helloacm.com/compute-powermod-abn/`

As already said in the design section, the number of moduli used had to be limited as much as possible. This optimization was faster to implement in comparison to the other two. It took around one week and a half to implement this one. All modulo function calls were removed in the files gfp2.ts, gfp6.ts and gfp12.ts. Unfortunately this can not be done that easily, when removing the moduli, the operations such as multiplication and exp have a complexity depending on the length of the operands. Hence, removing all moduli from these functions are making things worse and the tests do not end. This is why the moduli in these two functions are kept. The mod operator is also implemented for these three files.

In curve-point.ts, all mod(p) were removed in the function add, dbl and makeAffine except for the return values. This means that 24 mod(p) operations out of 38 were deleted. Unlike the file opt-ate.ts, a mod(p) was added at each line where a square or exp operation was previously. The moduli in this file do not add much time because it is the function at the higher abstraction level that is called fewer times. The fact that in this file the numbers to compute are smaller, makes the program run faster.

Then it was realised that the moduli in the multiply (from gfp2.ts, gfp6.ts and gfp12.ts) was still expensive time wise. The implementation of the function was changed from mul(a), where a is either gfp2 or gfp6, to mul(a, b?) where b is a boolean. If this boolean is set to true, then the moduli in the function are not computed. Hence, it could be decided in particular places if a modulo has to take place. This optional boolean was chosen to avoid changing every mul call of the program but only particular ones. When mul and square from gfp12 is called, there is no modulo that is performed. When mul from gfp6 is called, regardless is the boolean value, there is no modulo performed in the call of gfp2. This could gain around 4 to 5 ms in the optimization. In the results section, this small optimization will not be analysed.

The tests must be adapted to the fact that functions are not returning values modulo p any more.

At this point, the results encountered were satisfying, it was decided to move on to another optimization. However, this optimization can be taken further to check whether each exact mod is needed. The problem is that the benchmark is not precise enough to know if small changes are improving the program.

3.4 Pool memory

As said in the design section, there are many gfp, gfp2, gfp6 and gfp12 created. To avoid this problem, the memory pool will be used. The first problem encountered was cyclic import. To solve this issue, the pool of gfp, gfp2, gfp6, gfp12 must be created in the gfp.ts, gfp2.ts, gfp6.ts and gfp12.ts file respectively. To create the pool, the deePool.create() function using the object factory is called:

```
export const GfPPool1 = deePool.create(
    function makeGFP1(){
        return new GfP(0n)
    })
```

The object factory must be a function that produces a single new empty object. This produces a new pool instance with a gfp of value 0. The same code is used and adapted to gfp2.ts, gfp6.ts, gfp12.ts. The objects from the pools should be used to replace the temporary variables in each function.

After the creation of the pools, all functions in gfp.ts are adjusted such as the modifications are done in place in the current object "this". The arguments must not be changed inside the function. For example, the line was previously t = a.mul(b) which multiplies a with b and returns a new gfp t. Now it is written t.mul(a, b), it computes the multiplication of a and b and writes the result in t which is a temporary variable that came from the pool. The final code for the multiplication function is as follows:

```
mul(a: GfP, b: GfP): GfP {
    this.v = a.v * b.v
    return this
}
```

This method is used exactly in the same way for the other functions such as add, sub, sqr, pow, mod, invmod, negate and shiftLeft. Three new functions are implemented: setValue(BigInt), copy(GfP) and clone(GfP).

These changes are applied to gfp2.ts, gfp6.ts and gfp12.ts files too. A few functions that needed temporary variables had to be modified to use the pools. It was the case for mul in gfp2 for example. Instead of declaring a variable with the initialization for example: let tx = this.x.mul(a.y); Using the GfPPool1:

```
    let tx : GfP = GfPPool1.use()
    tx.mul(a.x, b.y)
    ...
    GfPPool1.recycle(tx)
```

The use of the pool in gfp2.ts, gfp6.ts and gfp12.ts are in functions such as mul, square, invert, frobeniusP2, mulTau and exp. If the current object has to be modified, it is allowed to use it as argument: this.x.mod(this.x, p): this will perform a modulus operation on this.x and return the result in this.x. The problem encountered during this optimization was that the function mulXi could not be used with the same variable: t.mulXi(t). The tests failed. This was due to the fact that a.x (the argument) was used for the computation of "this.y". Hence, a temporary variable was needed anyway which is a copy of the argument assuring that t.mulXi(t) can be used that way. The choice to return ”this” at most of the functions is justified by the fact that the functions calls can be chained. This avoids having 10 lines to make 10 operations, but all these operations stand on one line of code: t.mul(a, b).mul(t, c).mod(t, p)...

The last problem that was encountered with the gfp classes was the static ZERO and ONE constant. Since the classes are now mutable, a particular attention has to pay in regards to the references. When an empty object (GfP6, GfP12, CurvePoint, TwistedPoint, ...) is created, it sets default values with ZERO or ONE constants. This means that the object has the same reference as the static object. When this object is modified, the static object is modified at the same time. To avoid this problem, the code changed from:

```
private static ZERO = new GfP2(zeroBI, zeroBI);

public static zero(): GfP2 {
    return GfP2.ZERO;
}
```

to:

```
public static zero(): GfP2 {
    return new GfP2(zeroBI, zeroBI)
}
```

The same is done for ONE in gfp2-ts, ZERO and ONE in gfp6.ts and gfp12.ts, generator in curve-point.ts and twist-point.ts.

The constructors of GfP6, GfP12 and TwistedPoint are also modified to avoid this referencing problem: if a new GfP6 is created using 3 GfP2 already created, it will have the same references and if the GfP6 or any of the GfP2 is modified, the other will be transformed. Hence, to avoid this problem, a deep copy of the GfP2 has to be done. For each argument of the constructor, a deep copy is performed with the function getX, getY, getZ and getValue. For example, the line: ”this.x = x instanceof GfP2? new GfP2(x.getX().getValue(), x.getY().getValue()) : GfP2.zero()” is to copy deeply the x argument of the GfP6.

This means that the affectation operator (’=’) can not be used any more to set a GfP, GfP2, GfP6 or GfP12 to another one because of this referencing problem. Therefore, every class has his function copy which makes a deep copy of the object. For example, in the neg function of twist-point.ts file, which was implemented ”this.x = a.x” before, now it must be used with the copy function ”this.x.copy(a.x)”.

The files curve-points.ts, bn.ts and twist-point.ts need to be adapted with what discussed above. Nothing else is needed.

In the file opt-ate.ts, one must be careful to return an object to the pool after the use. In the function lineFunctionAdd for example, in the previous optimization, a temporary variable was returned. Using the pool, an object can not be taken from the pool, returned to the function but not returned to the pool. This might create a bigger pool than expected with leakage of memory. Therefore, the return values of the function have to be created via "new". In lineFunctionAdd, a, b and c are created with "let a : GfP2 = new GfP2(0n, 0n)" and the set to the value needed.

The last small optimization was the number of temporary variables used in every function. Since each function was using too many temporary variables, reducing the number of temporary variables was explored. A small algorithm was used: if two temporary variables were used in the same function but the first time the second variable was used was after the last time the first variable was used, then these two variables could be merged into one without any problem. This algorithm is not optimal. It could reduce significantly the number of temporary variables which results in smaller pools and memory usage. It results in a faster computing time which will not be discussed in the results section.

The tests must also be adapted to the change of the function signatures and mutable classes.

4 Evaluation and results

The goal of this project is to try different optimizations and to show if they work. The browser Chromium on Linux is used to compare the different optimizations and the computer is an Asus ROG GL553 with a dual boot Windows and Linux. The bundle was compiled with the development option for every test. Milliseconds might be gained using the production option since the bundle is minified[18]. The tests are all run in the same conditions: two tabs are opened, the first one is to write the results and the second one is to run the tests. The second tab is always closed and reopened again to avoid any problem related to the memory. The tests depend also on the computer, the OS and the scheduler, hence, it can be noticed a variation of a few milliseconds (1-2 ms usually).

4.1 General results

In this subsection, the results of each optimization will be presented with graphs using the average, minimum and maximum values. Then the speedup of each optimization will be compared with the original program which is called Master in the tables and figures.

The Table 1 represents the minimum, maximum and average value obtained on the Chromium Linux browser with 1000 key verifications on the Master and on the three different optimizations. The speedup can be seen

[18]`https://webpack.js.org/guides/production/`

in the Table 2. In the first line of this table, it shows the speedup of each optimization in comparison to the Master. The second line is the ratio of the total time for 1000 key verifications over the total time for 1000 key signatures. This ratio can be helpful to identify if an optimization improved one of the two functions more than the other.

A clear optimization can be seen using the BigInt implementation. This is the best speedup obtained for one optimization which is of the magnitude of 3.1. It can be explained quite easily. The Master program is using bn.js library for big numbers. In this library, the big numbers are represented as arrays which is an expensive type in computation and memory usage. Therefore, using the new BigInt native type makes the program run faster. This ratio from Table 2 is quite similar between Master and BigInt because the sign method and verify both benefit from the implementation of the BigInt.

	Master	BigInt	Modulus	Pool
Minimum time	166.3	54.91	33.36	37.46
Average time	180.26	58.21	36.92	41.08
Maximum time	222.79	73.12	55.9	59.26

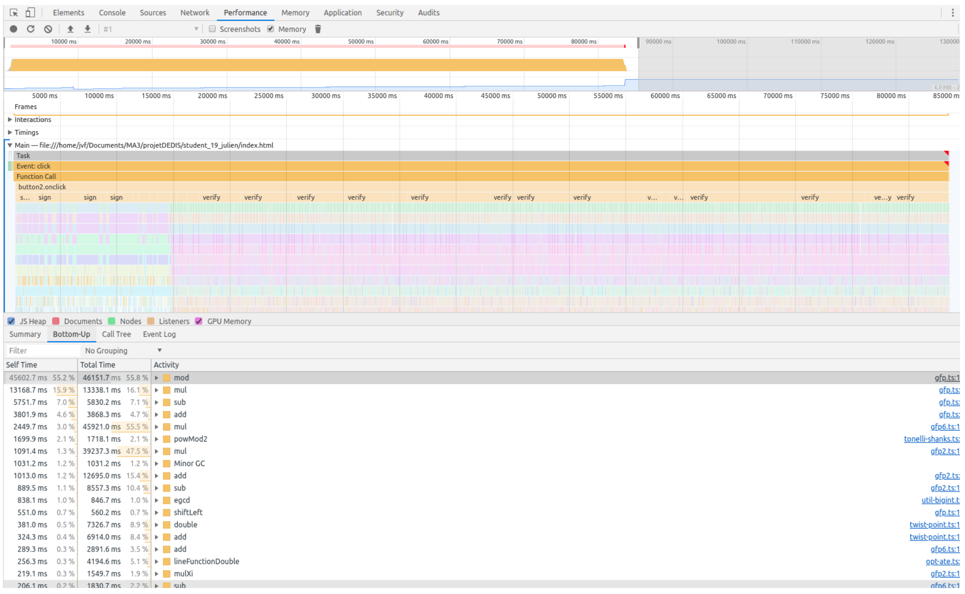
Table 1: Table of the minimum, average and maximum values in milliseconds for each optimization for 1000 key verifications.

	Master	BigInt	Modulus	Pool
Speedup of verification in comparison with Master	1	3.1	4.76	4.29
Verification total time / signature total time	5.5	5.08	3.21	3.6

Table 2: A comparison of the speedup of each optimization with the Master and the ratio between the total time to sign and verify 1000 keys.

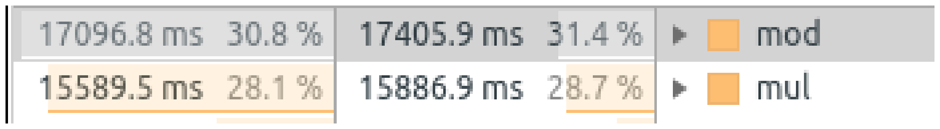
To choose the next optimization, as said above, the performance tool of the browser was used. The mod operation from gfp.ts file is used for around 21470ms which represents 55% of the total time of the program.

Comparing Figure 1 and 2, it can be seen that the mod operation went from 55% to 30% of the total time of the program. The mul operation is taking slightly more time since the numbers computated are bigger since fewer moduli are used. This increase in time for the mul operator is negligible in comparison to the reduction of the time for the mod operator. Therefore, a great improvement is noticed which results in a 4.76 speedup in comparison to the Master optimization. The speedup from BigInt optimization to



Self Time		Total Time		Activity	
45602.7 ms	55.2 %	46151.7 ms	55.8 %	mod	gfp.ts:1
13168.7 ms	15.9 %	13338.1 ms	16.1 %	mul	gfp.ts:
5751.7 ms	7.0 %	5830.2 ms	7.1 %	sub	gfp.ts:
3801.9 ms	4.6 %	3868.3 ms	4.7 %	add	gfp.ts:
2449.7 ms	3.0 %	45921.0 ms	55.5 %	mul	gfp6.ts:1
1699.9 ms	2.1 %	1718.1 ms	2.1 %	powMod2	tonelli-shanks.ts:
1091.4 ms	1.3 %	39237.3 ms	47.5 %	mul	gfp2.ts:1
1031.2 ms	1.2 %	1031.2 ms	1.2 %	Minor GC	
1013.0 ms	1.2 %	12695.0 ms	15.4 %	add	gfp2.ts:
889.5 ms	1.1 %	8557.3 ms	10.4 %	sub	gfp2.ts:1
838.1 ms	1.0 %	846.7 ms	1.0 %	egcd	util-bigint.t
551.0 ms	0.7 %	560.2 ms	0.7 %	shiftLeft	gfp.ts:1
381.0 ms	0.5 %	7326.7 ms	8.9 %	double	twist-point.ts:1
324.3 ms	0.4 %	6914.0 ms	8.4 %	add	twist-point.ts:1
289.3 ms	0.3 %	2891.6 ms	3.5 %	add	gfp6.ts:1
256.3 ms	0.3 %	4194.6 ms	5.1 %	lineFunctionDouble	opt-ate.ts:
219.1 ms	0.3 %	1549.7 ms	1.9 %	mulXi	gfp2.ts:1
206.1 ms	0.2 %	1830.7 ms	2.2 %	sub	gfp6.ts:1

Figure 1: The result of the performance method with the BigInt optimization.



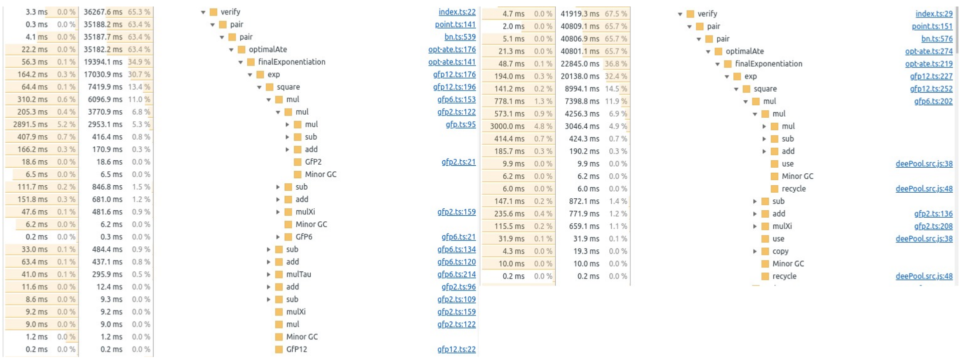
17096.8 ms	30.8 %	17405.9 ms	31.4 %	mod
15589.5 ms	28.1 %	15886.9 ms	28.7 %	mul

Figure 2: The result of the performance method with the Modulus optimization for the mod and mul functions.

Modulus optimization is around 1.54. The ratio in Table 2 is also smaller which means that the improvement is better for the verification function than for the signing. This is explained because the sign method from the package sign/bdn/ uses the function mul from curve-point.ts which does not use many moduli. Hence, the sign function does not benefit much from this optimization, unlike the verify method which uses many moduli. For this reason, the ratio went from 5.5 to 3.21.

Surprisingly, the last optimization was not successful. Good results from this last optimization were expected. Indeed, the number of created GfP objects in the modulus optimization was around 318,000,000 and the number of GfP2 was around 90,400,000. These numbers are enormous, hence, a decrease of these numbers using a memory pool could be expected and thus, making the program faster. The decrease of these numbers was observed but unfortunately the program is slower with this optimization.

In comparison, the numbers of created GfP and GfP2 objects went down to 8,200,000 and 4,080,000 respectively. This shows that these numbers could be decreased significantly. Looking at the result, the Modulus optimization is on average 4 to 5 ms faster than the Pool optimization. After analysing the performance method with the browser, using the Call Tree, no explanation could be found exactly where there is this different. The biggest difference is in the exp method from the gfp12.ts file: from 17 seconds to 20 seconds. Inside this exp method, the mul and square methods from gfp12.ts have also a difference in time which is from 7.4 seconds to 9 seconds for the square and 6.8 seconds to 8.2 seconds for the mul. Going deeper in square method in the Call Tree as shown on the Figure 3, there is a loss of time not explicit in the mul function in gfp2.ts file. One hypothesis would be the management of the pool which makes the program run slower. However, the supervisor said that the deePool package was simple and it was very unlikely that this loss would come from there. Another hypothesis would be that JavaScript runtime environment is managing the memory efficiently which makes the use of a memory pool another layer to manage. This might cause a loss of time. Even with many more objects as the Modulus optimization, the program run faster which could explain why there is this loss in this optimization.



```
   3.3 ms  0.0 %  36267.6 ms  65.3 %   ▼ verify                                  index.ts:22
   0.3 ms  0.0 %  35188.2 ms  63.4 %     ▼ pair                                  point.ts:141
   4.1 ms  0.0 %  35187.7 ms  63.4 %       ▼ pair                                bn.ts:539
  22.2 ms  0.0 %  35182.2 ms  63.4 %         ▼ optimalAte                        opt-ate.ts:176
  56.3 ms  0.1 %  19394.1 ms  34.9 %           ▼ finalExponentiation             opt-ate.ts:141
 164.2 ms  0.3 %  17030.9 ms  30.7 %             ▼ exp                           gfp12.ts:176
  64.4 ms  0.1 %   7419.9 ms  13.4 %               ▼ square                      gfp12.ts:196
 310.2 ms  0.6 %   6096.9 ms  11.0 %                 ▼ mul                       gfp6.ts:153
 205.3 ms  0.4 %   3770.9 ms   6.8 %                   ▼ mul                     gfp2.ts:122
2891.5 ms  5.2 %   2953.1 ms   5.3 %                     ▶ mul                   gfp.ts:95
 407.9 ms  0.7 %    416.4 ms   0.8 %                     ▶ sub
 166.2 ms  0.3 %    170.9 ms   0.3 %                     ▶ add
  18.6 ms  0.0 %     18.6 ms   0.0 %                       GfP2                  gfp2.ts:21
   6.5 ms  0.0 %      6.5 ms   0.0 %                       Minor GC
 111.7 ms  0.2 %    846.8 ms   1.5 %                   ▶ sub
 151.8 ms  0.3 %    681.0 ms   1.2 %                   ▶ add
  47.6 ms  0.1 %    481.6 ms   0.9 %                   ▶ mulXi                   gfp2.ts:159
   6.2 ms  0.0 %      6.2 ms   0.0 %                     Minor GC
   0.2 ms  0.0 %      0.3 ms   0.0 %                   ▶ GfP6                    gfp6.ts:21
  33.0 ms  0.1 %    484.4 ms   0.9 %                 ▶ sub                       gfp6.ts:134
  63.4 ms  0.1 %    437.1 ms   0.8 %                 ▶ add                       gfp6.ts:120
  41.0 ms  0.1 %    295.9 ms   0.5 %                 ▶ mulTau                    gfp6.ts:214
  11.6 ms  0.0 %     12.4 ms   0.0 %                 ▶ add                       gfp2.ts:96
   8.6 ms  0.0 %      9.3 ms   0.0 %                 ▶ sub                       gfp2.ts:109
   9.2 ms  0.0 %      9.2 ms   0.0 %                   mulXi                     gfp2.ts:159
   9.0 ms  0.0 %      9.0 ms   0.0 %                   mul                       gfp2.ts:122
   1.2 ms  0.0 %      1.2 ms   0.0 %                   Minor GC
   0.2 ms  0.0 %      0.2 ms   0.0 %                   GfP12                     gfp12.ts:22
```

```
   4.7 ms  0.0 %  41919.3 ms  67.5 %   ▼ verify                                  index.ts:29
   2.0 ms  0.0 %  40809.1 ms  65.7 %     ▼ pair                                  point.ts:151
   5.1 ms  0.0 %  40806.9 ms  65.7 %       ▼ pair                                bn.ts:576
  21.3 ms  0.0 %  40801.1 ms  65.7 %         ▼ optimalAte                        opt-ate.ts:274
  48.7 ms  0.1 %  22845.0 ms  36.8 %           ▼ finalExponentiation             opt-ate.ts:219
 194.0 ms  0.3 %  20138.0 ms  32.4 %             ▼ exp                           gfp12.ts:227
 141.2 ms  0.2 %   8994.1 ms  14.5 %               ▼ square                      gfp12.ts:252
 778.1 ms  1.3 %   7398.8 ms  11.9 %                 ▼ mul                       gfp6.ts:202
 573.1 ms  0.9 %   4256.3 ms   6.9 %                   ▼ mul
3000.0 ms  4.8 %   3046.4 ms   4.9 %                     ▶ mul
 414.4 ms  0.7 %    424.3 ms   0.7 %                     ▶ sub
 185.7 ms  0.3 %    190.2 ms   0.3 %                     ▶ add
   9.9 ms  0.0 %      9.9 ms   0.0 %                       use                   deePool.src.js:38
   6.2 ms  0.0 %      6.2 ms   0.0 %                       Minor GC
   6.0 ms  0.0 %      6.0 ms   0.0 %                       recycle               deePool.src.js:48
 147.1 ms  0.2 %    872.1 ms   1.4 %                   ▶ sub
 235.6 ms  0.4 %    771.9 ms   1.2 %                   ▶ add                     gfp2.ts:136
 115.5 ms  0.2 %    659.1 ms   1.1 %                   ▶ mulXi                   gfp2.ts:208
  31.9 ms  0.1 %     31.9 ms   0.1 %                     use                     deePool.src.js:38
   4.3 ms  0.0 %     19.3 ms   0.0 %                   ▶ copy
  10.0 ms  0.0 %     10.0 ms   0.0 %                     Minor GC
   0.2 ms  0.0 %      0.2 ms   0.0 %                     recycle                 deePool.src.js:48
```

Figure 3: On the left, the call tree for the Modulus optimization. On the right, the call tree for the Pool optimization.

The grah on Figure 4 represents the average values for one key verification. When running the benchmark, it computes 2, 10, 100, 500 and 1000 key signatures and verifications. The different optimizations can be compared. The best speedup can be seen from Master to BigInt. The values are constant from 2 to 1000 key verifications but since it is an average, the maximum values are having bigger weights with less key verifications. This is why all values obtained are compared in the graph from the Figure 5 which shows the timing values for each key verification when 100 key verifications are done. It shows that the first key verification takes significantly more time. Then it decreases until reaching the average value. At the beginning of a program in JavaScript, it ignores all imports until they are needed. Hence, when the first verification starts, all these imports have to be compiled which makes the first one slower than following ones. Browsers usually uses Just-In-Time compilation which compiles JavaScript to executable bytecode just as it is about to run[19]. This shows why the average values is going down in Figure 4 from 2 key verifications to 1000 key verifications.

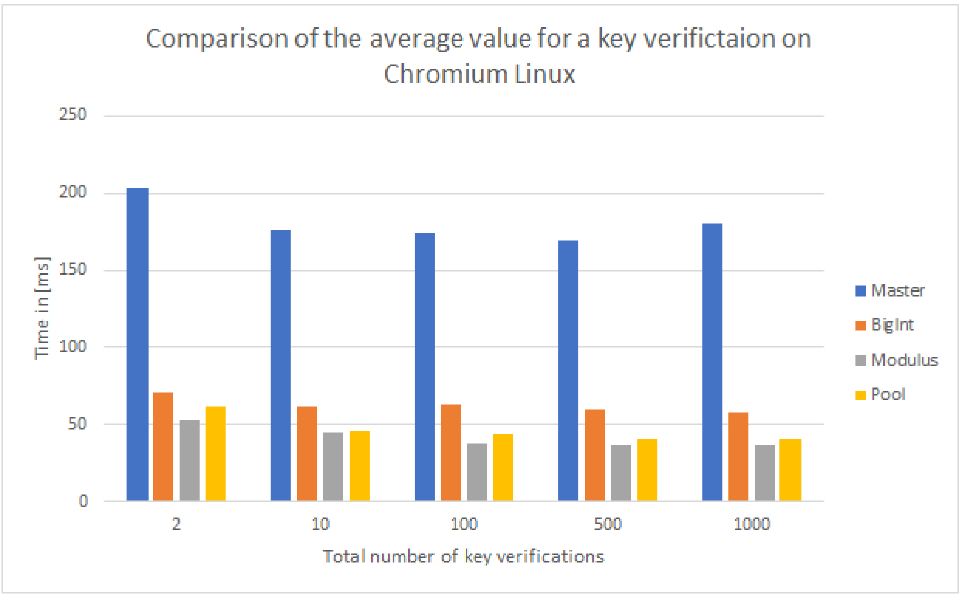


Figure 4: Graph comparing the average results for the key verifications.

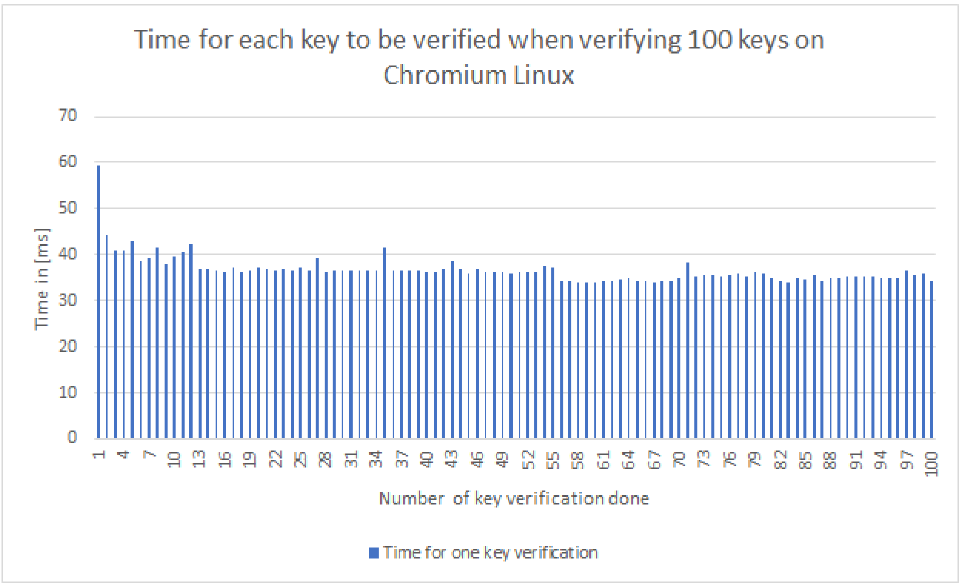


Figure 5: Graph comparing the different timing values when verifying 100 keys.

[19]`https://web.stanford.edu/class/cs98si/slides/overview.html`

4.2 Browsers comparison

A quick comparison on different browsers was asked for this project. The same program ran, on the same condition on different browsers and operating systems (Linux and Windows). For NodeJS, the program was tested on Linux. Both operating systems are on the same computer. The comparison graph can be found in the Figure 6. One remark about the browsers is that Opera, Edge and Chrome are using the same engine which is V8. This might explain why their results are similar to each other. Using Firefox or Chrome, the results are also similar on Linux and Windows. The results between Chrome and Firefox are contrasting. The difference between these two browsers is the engine used: Chrome is using the V8 JavaScript engine, whereas Firefox is using SpiderMonkey JavaScript engine. An hypothesis about this difference in the results could be that Chrome started earlier to implement the BigInt type in their browser[20] hence, Chrome's engine is handling in a better way the BigInt type. In this graph, NodeJs is also compared to the browsers. JavaScript is a programming language used for the browsers and for NodeJS. NodeJS is JavaScript runtime environment that executes JavaScript code outside of a browser. It has an event-driven architecture capable of asynchronous I/O. Similar to Chrome, it uses the V8 JavaScript engine but also combines an event loop, a low-level I/O API and a file system I/O[21].

The fact that NodeJS is slower than Chrome is expected since there are more abstraction level but a difference of around 130ms was not expected. This can be explained by the event loop which might slow down NodeJS and might even create small interruptions. The goal of the project was to improve the program for the browsers and nothing was done for NodeJS. To improve Kyber for NodeJS could be another project.

5 Conclusion

With this project, it was shown that the actual kyber implementation in JavaScript can be improved. The obtained speedup is around 4.76 faster than the Master. The supervisor said when 50ms key verification was reached that it would be impossible to be less than 30ms. In Chrome Windows, the best value is 28.6ms! It was disappointing to realise that the pool optimization did not improve the project. It would be recommended to test with another pool package or even to implement a pool memory management to be sure that it was not the package deePool that slowed down the project. Due to time constraints, it could not be tried.

A different optimization could have been done: the usage of two 53 bits

[20]`https://v8.dev/features/bigint#polyfilling-transpiling`

[21]`https://en.wikipedia.org/wiki/Node.js`

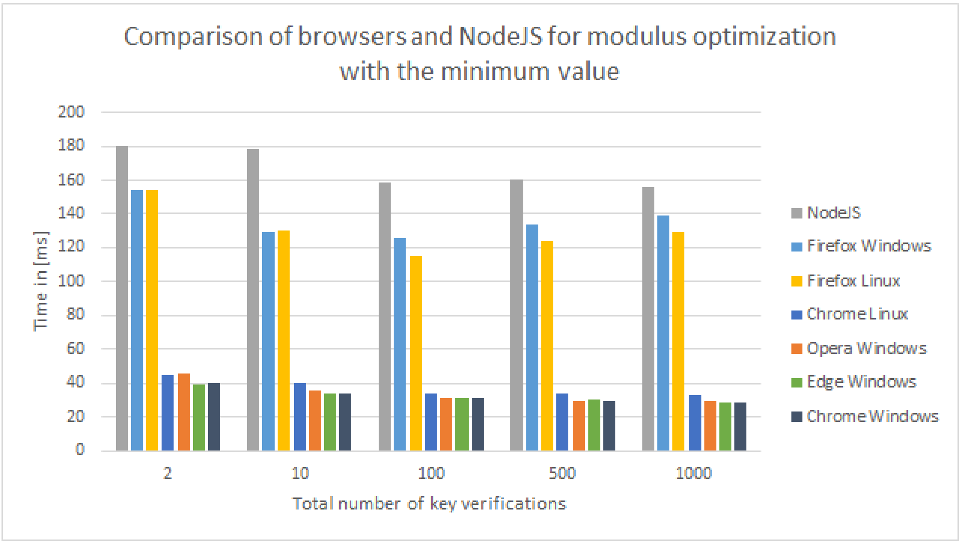


Figure 6: Graph comparing the minimum time of each browser for the key verifications.

integers concatenated instead of the BigInt type. But the improvement with the BigInt was so performing that it was decided not to try this optimization and move on to other optimizations.

Another improvement that can be easily done in other language but harder in JavaScript is the parallelization of the pairing in the function verify from the package sign/bls. This could have a speedup close to 2 since the pairing is the most expensive part of the verification function. Unfortunately, it is not the easiest language to have parallelism.

With the project, it is hoped that the link verifications will be turned on on the SkipChain Explorer. I really enjoyed this project as it taught me many different aspects of the optimization of a program. I am thankful to Gaylor Bosson which was my supervisor, he was patient and took the time for all questions I had during the project and to the DEDIS Lab which accepted me for this project. The repository can be found on the link[22].

[22]`https://github.com/dedis/student_19_julien/`

References

[1] https://github.com/dedis/kyber

[2] https://github.com/dedis/cothority

[3] https://www.epfl.ch/labs/dedis/

[4] http://status.dedis.ch/#/

[5] https://www.npmjs.com/package/bn.js

[6] https://www.stefanjudis.com/today-i-learned/measuring-execution-time-more-precisely-in-the-browser-and-node-js/

[7] https://developer.mozilla.org/fr/docs/Web/API/Performance

[8] https://v8.dev/features/bigint

[9] https://github.com/tc39/proposal-bigint

[10] https://en.wikipedia.org/wiki/ECMAScript

[11] https://github.com/tc39/proposals/blob/master/finished-proposals.md

[12] https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/BigInt#Usage_recommendations

[13] https://www.npmjs.com/package/bigint-buffer

[14] https://www.npmjs.com/package/deepool

[15] https://github.com/lpcsmath/egcd/blob/master/JavaScript/egcd.js

[16] https://helloacm.com/compute-powermod-abn/

[17] https://webpack.js.org/guides/production/

[18] https://web.stanford.edu/class/cs98si/slides/overview.html

[19] https://en.wikipedia.org/wiki/Node.js

[20] https://github.com/dedis/student_19_julien/