Improving the Performance of Cryptographic Voting Protocols

Rolf Haenni[1], Philipp Locher[1], and Nicolas Gailly[2]

[1] Bern University of Applied Sciences, CH-2501 Biel, Switzerland
{rolf.haenni,philipp.locher}@bfh.ch
[2] École Polytechnique Fédérale de Lausanne, CH-1015 Lausanne, Switzerland
nicolas.gailly@epfl.ch

Abstract. Cryptographic voting protocols often rely on methods that require a large number of modular exponentiations. Corresponding performance bottlenecks may appear both on the server and the client side. Applying existing optimization techniques is often mentioned and recommended in the literature, but their potential has never been analyzed in depth. In this paper, we investigate existing algorithms for computing fixed-base exponentiations and product exponentiations. Both of them appear frequently in voting protocols. We also explore the potential of applying small-exponent techniques. It turns out that using these techniques in combination, the overall computation time can be reduced by two or more orders of magnitude.

1 Introduction

Parties involved in cryptographic protocols often need to calculate a large number of modular exponentiations $z = b^e \bmod p$ (modexp) with large numbers b, e, and p.[3] With regard to performance, other computational tasks are often negligible. This is why optimizing modexp computations is the most promising option for improving the overall performance of an online voting system. Often, particular attention must be given to the client side, especially if it is implemented as a web application in JavaScript, which is known for its limited performance relative to native-code applications. Clearly, computational bottlenecks on the client may lead to critical usability problems and should therefore be avoided.

1.1 Problem Description and Context

In this paper, we consider the common setup of an ElGamal encryption scheme, which is often used for encrypting votes in cryptographic voting protocols. Let p denote a safe prime and $\mathbb{Z}_p^* = \{1, \ldots, p-1\}$ the corresponding multiplicative group of integers modulo p. This group has a sub-group $\mathbb{G}_q \subset \mathbb{Z}_p^*$ of prime order

[3] Exponentiations in groups such as elliptic curves, where the potential of applying the same type of optimizations is exactly the same, are less frequently used in voting protocols. Here we focus on multiplicative groups of integers modulo p, but our theoretical results are all applicable to the general case.

$q = \frac{p-1}{2}$, for which the decisional Diffie-Hellman (DDH) problem is believed to be hard. Since q is prime, all elements of $\mathbb{G}_q \setminus \{1\}$ are generators of $\mathbb{G}_q$. For such a generator $b \in \mathbb{G}_q$ and an exponent $e \in \mathbb{Z}_q$, computing the modular exponentiation $z = \mathsf{Exp}(b, e, p) = b^e \bmod p \in \mathbb{G}_q$ is the basic computational task considered in this paper. According to current recommendations [3], we have to deal with numbers of following bit lengths:

$$2048 \leq |p|, \quad 2 \leq |b| \leq |p|, \quad 112 \leq |e| \leq |p| - 1, \quad |z| = |p|.$$

In our theoretical analysis of different modexp algorithms, we will see that $\ell = |e|$ is one of the main parameters that determines the running time. If ℓ is equal or close to the above lower bound, we call e a *short exponent*.[4] Similarly, b is a *short base*, if $|b|$ is equal or close to 2. In all cases, we assume that b and e are drawn from a random uniform distribution.

The computational task considered in this paper consists of N different modexp instances for the same modulo p. Therefore, let $\boldsymbol{b} = (b_1, \ldots, b_N) \in \mathbb{G}_q^N$ and $\boldsymbol{e} = (e_1, \ldots, e_N) \in \mathbb{Z}_q^N$ denote the given vectors of values b_i and e_i. In the most general case, we need to compute values $z_i = \mathsf{Exp}(b_i, e_i, p)$ independently. The corresponding *multiple exponentiation problem* is denoted by

$$\boldsymbol{z} = \mathsf{MultiExp}(\boldsymbol{b}, \boldsymbol{e}, p) \in \mathbb{G}_q^N,$$

where $\boldsymbol{z} = (z_1, \ldots, z_N)$ is the vector of output values. We use $\mathsf{MultiExp}$ as reference point for judging the benefits of optimization algorithms, which can be applied to the following two special cases:

- *Product Exponentiation.* Compute the product $z = \prod_{i=1}^{N} z_i \bmod p$ of values $z_i = \mathsf{Exp}(b_i, e_i, p)$ for inputs $\boldsymbol{b} = (b_1, \ldots, b_N)$ and $\boldsymbol{e} = (e_1, \ldots, e_N)$:
$$z = \mathsf{ProductExp}(\boldsymbol{b}, \boldsymbol{e}, p) \in \mathbb{G}_q.$$
- *Fixed-Base Exponentiation.* Compute $\boldsymbol{z} = (z_1, \ldots, z_N)$ of values $z_i = \mathsf{Exp}(b, e_i, p)$ for inputs $\boldsymbol{b} = (b, \ldots, b)$ and $\boldsymbol{e} = (e_1, \ldots, e_N)$:
$$\boldsymbol{z} = \mathsf{FixedBaseExp}(b, \boldsymbol{e}, p) \in \mathbb{G}_q^N.$$

A similar special case arises for $\boldsymbol{e} = (e, \ldots, e)$. However, since the benefits of algorithms for solving such *fixed-exponent exponentiation* problems are rather limited (see [10, Section 14.6.2]), we do not consider them in this paper.

[4] There are multiple reasons for working with short exponents. In certain applications of some cryptographic schemes, a much smaller subgroup $\mathbb{G}_q \subset \mathbb{Z}_p^*$ is sufficient. To resist against the best available DL algorithms, the minimal bit length of q in such cases is 2λ, where λ denotes the security strength, for example $|q| = 224$ for $\lambda = 112$. Corresponding exponents $e \in \mathbb{Z}_q$ are then inherently restricted to $|q|$ bits. In larger groups, smaller exponents are sometimes selected on purpose, for example in the case of a challenge $c \in \mathbb{Z}_{2^\lambda}$ in a zero-knowledge proof or in systems relying on the *short-exponent discrete logarithm* (DLSE) assumption, in which short exponents $e \in \mathbb{Z}_{2^{2\lambda}}$ deliver the same provable security under a slightly stronger intractability assumption. For example, using the ElGamal encryption scheme with short randomizations has been proven IND-CPA secure under the DLSE assumption [7].

1.2 Contribution and Paper Overview

The goal of this paper is to increase public awareness of the potential performance benefits that results from applying the most appropriate modexp algorithms to a particular given computational task in a cryptographic voting protocol. The state-of-the-art algorithms for solving MultiExp, FixedBaseExp, and ProductExp most efficiently are presented in Section 2. We summarize the algorithmic and theoretical background of the available methods and provide an analysis of the expected computational costs. Since all algorithms are parameterized, we give instructions for finding optimal algorithm parameters in a given cryptographic setup. The maximal performance benefits when running the algorithms with optimal parameters are analyzed for FixedBaseExp and ProductExp. For some of the presented algorithms, we are not aware of any references in the literature. One of the presented algorithm turns out to be equivalent to the well-known comb method [9], but we believe that our description is more intuitive.

A more practical perspective of this topic is given in Section 3. The target audience of this section are practitioners and developers of online voting systems designed for real-world elections. We performed different performance tests for various algorithms and measured their effective running times on different platforms. A special focus is given to the client side, in which performance bottlenecks are more likely to appear.

2 Performance Analysis of Exponentiation Algorithms

Most programming languages or mathematical libraries providing large number arithmetic have a built-in support for modexp computations. They usually implement general-purpose modexp algorithms from [10], for example Alg. 14.82, Alg. 14.83, or Alg. 14.85, which we will later call HAC 14.82, HAC 14.83, and HAC 14.85. Using such algorithms, the average time for solving MultiExp is exactly N times the average time for solving Exp. General algorithms for Exp are discussed in Section 2.2. We will use them as reference points for evaluating the performance of several optimization algorithms. The results obtained for ProductExp and FixedBaseExp are discussed in Sections 2.3 and 2.4.

2.1 Measurement Methodology

In our theoretical analysis of an algorithm Alg for solving $\mathsf{Exp}(b, e, p)$, we count the number $M_{\mathsf{Alg}}(\ell)$ of multiplications needed for exponents of length $\ell = |e|$. For reasons of simplicity, we do not distinguish between squaring and multiplication operations, i.e., we assume that they are equally expensive, which may not necessarily be true in every library. In case of ProductExp and FixedBaseExp algorithms for ℓ-bit exponents, we count the total number $M_{\mathsf{Alg}}(\ell, N)$ of multiplications needed to solve the entire problem. To compare them with general-purpose algorithms, we compute the *average number of multiplications per modexp*,

$$\widetilde{M}_{\mathsf{Alg}}(\ell, N) = \frac{M_{\mathsf{Alg}}(\ell, N)}{N},$$

and call it *relative (theoretical) running time* of Alg. Both $M_{\mathsf{Alg}}(\ell)$ and $\widetilde{M}_{\mathsf{Alg}}(\ell, N)$ may depend on algorithm parameters $\kappa_1, \ldots, \kappa_r$. We either include them explicitly in our notation as $M_{\mathsf{Alg}}^{\kappa_1,\ldots,\kappa_r}(\ell)$ and $\widetilde{M}_{\mathsf{Alg}}^{\kappa_1,\ldots,\kappa_r}(\ell, N)$, or we skip them to indicate that optimal parameters κ_i^{opt} have been chosen:

$$M_{\mathsf{Alg}}^{\mathsf{opt}}(\ell) := M_{\mathsf{Alg}}^{\kappa_1^{\mathsf{opt}},\ldots,\kappa_r^{\mathsf{opt}}}(\ell) \quad \text{and} \quad \widetilde{M}_{\mathsf{Alg}}^{\mathsf{opt}}(\ell, N) := \widetilde{M}_{\mathsf{Alg}}^{\kappa_1^{\mathsf{opt}},\ldots,\kappa_r^{\mathsf{opt}}}(\ell, N).$$

If we take $M_{\mathsf{Alg}^*}^{\mathsf{opt}}(\ell)$ of a general-purpose modexp algorithm Alg^* as reference point for evaluating the performance of an optimization algorithm Alg, both of them instantiated with optimal algorithm parameters, we can measure the benefit of Alg relative to Alg^* by computing the fraction

$$\mu_{\mathsf{Alg}}(\ell, N) = M_{\mathsf{Alg}^*}^{\mathsf{opt}}(\ell) \,/\, \widetilde{M}_{\mathsf{Alg}}^{\mathsf{opt}}(\ell, N).$$

This number will be called *(theoretical) impact factor* of algorithm Alg in problem instances of size N and exponents of size ℓ. To measure the benefit of combining an optimization algorithm Alg with short-exponent techniques, for example in a setting with $\ell_{\mathsf{long}} = 2047$ and $\ell_{\mathsf{short}} = 224$, we compute the fraction

$$\mu_{\mathsf{Alg}}^*(\ell_{\mathsf{long}}, \ell_{\mathsf{short}}, N) = M_{\mathsf{Alg}^*}^{\mathsf{opt}}(\ell_{\mathsf{long}}) \,/\, \widetilde{M}_{\mathsf{Alg}}^{\mathsf{opt}}(\ell_{\mathsf{short}}, N).$$

All modexp algorithms described in this section can benefit more or less equally from techniques known as *Montgomery* or *Barrett reduction* as described as Alg. 14.32 and Alg. 14.42 in [10]. We do not study their potential in this paper.

2.2 General-Purpose Exponentiation Algorithms

The most fundamental problem considered in this paper is the computation of a single value $z = \mathsf{Exp}(b, e, p)$. A widely implemented algorithm is the *window method* as described in HAC 14.82, in which the ℓ-bits exponent is written as $e = (e_{t-1} \cdots e_1 e_0)_B$ in base $B = 2^k$. The parameter k is called *window size* and $t = \lceil \frac{\ell}{k} \rceil$ denotes the number of windows $e_i \in \mathbb{Z}_{2^k}$. The algorithm processes the bits of each window en bloc by decomposing b^e using Horner's method:

$$b^e = b^{\sum_{i=0}^{t-1} e_i B^i} = b^{e_0}(b^{e_1}(b^{e_2} \cdots (b^{e_{t-2}}(b^{e_{t-1}})^B)^B \cdots)^B)^B.$$

This expression can be evaluated from inside to outside starting with the leftmost window e_{t-1}. The resulting iteration corresponds to HAC 14.82. If all values b^{e_i} have been precomputed and stored in a table, then the iteration requires $k(t-1)$ squarings and $t - 1 = \lfloor \frac{\ell-1}{k} \rfloor$ multiplications. The precomputation table may contain up to 2^k entries, which can be computed using $2^k - 2$ multiplications.

To reduce the size of this table and therefore to improve the overall computation time, consider the decomposition of each $e_i \neq 0$ into $e_i = u_i 2^{v_i}$ such that u_i is odd (e_{t-1} remains untouched). For $e_i = 0$, let $u_i = v_i = 0$. This leads to

$$b^e = (b^{u_0}((b^{u_1}((b^{u_2} \cdots ((b^{u_{t-2}}(b^{e_{t-1}})^{2^{k-v_{t-2}}})^{2^{v_{t-2}}} \cdots)^{2^{k-v_1}})^{2^{v_1}})^{2^{k-v_0}})^{2^{v_0}},$$

from which the improved window algorithm HAC 14.83 follows. Here, the precomputation table of all possible odd values $u_i \in \mathbb{Z}_{2^k}$ contains at most 2^{k-1} entries, which can be generated using the same amount of multiplications.

To compute the running time of HAC 14.83 as precisely as possible, we have to take into account that e may contain some windows $e_i = 0$. We assume that such cases, in which one multiplication is saved, appear with probability $P_k = \frac{2^k-1}{2^k} \in [0.5, 1)$. This leads to

$$M^k_{\mathsf{HAC\,14.83}}(\ell) = 2^{k-1} + (k + P_k)\left\lfloor \frac{\ell - 1}{k} \right\rfloor + 1 \; < \; 2^{k-1} + \ell \cdot \frac{k+1}{k},$$

which we will later use as reference point for evaluating the performance of several optimization algorithms.[5]

The remaining question regarding the window method is the selection of the optimal parameter k^{opt} that minimizes $M_{\mathsf{HAC\,14.83}}(k, \ell)$. To get the desired value for a given ℓ, we can either solve $\frac{\mathrm{d}}{\mathrm{d}k}[M^k_{\mathsf{HAC\,14.83}}(\ell)] = 0$ numerically, for example using Newton's method, or perform an exhaustive search over $1 \le k \le \ell$. The results for $80 \le \ell \le 15360$ are summarized in Table 1. The mapping from ℓ to k^{opt} is unique within the ranges given in Table 1, but not in the areas between these ranges, where k^{opt} jumps forth an back between two adjacent values.

ℓ	82–184	217–545	566–1434	1465–3759	3802–9368	>9425
k^{opt}	4	5	6	7	8	9

Table 1: Optimal window sizes k^{opt} in HAC 14.83 for different exponent lengths ℓ.

An even better performance offers the *sliding window method* as implemented in HAC 14.85, in which one multiplication can be saved in the average after processing $\frac{k}{2}$ windows, i.e., $\frac{2t}{k}$ multiplications can be saved in total. Since this is a non-negligible quantity, HAC 14.85 is the recommended method in [10]. Nevertheless, HAC 14.82 and HAC 14.83 (and even HAC 14.79) are implemented in some arithmetic libraries for large integers.

As a numerical example, consider the common cryptographic setting with $|p| = 2048$ and exponents of length $\ell = |e| = 2047$. In this case, we select $k^{\mathsf{opt}} = 7$ for getting the best possible running time $M^{\mathsf{opt}}_{\mathsf{HAC\,14.83}}(2047) = 2401$. This is about 22% faster than standard square-and-multiply, which corresponds to $M^1_{\mathsf{HAC\,14.83}}(2047) = 3071$ for windows of size $k = 1$. Using the sliding window method, the performance improves by another 3% to $M^{\mathsf{opt}}_{\mathsf{HAC\,14.85}}(2047) = 2318$.

2.3 Algorithms for Product Exponentiations

Product exponentiation problems $\mathsf{ProductExp}(\boldsymbol{b}, \boldsymbol{e}, p)$ can be solved in a naïve way by computing the product $z = \prod_{i=1}^{N} z_i \bmod p$ of the results z_i obtained from calling an algorithm from the previous section separately for all N pairs (b_i, e_i). As we will see in this section, this is far from being an optimal solution.

[5] The precomputation of HAC 14.82, HAC 14.83, and HAC 14.85 gets much faster for a small base. For values such as $b = 2$ or $b = 4$, multiplication during precomputation corresponds to shifting the bits a few positions to the left (modulo p), which is obviously much faster than regular multiplications. In such a case, our theoretical analysis based on counting modular multiplications gets inaccurate.

A special-purpose algorithm for this problem is HAC 14.88, but it only performs well for small N. The reasons for this is the size of the precomputation table, which grows exponentially with N. The total relative running time is as follows:

$$\widetilde{M}_{\mathsf{HAC\,14.88}}(\ell, N) = \frac{(2^N - N - 1) + (\ell - 1)(1 + P_N)}{N} < \frac{2^N + 2\ell}{N}.$$

If ℓ is fixed in this expression, we can derive the problem size N for which the algorithm performs best. For $\ell = 2047$, the best relative running time is $\widetilde{M}_{\mathsf{HAC\,14.88}}(2047, 9) = 510$, which we get for $N = 9$. For this particular case, the algorithm performs almost five times better than HAC 14.83, but it quickly starts to perform (much) worse when N gets larger. In the light of these considerations, applying HAC 14.88 directly for solving ProductExp is only possible for very small problem instances. However, it can be used as a building block for algorithms that perform well in general. The most obvious way is to split the full task into $s = \lfloor \frac{N}{m} \rfloor$ sub-tasks of size m and one sub-task of size $r = N \bmod m$.

To formalize this idea, let $I_j = \{jm + 1, \ldots, jm + m\}$ be the indices of sub-task $0 \leq j \leq s - 1$ and $I_s = \{sm + 1, \ldots, N\}$ the set of indices of sub-task s. The problem can then be decomposed into

$$z = \prod_{i=1}^{N} a_i^{e_i} \bmod p = \prod_{j=0}^{s} \prod_{i \in I_j} a_i^{e_i} \bmod p = \prod_{j=0}^{s} \mathsf{ProductExp}(\boldsymbol{b}_j, \boldsymbol{e}_j, p),$$

where $\boldsymbol{b}_j$ and $\boldsymbol{e}_j$ denote corresponding sub-vectors from $\boldsymbol{b} = \boldsymbol{b}_0 || \cdots || \boldsymbol{b}_s$ and $\boldsymbol{e} = \boldsymbol{e}_0 || \cdots || \boldsymbol{e}_s$. The relative running time of the resulting Alg.1 is as follows:

$$\widetilde{M}^{m}_{\mathsf{Alg.1}}(\ell, N) = \frac{sm \cdot \widetilde{M}_{\mathsf{HAC\,14.88}}(\ell, m) + r \cdot \widetilde{M}_{\mathsf{HAC\,14.88}}(\ell, r) + s}{N} \approx \widetilde{M}_{\mathsf{HAC\,14.88}}(\ell, m).$$

It follows that this algorithm performs best by selecting the parameter m according to the above discussion of the optimal value N in HAC 14.88. As an example, we select $m^{\mathsf{opt}} = 9$ for $\ell = 2047$, which leads to $\widetilde{M}^{\mathsf{opt}}_{\mathsf{Alg.1}}(2047, N) = 510$ and $\mu_{\mathsf{Alg.1}}(2047, N) = 4.7$ for large input sizes N. For a setting with $\ell_{\mathsf{long}} = 2047$ and $\ell_{\mathsf{short}} = 224$, we get $m^{\mathsf{opt}} = 7$ and $\mu^{*}_{\mathsf{Alg.1}}(2047, 224, N) = 29.99$.

The benefit of Alg.1 is already appealing, but it can be improved even further. For this, we need to drill a hole into HAC 14.88, by placing the squaring operation in Step 3 outside the brackets over all sub-tasks of Alg.1. In Alg.2, we assume that N is a multiple of m, which we can always achieve by filling up the inputs with $m - r$ additional values $b_i = 1$ and $e_i = 0$. Furthermore, let $\boldsymbol{E}_j \in \{0, 1\}^{m \times \ell}$ be the binary *exponent array* of $\boldsymbol{e}_j$, whose rows are the binary representations of the exponents from $\boldsymbol{e}_j$ [10]. Let $\boldsymbol{E}_j[l]$ denote the l-th column of $\boldsymbol{E}_j$.

In the resulting Alg.2, which is a synthesis of Alg.1 and HAC 14.88, we initially perform the precomputation for all s sub-tasks. In the main loop of the algorithm, we see that the loop over the sub-tasks only performs one multiplication in each step, but no squarings. This reduces the total number of squarings from $s \cdot (\ell - 1)$ to $\ell - 1$. Therefore, the relative running time of Alg.2 is strictly smaller than the

Algorithm: $\mathsf{ProductExp}_m(\boldsymbol{b}, \boldsymbol{e}, p)$
Input: Bases $\boldsymbol{b} = \boldsymbol{b}_0 || \cdots || \boldsymbol{b}_s$
Exponents $\boldsymbol{e} = \boldsymbol{e}_0 || \cdots || \boldsymbol{e}_s$
Modulus p
Sub-task size $1 \leq m \leq N$
$z \leftarrow 1$
for $j = 0, \ldots, s$ **do**
  $z_j \leftarrow \mathsf{HAC\ 14.88}(\boldsymbol{b}_j, \boldsymbol{e}_j, p)$
  $z \leftarrow z \cdot z_j \bmod p$
return z

Algorithm 1: Simple product exponentiation algorithm based on HAC 14.88.

Algorithm: $\mathsf{ProductExp}_m(\boldsymbol{b}, \boldsymbol{e}, p)$
Input: Bases $\boldsymbol{b} = \boldsymbol{b}_0 || \cdots || \boldsymbol{b}_{s-1}$
Exponents $\boldsymbol{e} = \boldsymbol{e}_0 || \cdots || \boldsymbol{e}_{s-1}$
Modulus p
Sub-task size $1 \leq m \leq N$
for $i = 0, \ldots, 2^m - 1$ **do**
  $(i_{m-1}, \ldots, i_0)_2 \leftarrow i$
  for $j = 0, \ldots, s-1$ **do**
    $(b_0, \ldots, b_{m-1}) \leftarrow \boldsymbol{b}_j$
    $B_{ij} \leftarrow \prod_{l=0}^{m-1} b_l^{i_l} \bmod p$
$z \leftarrow 1$
for $l = 0, \ldots, \ell - 1$ **do**
  $z \leftarrow z^2 \bmod p$
  for $j = 0, \ldots, s-1$ **do**
    $i \leftarrow \boldsymbol{E}_j[l]$
    $z \leftarrow z \cdot B_{ij} \bmod p$
return z

Algorithm 2: Improved product exponentiation algorithm based on HAC 14.88 and Alg.1.

relative running time of Alg.1:

$$\widetilde{M}^m_{\mathsf{Alg.2}}(\ell, N) = \frac{s \cdot (2^m - m - 1) + (\ell - 1) + (\ell s - 1) \cdot P_m}{N} < \frac{2^m + \ell}{m} + \frac{\ell}{N}.$$

To compare this result with the numbers from above, we select $m^{\mathsf{opt}} = 9$ for $\ell = 2047$ (see Table 2), which leads to $\widetilde{M}^{\mathsf{opt}}_{\mathsf{Alg.2}}(2047, N) = 282$ and $\mu_{\mathsf{Alg.2}}(2047, N) = 8.51$ for large input sizes N. For $\ell_{\mathsf{long}} = 2047$ and $\ell_{\mathsf{short}} = 224$, we select $m^{\mathsf{opt}} = 6$ to get $\mu^*_{\mathsf{Alg.2}}(2047, 224, N) = 52.15$. In both settings, Alg.2 is therefore approximately 45% more efficient than Alg.1. Note that we are not aware of any published document, in which Alg.2 is described and analyzed in this way.

To conclude our analysis of product exponentiation algorithms, we show in Table 2 the mapping from $80 \leq \ell \leq 15360$ to m^{opt} in a similar way as for k^{opt} in Table 1, i.e., with some fuzzy areas between the ranges of two adjacent values. It shows that for Alg.2 the optimal parameter m^{opt} is usually smaller than for Alg.1. This implies that Alg.2 benefits from smaller precomputation tables.

Alg.1	ℓ	80–168	180–397	415–914	939–2068	2101–4625	4666–10270	>10321
	m^{opt}	6	7	8	9	10	11	12

Alg.2	ℓ	80–147	174–349	380–802	845–1839	1896–4148	4231–9284	>9285
	m^{opt}	5	6	7	8	9	10	11

Table 2: Optimal sub-task size m^{opt} for different exponent lengths ℓ in Alg.1 and Alg.2.

2.4 Algorithms for Fixed-Base Exponentiations

Two of the most common and frequently cited fixed-base exponentiation algorithms are the *fixed-base windowing method* by Brickell et al. [2] and the *comb method* by Lim and Lee [8,9]. Brickell's method is strictly less efficient than the comb method, which itself can be seen as a generalization of the following idea. Let $\boldsymbol{z} = \mathsf{FixedBaseExp}(b, \boldsymbol{e}, p)$ be the problem instance to solve for given inputs b, $\boldsymbol{e} = (e_1, \ldots, e_N)$, and p. Similar to the window method of Section 2.2, we define a bit length $1 \le k \le \ell$, into which the exponents are decomposed. If we consider a single exponent e, we write it as $e = (e_{t-1} \cdots e_1 e_0)_B$ in base $B = 2^k$, where $t = \lceil \frac{\ell}{k} \rceil$ denotes the resulting number of sub-exponents e_i of length k.

This decomposition of the exponent allows us to transform the computation of b^e into a product exponentiation problem of size t, which can be solved using HAC 14.88, Alg.1, or Alg.2:

$$b^e = b^{\sum_{i=0}^{t-1} e_i B^i} = \prod_{i=0}^{t-1} b^{e_i B^i} = \prod_{i=0}^{t-1} (b^{B^i})^{e_i} = \prod_{i=0}^{t-1} (b^{2^{ik}})^{e_i}$$
$$= \mathsf{ProductExp}((b_0, \ldots, b_{t-1}), (e_0, \ldots, e_{t-1})), p), \text{ for } b_i = b^{2^{ik}}.$$

The crucial point to observe here is that $\boldsymbol{b} = (b_0, \ldots, b_{t-1})$, which only depends on b, will be the same for all N computations $z_i = b^{e_i}$ with base b. This implies that the precomputation of the ProductExp algorithm only needs to be conducted once for solving a full FixedBaseExp problem of size N.

To describe and analyze the algorithm resulting from this idea, let's assume that the selected ProductExp algorithm memoizes the precomputation tables from previous calls, for example by storing them in a dictionary. Therefore, whenever the same vector $\boldsymbol{b}$ is used more than once, the precomputation table is already available. Clearly, the performance of the resulting Alg.3 depends strongly on this assumption, because then the precomputation table can be amortized over the N modexps. In the same way, the values $\boldsymbol{b} = (b_0, \ldots, b_{t-1})$ precomputed in Alg.3 must stored for later use.

Let Alg.3.1 and Alg.3.2 denote the algorithms obtained from combining Alg.3 with HAC 14.88 and Alg.2, respectively. While Alg.3.1 is strictly inferior to Alg.3.2, it is the combination we found in some libraries (see Section 3.1). Note that we are not aware of any description of Alg.3.2 in this form in a published document, nor of any existing implementation. The relative running times of the algorithms, which depend on both ℓ and N, are as follows (using $E(x) = 2^x - x - 1$):

$$\widetilde{M}^{k}_{\mathsf{Alg.3.1}}(\ell, N) = \frac{E(t) + (t-1)k}{N} + (k-1)(1+P_t) < \frac{2^t + \ell}{N} + 2k,$$
$$\widetilde{M}^{k,m}_{\mathsf{Alg.3.2}}(\ell, N) = \frac{sE(m) + (t-1)k}{N} + (k-1) + (ks-1)P_m < \frac{s \cdot 2^m + \ell}{N} + ks + k.$$

Both versions of the algorithm have the same main parameter $1 \le k \le \ell$. As soon as k is fixed in Alg.3.2 and N is sufficiently large, we can select $1 \le m^{\mathsf{opt}} \le t$ deterministically from Table 2. The selection of k^{opt} for a given pair (ℓ, N) is therefore the main optimization problem to solve in both algorithms. We have

Algorithm: $\mathsf{FixedBaseExp}_{k,m}(b, \boldsymbol{e}, p)$
Input: Base b
Exponents $\boldsymbol{e} = (e_1, \ldots, e_N)$, $e_i = (e_{i,t-1} \cdots, e_{i,1} e_{i,0})_B$
Modulus p
Block size $1 \le k \le \ell$, $B = 2^k$
Sub-task size $1 \le m \le t$
for $i = 0, \ldots, t-1$ **do**
  $b_i \leftarrow b$
  if $i < t-1$ **then**
    for $j = 1, \ldots, k$ **do**
      $b \leftarrow b^2 \bmod p$
$\boldsymbol{b} = (b_0, \ldots, b_{t-1})$
for $i = 1, \ldots, N$ **do**
  $z_i \leftarrow \mathsf{ProductExp}_m(\boldsymbol{b}, (e_{i,0}, \ldots, e_{i,t-1}), p)$ // using HAC 14.88, Alg.1, or Alg.2
$\boldsymbol{z} \leftarrow (z_1, \ldots, z_N)$
return $\boldsymbol{z}$

Algorithm 3: Fixed-base exponentiation algorithm based on HAC 14.88, Alg.1, or Alg.2. In case of HAC 14.88, the parameter m is irrelevant.

computed optimal parameters for $\ell = 2047$ and problem sizes $1 \le N \le 10^7$. Figure 1 shows the resulting impact factors $\mu_{\mathsf{Alg}}(2047, N)$ and $\mu^*_{\mathsf{Alg}}(2047, 224, N)$.

The aforementioned comb method by Lim and Lee also has two parameters $1 \le b \le a \le \ell$. Here, we refer to its description in [10] as HAC 14.117. For $h = \lceil \ell/a \rceil$, $v = \lceil a/b \rceil$, its running time

$$\widetilde{M}^{a,b}_{\mathsf{HAC\ 14.117}}(\ell, N) = \frac{vE(h) + (h-1)a + (v-1)b}{N} + (b-1) + (bv-1)P_h$$

is exactly the running time of $\widetilde{M}^{k,m}_{\mathsf{Alg.3.2}}(\ell, N)$ for $a = ks$ and $b = k$ (and therefore $h = m$ and $v = s$). This implies that Alg.3.2 and HAC 14.117 are essentially the *same* algorithms. Note that by setting $a = b = k$ (or $v = s = 1$), they contain the strictly inferior Alg.3.1 as a special case.

2.5 Use Case: Cryptographic Shuffle

A particular use case for applying the algorithms presented in this paper is the shuffling of a list $E = \langle e_1, \ldots, e_n \rangle$ of ElGamal encrypted votes $e_i = (g^{r_i}, m_i p k^{r_i})$ in a verifiable re-encryption mix-net. This is one of the most time-consuming components in many voting protocols. Note that n is of the same order of magnitude as the size of the electorate, i.e., possibly several millions in a large election context. Two instances of FixedBaseExp are needed for re-encrypting the encrypted votes in a single mixing step.

The particular shuffle proof system by Wikström and Terelius [13,14] requires a total number of $8n+5$ modexps for generating the proof and $9n+11$ modexps for verifying the proof. In Table 3, we derived from [5] a more detailed overview of the necessary number of modexps of this particular approach. Note that some

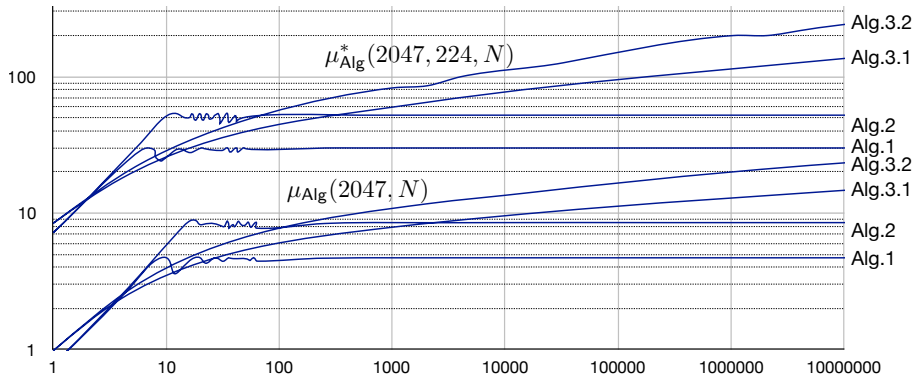

Fig. 1: Impact factors $\mu_{\mathsf{Alg}}(2047, N)$ and $\mu^*_{\mathsf{Alg}}(2047, 224, N)$ of different algorithms. The plotted curves show that ProductExp algorithms (Alg.1 and Alg.2) converge quickly to a constant speedup, whereas FixedBaseExp algorithms (Alg.3.1 and Alg.3.2) increase their speedup with increasing problem sizes. The curves also show that the benefit of short exponent-techniques multiplies the benefit of the optimization algorithms.

of the involved exponents play the role of a challenge in the proof protocol, i.e., their lengths are restricted to the security strength λ. Therefore, we make a distinction between exponents of length $\ell_{\mathsf{long}} = |q| = |p| - 1$ and $\ell_{\mathsf{short}} = \lambda$.

To evaluate the usage of Alg.2 and Alg.3.2 in a verifiable mix-net, we calculated relative running times (number of modular multiplications per encrypted vote) and impact factors (benefit relative to HAC 14.83) of the shuffle algorithms for $\ell_{\mathsf{long}} = 2047$, $\ell_{\mathsf{short}} = 112$, and $n \in \{10, 100, \ldots, 10^6\}$. Table 4 shows that all shuffle algorithms benefit considerably from optimized modexp algorithms. Shuffling itself is up to 20 times and generating the proof up to 13 times more efficient. The smallest benefit results for the proof verification, which is between 3–4 times more efficient. These numbers can be improved even further by applying short-exponent techniques to the ElGamal encryptions.

		Shuffle	**Generate Proof**	**Verify Proof**	
		ℓ_{long}	ℓ_{long}	ℓ_{long}	ℓ_{short}
Exp		–	–	$n + 7$	n
ProductExp		–	$3n$	$3n$	$3n$
FixedBaseExp	g	n	$4n + 4$	$n + 4$	–
	h	–	n	–	–
	pk	n	1	–	–
Total		$2n$	$8n + 5$	$5n + 11$	$4n$

Table 3: Number of exponentiations for shuffling n votes in a verifiable mix-net.

	$n = 10$		$n = 100$		$n = 1000$		$n = 10^4$		$n = 10^5$		$n = 10^6$	
Shuffle	1216	3.95	626	7.66	450	10.66	352	13.63	286	16.78	240	19.99
Generate proof	2563	7.08	1884	9.06	1623	10.44	1463	11.58	1359	12.46	1283	13.20
Verify proof	5132	2.96	3486	3.68	3306	3.81	3276	3.84	3265	3.85	3262	3.85

Table 4: Relative running times (1st column) and impact factors (2nd column) of different shuffle algorithms in a setting with $\ell_{\mathsf{long}} = 2047$ and $\ell_{\mathsf{short}} = 112$.

3 Experimental Results

To confirm the theoretical results from Section 2, we performed various tests on different platforms. Generally, client-side performance is more critical as neither the hardware nor the runtime environment can be influenced directly.

3.1 Technologies

On both the client and the server side, we focused on testing popular open-source libraries that implement the algorithms analyzed in Section 2. On the server side, the choice is limited as GMP can be regarded as a de facto standard for multiple precision arithmetic. On the other hand, there are a number of potential JavaScript libraries available to be used in browser applications.

Server. The *GNU Multiple Precision Arithmetic Library* (GMP) is a C library that aims to be the fastest arbitrary-precision arithmetic library [4]. The most critical inner loops are written in optimized assembly code, specialized for different processors. There exist wrappers to many other programming language, such as C++, Java, or Python, increasing the scope of GMP remarkably. Modular exponentiation is implemented based on the sliding-window method (HAC 14.85) with Montgomery reduction. Unfortunately, GMP does not offer algorithms for fixed-base or product exponentiations. However, the *GMP Modular Exponentiation Extension* (MEE) by Douglas Wikström offers them both [16].

Client. Below we list the JavaScript libraries considered in our analysis. Their particularities relative to modular exponentiation is summarized in Table 5.

- **JSBN** is a lightweight implementation of large number arithmetic mainly developed by Tom Wu between 2009 and 2013 [18]. A few minor bugs have been fixed in recent years, but otherwise the project seems to be inactive today [12]. Modexp computations are based on the sliding-window technique in combination with Barret and Montgomery reductions.
- **Leemon** is another lightweight implementation of large number arithmetic developed by Leemon C. Baird between 2000 and 2013. Bug fixes to the code available on GitHub have been made until 2016 [1]. Modexps are computed with the square-and-multiply algorithm and Montgomery reduction.
- **VJSC** is a cryptographic library especially tailored for application in electronic voting protocols developed by Douglas Wikström [15,17]. The library

is available on GitHub since February 2018. Modexp computations are performed by the improved window method. As a unique feature, VJSC offers integrated support for product and fixed-base exponentiation.
– **MiniGMP** provides a subset of the features of the GMP library [11]. Since it consists of pure C code, i.e., without any assembly optimization, it can be compiled into the WebAssembly format and used for web applications. Modexp computations are based on the square-and-multiply method.

Independently of the actual modexp performance, VJSC and MiniGMP are currently the best maintained libraries. Both of them are well tested and documented. The disadvantage of using MiniGMP in a web application is its dependency to the WebAssembly technology, which has been introduced only recently. In addition to the libraries listed above, there is also the *bn.js* JavaScript library for big numbers [6]. Its main target are elliptic curves and hence it is optimized for calculations with 256-bit numbers. For example, the window size within the modular exponentiation algorithm is hard-coded to $k = 4$.

Library	**JSBN**	**Leemon**	**VJSC**	**MiniGMP**	**GMP/MEE**
Language	JavaScript	JavaScript	JavaScript	C	C
Author(s)	T. Wu	L. C. Baird	D. Wikström	N. Möller	T. Granlund D. Wikström
Exp	HAC 14.85*	HAC 14.79*	HAC 14.83	HAC 14.79	HAC 14.85*
ProductExp	*unsupported*	*unsupported*	HAC 14.88	*unsupported*	Alg.2
FixedBaseExp	*unsupported*	*unsupported*	Alg.3.1	*unsupported*	Alg.3.1

Table 5: JavaScript and C libraries for large integer arithmetic. Algorithm marked with a star (*) use Montgomery reduction.

Parallelism. A natural strategy to speed-up computations on a multi-core CPU is to execute certain tasks in parallel. On the server side, defining and executing tasks in parallel is well supported and easy to implement in many programming languages. On the client side, the situation is slightly different. Although current personal devices (notebooks, tablet computers, mobile phones) have all multi-core CPUs and hence, parallelism is possible from a hardware perspective, JavaScript code is intended to be executed in a single thread. Only recent advancements in the area of so-called *web workers* bring concurrency also to JavaScript by allowing web pages to run scripts in background threads. Once created, a web worker runs completely independent of the main script without any shared memory. Communication from and to the web workers goes via an asynchronous event bus. Web workers are already supported by all major web browsers, so performance benefits can be expected on all up-to-date platforms.

The remaining problem is to find a strategy that optimizes the overall benefit of using parallel computing in combination with other optimization techniques. For example, to circumvent the lack of shared memory, passing large precomputation tables for fixed-base exponentiations to different web workers might not be the best strategy. On the other hand, if multiple fixed-base exponentiations

for different bases must be computed, a web worker could be created for each base. The overall computation time would then be decreased by several factors depending on the number of available cores. As the benefit of parallelism strongly relies on the underlying hardware and on the concrete computations to perform, we have excluded this aspect in the following performance analysis.

3.2 Performance Analysis

We are now going to present the results from our experiments of computing modular exponentiations with different optimizations, different libraries, and different runtime environments. All experiments were conducted on the same computer (MacBook Pro 2.9Ghz Intel Core i7) and the same web browser (Firefox v63.0.3).[6] The goal of this subsection is to present the magnitude of what can be expected in practice and to demonstrate that this magnitude corresponds to the theoretical results from Section 2. All results can be reproduced reliably with deviations in a range of about $\pm 5\%$.

Evaluation of Libraries. We first conducted an experiment to evaluate the performance of the different libraries for large number arithmetic. We computed with each library a series of 100 modular exponentiations. Table 6 shows the resulting average running times for a single exponentiation. On the server side, the results are somewhat surprising regarding the time difference between GMP and MiniGMP. There are two main reasons for that. First, GMP implements better algorithms than MiniGMP (see Table 5), and second, GMP provides highly optimized assembly code. In our test environment, turning off assembly optimizations makes GMP approximately three times less efficient.

Regarding the results obtained for JavaScript, we conclude that none of the JavaScript libraries can keep up with native GMP. The comparison of the different JavaScript libraries also points out the impact of selecting the best algorithm, which explains that VJSC and JSBN offer a better performance than Leeman and MiniGMP/WASM.[7] Interestingly, VJSC without Montgomery reduction performs better than JSBN with Montgomery reduction. This shows the importance of other (hidden) factors such as an optimized implementation for the given environment.

Product and Fixed-Base Exponentiation. To analyze the benefits of the optimization techniques from Section 2, we decided to conduct server-side experiments with GMP/MEE and client-side experiments with VJSC. It was required

[6] Using the same testbed, we performed further experiments on different platforms such as tablet computers and mobile phones. We obtained very similar test results on all platforms, but for reasons of brevity, we do not include them in our discussion.

[7] We were surprised to observe that MiniGMP compiled into WASM does not provide an important advantage over pure JavaScript. We have no explanation for this, but from the tests that we conducted, we can exclude that this is due to some communication overhead between WASM and JavaScript. By passing exactly the same amount of data from JavaScript to WASM, we observed that computing n modexps in a single call is almost exactly n times more expensive than computing a single modexp.

	Server		Client			
ℓ	GMP	MiniGMP	VJSC	JSBN	Leeman	MiniGMP/WASM
2048	3.05ms	19.23ms	81.55ms	105.68ms	181.89ms	133.59ms
3072	8.97ms	63.14ms	248.69ms	332.81ms	589.74ms	447.27ms

Table 6: Average running times for modular exponentiations in different libraries.

to adjust VJSC slightly, as VJSC selects the parameter k of Alg.3.1 based on $|p|$ instead of $\ell = |e_i|$, which is sub-optimal for small exponents. The experiments were conducted for problems of size $N \in \{10^2, 10^3, \ldots, 10^6\}$ and two different security strengths $\lambda = 112$ and $\lambda = 128$. The absolute running times were measured over the whole experiment and then divided by the problem size N. We also computed corresponding impact factors to demonstrate the benefit of the optimization algorithms over plain modexp computations.

Using GMP/MEE (see Table 7), short exponents yield the expected performance gain independently of the problem size N (between 7–8 for $\lambda = 112$ and 10–11 for $\lambda = 128$). Also independent of N is the benefit of Alg.2 for product exponentiations, which is between 5–6 times faster than computing plain modexps. For Alg.3.1, the amortization of the precomputation can be observed by looking at the increasing benefit when N gets larger. The measurements also demonstrate that the benefit of short exponents multiples the benefit of the optimization algorithm. For $\lambda = 128$ and $N = 10^6$, for example, fixed-base exponentiations with short exponents results in an impact factor of $99.2 \approx 9.0 * 10.8$.

		$\lambda = 112$				$\lambda = 128$			
N	**Algorithm**	2048/2047		2048/224		3072/3071		3072/256	
100	HAC 14.85	3.049	1	0.435	7.0	8.969	1	0.902	9.9
	Alg.2	0.637	4.8	0.113	27.0	1.708	5.25	0.196	45.8
	Alg.3.1	0.799	3.8	0.104	29.3	1.999	4.5	0.213	42.0
1,000	HAC 14.85	2.980	1	0.360	8.0	8.852	1	0.797	11.1
	Alg.2	0.610	4.9	0.108	27.6	1.508	5.8	0.207	42.8
	Alg.3.1	0.588	5.1	0.079	37.7	1.556	5.7	0.170	52.1
10,000	HAC 14.85	3.008	1	0.367	8.2	8.831	1	0.816	10.8
	Alg.2	0.636	4.7	0.100	30.1	1.518	5.8	0.204	43.3
	Alg.3.1	0.495	6.1	0.066	45.6	1.288	6.9	0.137	64.5
100,000	Alg.3.1	0.422	7.1	0.050	60.2	1.122	7.9	0.111	79.6
1,000,000	Alg.3.1	0.389	7.7	0.044	68.4	0.983	9.0	0.089	99.2

Table 7: Relative running times in milliseconds (1st columns) and impact factors (2nd column) of different algorithms using GMP/MEE.

Using VJSC in a web browser (see Table 8), the resulting impact factors are similar to GMP/MEE. However, some of the values are slightly misleading

because of the less optimized plain modexp implementation in VJSC. This explains that Alg.1 for product exponentiation in VJSC has only a slightly lower impact factor in comparison with Alg.2 in GMP/MEE, although theory predicts a difference of approximately 37%.

Overall, the conducted performance analysis shows that in practice the observed benefits of the optimizations are slightly lower than what could be expected from theory. Possible reasons are manifold. In the theoretical analysis, some simplifications have been made, like for example the counting of squarings and multiplications in the same way. On the other hand, specific optimizations on an implementation level are manifested to varying degrees depending on the computation. The plain modexp in GMP is strongly optimized including Montgomery reduction, straining the theoretical results based on counting multiplications. Nevertheless, the presented optimization techniques accelerate the computation of multiple modexps also in practice by orders of magnitude.

		$\lambda = 112$				$\lambda = 128$			
N	**Algorithm**	2048/2047		2048/224		3072/3071		3072/256	
100	HAC 14.83	81.55	1	11.73	7.0	248.69	1	25.10	9.9
	Alg.1	18.69	4.4	3.22	25.3	58.44	4.3	7.52	33.1
	Alg.3.1	15.16	5.4	2.89	28.2	47.41	5.2	6.14	40.5
1,000	HAC 14.83	81.83	1	11.79	7.0	254.80	1	25.25	10.1
	Alg.1	17.85	4.6	3.11	26.3	55.49	4.6	7.21	35.3
	Alg.3.1	10.81	7.6	1.67	49.0	32.71	7.8	3.72	68.5

Table 8: Relative running times in milliseconds (1st columns) and impact factors (2nd column) of different algorithms using VJSC.

4 Conclusion

Our analysis of modular exponentiation algorithms in this paper demonstrates the potential of the available optimized methods for different types of exponentiation problems. While product exponentiation problems can be solved 5–10 times more efficiently, we can solve large fixed-based exponentiation problems up to two orders of magnitude more efficiently than with conventional methods. Using short-exponent techniques, the impact of these methods can be strengthened by another order of magnitude. The resulting overall benefit is very promising for making cryptographic protocols more efficient, particularly for web applications on the client side. On the server side, we also obtain a considerable speedup, for example for shuffling a list of encryptions in a verifiable mix-net. We expect similar benefits for other cryptographic tasks.

Regarding the available libraries implementing the algorithms presented in this paper, we believe that there is some room for future work. For the best available algorithms for fixed-base exponentiation, Alg.3.2 or HAC 14.117, we were surprised not to find an implementation in any of the libraries we looked at. By looking at the source code of these libraries, we also realized that they

do not always select optimal algorithm parameters. Improving and completing these libraries is an open task, for which this paper provides a solid starting point.

References

1. Baird, L. C.: Big Integer Library by Leemon, `https://github.com/Evgenus/BigInt`
2. Brickell, E.F., Gordon, D.M., McCurley, K.S., Wilson, D.B.: Fast exponentiation with precomputation. In: Rueppel, R.A. (ed.) EUROCRYPT'92, 11th Workshop on the Theory and Application of Cryptographic Techniques. pp. 200–207. LNCS 658, Balatonfüred, Hungary (1992)
3. Giry, D.: Cryptographic Key Length Recommendation, `https://www.keylength.com`
4. Granlund, T.: The GNU Multiple Precision Arithmetic Library – Edition 6.1.2, `https://gmplib.org`, (2016)
5. Haenni, R., Locher, P., Koenig, R.E., Dubuis, E.: Pseudo-code algorithms for verifiable re-encryption mix-nets. FC'17, 21st International Conference on Financial Cryptography. pp. 370–384. LNCS 10323, Silema, Malta (2017)
6. Indutny, F.: BigNum in Pure Javascript, `https://github.com/indutny/bn.js`
7. Koshiba, T., Kurosawa, K.: Short exponent Diffie-Hellman problems. In: Bao, F., Deng, R., Zhou, J. (eds.) PKC'04, 7th International Workshop on Theory and Practice in Public Key Cryptography. pp. 173–186. LNCS 2947, Singapore (2004)
8. Lee, P.J., Lim, C.H.: Method for exponentiation in a public-key cryptosystem. United States Patent No. 5999627 (December 1999)
9. Lim, C.H., Lee, J.P.: More flexible exponentiation with precomputation. In: Desmedt, Y. (ed.) CRYPTO'94, 14th Annual International Cryptology Conference on Advances in Cryptology. pp. 95–107. LNCS 839, Santa Barbara, USA (1994)
10. Menezes, A.J., van Oorschot, P.C., Vanstone, S.A.: Handbook of Applied Cryptography. CRC Press, Boca Raton, USA (1996)
11. Möller, N.: Mini-GMP – A Minimalistic Implementation of a GNU GMP Subset, `https://godoc.org/modernc.org/minigmp`
12. Perlitch, A.: JSBN – Javascript Big Number, `https://github.com/andyperlitch/jsbn`
13. Terelius, B., Wikström, D.: Proofs of restricted shuffles. In: Bernstein, D.J., Lange, T. (eds.) AFRICACRYPT'10, 3rd International Conference on Cryptology in Africa. pp. 100–113. LNCS 6055, Stellenbosch, South Africa (2010)
14. Wikström, D.: A commitment-consistent proof of a shuffle. In: Boyd, C., González Nieto, J. (eds.) ACISP'09, 14th Australasian Conference on Information Security and Privacy. pp. 407–421. LNCS 5594, Brisbane, Australia (2009)
15. Wikström, D.: User Manual for the Verificatum Mix-Net – VMN Version 3.0.3. Verificatum AB, Stockholm, Sweden (2018)
16. Wikström, D.: GMP Modular Exponentiation Extension, `https://github.com/verificatum/verificatum-gmpmee`
17. Wikström, D.: Verificatum JavaScript Cryptography Library, `https://github.com/verificatum/verificatum-vjsc`
18. Wu, T.: RSA and ECC in JavaScript, `http://www-cs-students.stanford.edu/~tjw/jsbn`