Outsourcing Modular Exponentiation in Cryptographic Web Applications*

Pascal Mainini and Rolf Haenni

Bern University of Applied Sciences, CH-2501 Biel/Bienne, Switzerland
{pascal.mainini,rolf.haenni}@bfh.ch

Abstract. Modern web applications using advanced cryptographic methods may need to calculate a large number of modular exponentiations. Performing such calculations in the web browser efficiently is a known problem. We propose a solution to this problem based on outsourcing the computational effort to untrusted exponentiation servers. We present several efficient outsourcing protocols for different settings and a practical implementation consisting of a JavaScript client library and a server application. Compared to browser-only computation, our solution improves the overall computation time by an order of magnitude.

1 Introduction

Due to the limited performance of interpreted JavaScript code, web browsers are relatively slow computational environments compared to high-performance servers running compiled native or pre-compiled VM code. With recent performance improvements of the most common JavaScript engines, this is no longer a real limitation for most modern web applications. However, exceptionally expensive client-side computations are required in applications of public-key cryptography. Usually, the most critical operation in such applications is modular exponentiation (modexp), i.e., the computation of $z = x^y \bmod n$ for given integer inputs x, y, and n of length 2048 bits or higher. While web browsers compute modexps efficiently to establish TLS connections to servers, JavaScript developers have no built-in access to such a primitive, not even using the recently standardized Web Cryptography API.[1] To allow the development of cryptographic code in JavaScript, several libraries provide an API for dealing with large integers and an implementation of the most important arithmetic operations. With the best libraries available today, computing a small number of modexps is possible in a modern web browser, but the performance is more than one order of magnitude inferior compared to native code.[2]

*This is an extended version of a paper accepted and presented at the Voting'18 workshop of the Financial Cryptography and Data Security 2018 conference. It will be included in the conference's LNCS proceedings and available on the Springer web site.

[1]The Web Cryptography API offers operations for Diffie-Hellman key exchanges and DSA signatures, but currently only elliptic curves are supported. Therefore, we do not see a way of exploiting this interface for computing modular exponentiations.

[2]We expect significant performance improvements in libraries making use of the recently introduced WebAssembly technology for web browsers.

If a large number of modexps needs to be computed in a cryptographic application, the limited performance of JavaScript leads to major usability problems. In such cases, calculating the modexps may take several minutes, which is not tolerated by most users. Examples of such applications exist in the context of cryptographic voting protocols. In [5,6], for example, the web client used for vote casting requires up to $2k$ modexps in a k-out-of-n election. In parliamentary elections, where k represents the number of seats and n the number of candidates, it can happen that several hundred modexps need to be computed in the web browser for these protocols. The problem gets even worse in advanced voting protocols with extended security properties. For instance, in the protocol presented in [9], depending on the size of the electorate and the chosen security parameters, several thousand modexps may be required for ensuring everlasting privacy while casting a vote. Cases like this cannot be handled in reasonable time by JavaScript engines in current web browsers.

To solve this problem, we propose to outsource modexp computations to external *exponentiation servers*. Note that modexp computations in cryptographic applications often involve secret values such as private keys or encryption randomizations. Therefore, the main challenge of this approach is to ensure that the input parameters—the base x, the exponent y, or both x and y—and the output parameter z remain secret, even if exponentiation servers are not fully trustworthy (the modulus n is usually a public parameter). Secret parameters must therefore be cryptographically blinded before sending them out. Another challenge is to ensure the correctness of the results in the presence of servers that may act maliciously, or at least to detect such attacks with adequate probability. Client-side algorithms for dealing with these challenges must do so without falling back on expensive operations.

A very different, but more common approach to speed up expensive cryptographic computations on limited devices is working with elliptic curves. For providing equivalent security, point multiplications on such curves are significantly faster than exponentiations in modular groups. The main problem with elliptic curves in voting protocols such as the ones mentioned above is the difficulty of encoding complex voting options as curve points (while preserving the encryption homomorphism). In [5,6], for example, voting options are encoded as a product of prime numbers. Using modular groups, such products can be efficiently aggregated under encryption and decoded after decryption. We are not aware of an equivalent encoding for elliptic curves.

1.1 Related Work

There is a large amount of literature about outsourcing modular exponentiations. The approaches can be classified along two lines. The first is the number of required exponentiation servers. There are approaches for one, two, or four servers. In the two-server and four-server cases, it is assumed that the servers do not collude and that they can be reached over confidential channels. No such assumptions exist in the one-server case. Server-side authentication is a requirement in all cases to ensure the origin of the server responses. The second

classification criterion is the adversary model attributed to the exponentiation servers. The main differentiation is between semi-honest and malicious servers. In the semi-honest model, no particular measures need to be taken to ensure the correctness of the responses.

A comprehensive analysis and compilation of one-server protocols for semi-honest adversaries can be found in [3]. This document also contains proven optimality results for certain protocols in form of lower bounds for the total number of necessary modular multiplications. The main drawback of most one-server protocols is the assumption that random pairs $(r, g^r \bmod n)$ can be generated efficiently by the client, where g is a fixed value. This may be difficult to achieve in a web application. Some protocols also require a large number of modular multiplications on the client, which reduces the potential performance gain of the outsourcing process. Similar remarks hold for the protocols presented in [1,8], which consider the one-server case in the presence of malicious adversaries.

The main reference in the literature on two-server outsourcing protocols is the paper by Hohenberger and Lysyanskaya [7]. They introduced a property called β-*checkability*, which means that deviations from the protocol by malicious servers are detected by the client with probability β or greater. Some other authors proposed similar protocols with improved efficiency [2,12]. A very different two-server approach based on the subset-sum-problem has been proposed in [10]. For a more detailed overview of the available references and methods, we refer to the summary given in [11].

1.2 Contribution and Paper Overview

The contribution of this paper consist of three parts. In Section 2, we present outsourcing protocols for the six most important settings. Except for the number of involved servers, our protocols are the most efficient ones in the literature, with only up to four client-side modular multiplications during the execution of the protocols. A detailed performance comparison is shown in Table 1.

The second contribution is the implementation of the outsourcing protocols from Section 2. To the best of our knowledge, such an implementation has not yet existed before. To enable the embedding of our implementation in a practical system, we provide a client library in JavaScript, which handles the secure communication with the servers and executes the outsourcing algorithms. The flexible architecture of this library enables the inclusion of further outsourcing algorithms from the literature. We also provide a server application in Java, which can be deployed on ordinary server infrastructure. Details of our implementation are given in Section 3.

The third contribution of this paper is the experimental performance analysis of our implementation in Section 4. Compared to browser-only computation, the analysis shows that our implementation improves the overall computation time by an order of magnitude. In the two use cases mentioned in the introduction, in which a large number of modexps need to be computed for casting a vote in the web browser, this solves the aforementioned usability problem. In Section 5, we summarize our findings and mention some remaining open problems.

Paper	Protocol Name	Secret		Number of					β
		Base	Exp.	Servers	ModExps	Mult.	Inv.	Rand.	
[3]	Protocol 7	yes	no	1	2	3	1	3	0
	Protocol 5	no	yes	1	$s \geq 1$	$\frac{\log p}{s+1}$	–	–	0
	Protocol 6	yes	yes	1	$s \geq 2$	$\frac{\log p}{s}$	–	2	0
this	Algorithm 1	yes	no	2	1	2	–	–	0
	Algorithm 2	no	yes	2	1	1	–	–	0
	Algorithm 3	yes	yes	4	1	4	–	–	0
	Algorithm 4	yes	no	2	2	2	–	–	$1/2$
	Algorithm 5	no	yes	2	2	1	–	–	$1/2$
	Algorithm 6	yes	yes	4	2	4	–	–	$1/2$
[7]	*Exp*	yes	yes	2	4	9	5	6	$1/2$
[2]	**Exp**	yes	yes	2	3	7	3	5	$2/3$

Table 1: Performance comparison of different outsourcing protocols. Each row of the table shows the number of servers involved in the corresponding protocol, the number of modexps computed by each server, the number of modular multiplications computed by the client, the number of multiplicative inverses computed by the client, the number of random pairs $(r, g^r \bmod n)$ generated by the client, and the checkability factor β. Some one-server protocols from [3] are omitted, for example the ones that are limited to a fixed base or the ones that are special cases of others.

2 Outsourcing Protocols

Most outsourcing algorithms in the literature are based on the same basic principles. Privacy is achieved by blinding x and y based on the homomorphic property of the exponentiation function, which comes in two flavors, depending on whether x or y is fixed:

$$\exp(x, y_1 + y_2) = x^{y_1+y_2} = x^{y_1} x^{y_2} = \exp(x, y_1) \exp(x, y_2),$$
$$\exp(x_1 x_2, y) = (x_1 x_2)^y = x_1^y x_2^y = \exp(x_1, y) \exp(x_2, y).$$

These properties also hold if all multiplications are performed modulo n and all additions modulo $\phi(n)$, where ϕ denotes the Euler function. Since $\phi(n)$ cannot be computed efficiently without knowledge of the prime factors of n, we restrict ourselves to the particular case where n is prime and $\phi(n) = n-1$. We emphasize this point by writing $x^y \bmod p$ instead of $x^y \bmod n$.

From a group theory perspective, we perform exponentiations in the multiplicative group $\mathbb{Z}_p^* = \{1, \ldots, p-1\}$ of integers modulo p, or in corresponding subgroups $\langle x \rangle \subseteq \mathbb{Z}_p^*$ generated by x. We denote such a subgroup by $\mathbb{G}_q = \langle x \rangle$ and assume that its order q (which divides $p-1$) is known in the given context. Operations in the exponent can then be computed in the additive group $\mathbb{Z}_q = \{0, \ldots, q-1\}$ of integers modulo q. Note that $\mathbb{Z}_p^*$ (and large subgroups $\mathbb{G}_q \subseteq \mathbb{Z}_p^*$) are by far the most widely used groups in cryptographic applications based on the discrete logarithm (DL), computational Diffie-Hellman (CDH), or decisional Diffie-Hellman (DDH) assumption. Other popular groups such as el-

liptic curves are not treated explicitly in this paper. However, all algorithms presented in this section generalize naturally to arbitrary groups.

In the next subsection, we introduce the most basic outsourcing protocols for semi-honest servers. The first protocol protects the secrecy of x, the second the secrecy of y, and the third the secrecy of x and y (every protocol protects the secrecy of $z = x^y \bmod p$). In Section 2.2, we show that each protocol of Section 2.1 can be extended easily to reach $1/2$-checkability in the presence of malicious servers. We present each two-server protocol by an algorithm $\mathsf{ModExp}(x, y, p, q, S_1, S_2)$ executed by the client, and each four-server protocol by an algorithm $\mathsf{ModExp}(x, y, p, q, S_1, S_2, S_3, S_4)$. These algorithms contain calls to $S_i.\mathsf{ModExp}(x_i, y_i, p)$, which invoke the transmission of x_i, y_i, and p to server $i \in \{1, 2, 3, 4\}$, and the receipt of the server's response over a secure channel. In every protocol, we assume that the servers are non-colluding.[3]

2.1 Semi-Honest Servers

In the semi-honest adversary model, it is assumed that every server involved in the outsourcing protocol executes $S_i.\mathsf{ModExp}(x_i, y_i, p)$ faithfully, i.e., the server always returns the correct result of computing $x_i^{y_i} \bmod p$ to the client.

Secret Base. If the base $x \in \mathbb{G}_q$ is secret and the exponent $y \in \mathbb{Z}_q$ is public, only y can be sent in cleartext to the involved servers. However, by decomposing x into values $x_1 \in_R \mathbb{G}_q$ (picked uniformly at random from $\mathbb{G}_q$) and $x_2 = x\, x_1^{-1} \bmod p$, which implies $x = x_1 x_2 \bmod p$, we can apply the homomorphic property of the exponentiation function,

$$x^y \equiv (x_1 x_2)^y \equiv x_1^y x_2^y \pmod{p},$$

to split the computation of $x^y \bmod p$ into $x_1^y \bmod p$ for the first server and $x_2^y \bmod p$ for the second server. Since x_1 is a random value and x_2 is derived from a random value, x remains entirely hidden from both servers. A disadvantage of this simple approach is that the client needs to compute the multiplicative inverse $x^{-1} \bmod p$, which is a relatively expensive operation.

A slightly different approach consists in selecting values $x_1 \in_R \mathbb{G}_q$ and $x_2 = x\, x_1 \bmod p$, which implies $x = x_1^{-1} x_2 \bmod p$. By applying again the homomorphic property of the exponentiation function, we obtain

$$x^y \equiv (x_1^{-1} x_2)^y \equiv (x_1^{-1})^y x_2^y \equiv x_1^{-y} x_2^y \pmod{p},$$

which implies that $x_1^{-y} \bmod p$ can be given to the first server and $x_2^y \bmod p$ to the second server. For the same reasons as above, x remains entirely hidden from both servers. The details of this procedure are depicted in Algorithm 1. Note that the main computational work for the client consists of two modular multiplications in $\mathbb{G}_q$ (we assume that operations in $\mathbb{Z}_q$ are negligible).

[3] Requiring multiple non-colluding servers is admittedly a strong assumption. We believe that this assumption can be justified, if adequate organizational measures are put in place. Otherwise, we suggest extending our protocols to three or more servers or considering the one-server protocols from [3].

Secret Exponent. In the opposite case of a public base $x \in \mathbb{G}_q$ and a secret exponent $y \in \mathbb{Z}_q$, only x can be sent in cleartext to the involved servers. Here, a viable solution results directly from applying the homomorphic property of the exponentiation function to $y = y_1 + y_2 \bmod q$ for values $y_1 \in_R \mathbb{Z}_q$ and $y_2 = y - y_1 \bmod q$:

$$x^y \equiv x^{y_1+y_2} \equiv x^{y_1} x^{y_2} \pmod{p}.$$

Algorithm 2 shows the procedure of outsourcing $x^{y_1} \bmod p$ to the first server and $x^{y_2} \bmod p$ to the second server. Since y_1 is a random value and y_2 is derived from a random value, y remains entirely hidden from both servers. Here the workload for the client is a single modular multiplication in $\mathbb{G}_q$.

Algorithm: $\mathsf{ModExp}(x, y, p, q, S_1, S_2)$
Input: Secret base $x \in \mathbb{G}_q$
Public exponent $y \in \mathbb{Z}_q$
Prime modulus p
Group order q
Semi-honest servers S_1, S_2
$x_1 \in_R \mathbb{G}_q$, $x_2 \leftarrow x\, x_1 \bmod p$
$z_1 \leftarrow S_1.\mathsf{ModExp}(x_1, -y \bmod q, p)$
$z_2 \leftarrow S_2.\mathsf{ModExp}(x_2, y, p)$
return $z_1 z_2 \bmod p$

Algorithm 1: Two-server outsourcing protocol for secret base and public exponent.

Algorithm: $\mathsf{ModExp}(x, y, p, q, S_1, S_2)$
Input: Public base $x \in \mathbb{G}_q$
Secret exponent $y \in \mathbb{Z}_q$
Prime modulus p
Group order q
Semi-honest servers S_1, S_2
$y_1 \in_R \mathbb{Z}_q$, $y_2 \leftarrow y - y_1 \bmod q$
$z_1 \leftarrow S_1.\mathsf{ModExp}(x, y_1, p)$
$z_2 \leftarrow S_2.\mathsf{ModExp}(x, y_2, p)$
return $z_1 z_2 \bmod p$

Algorithm 2: Two-server outsourcing protocol for public base and secret exponent.

Secret Base and Exponent. If both the base $x \in \mathbb{G}_q$ and the exponent $y \in \mathbb{Z}_q$ are secret, we can combine the two protocols from above to hide both values from the servers. As one can see in Algorithm 3, the resulting protocol requires four non-colluding servers, to which different tasks are assigned. These tasks result from decomposing x into $x_1 \in_R \mathbb{G}_q$ and $x_2 = x\, x_1 \bmod p$, as in Algorithm 1, and y into $y_1 \in_R \mathbb{Z}_q$ and $y_2 = y - y_1 \bmod q$, as in Algorithm 2. By applying both flavors of the homomorphic property simultaneously,

$$\begin{aligned} x^y &\equiv (x_1^{-1} x_2)^{y_1+y_2} \equiv (x_1^{-1} x_2)^{y_1} (x_1^{-1} x_2)^{y_2} \\ &\equiv x_1^{-y_1} x_1^{-y_2} x_2^{y_1} x_2^{y_2} \pmod{p}, \end{aligned}$$

we obtain respective tasks for the four servers. Note that any pair of colluding servers can reconstruct at least one of the two secret values. On the other hand, a single server alone learns nothing about x or y based on the two random input values. In the resulting protocol depicted in Algorithm 3, the workload for the client consists of four modular multiplications in $\mathbb{G}_q$.

2.2 Malicious Servers

In the literature on outsourcing modular exponentiation in the presence of malicious servers, various authors have applied a similar technique to detect a cheating server

Algorithm: $\mathsf{ModExp}(x, y, p, q, S_1, S_2, S_3, S_4)$
Input: Secret base $x \in \mathbb{G}_q$, secret exponent $y \in \mathbb{Z}_q$
Prime modulus p, group order q
Semi-honest servers S_1, S_2, S_3, S_4
$x_1 \in_R \mathbb{G}_q$, $x_2 \leftarrow x\, x_1 \bmod p$
$y_1 \in_R \mathbb{Z}_q$, $y_2 \leftarrow y - y_1 \bmod q$
$z_1 \leftarrow S_1.\mathsf{ModExp}(x_1, -y_1 \bmod q, p)$
$z_2 \leftarrow S_2.\mathsf{ModExp}(x_1, -y_2 \bmod q, p)$
$z_3 \leftarrow S_3.\mathsf{ModExp}(x_2, y_1, p)$
$z_4 \leftarrow S_4.\mathsf{ModExp}(x_2, y_2, p)$
return $z_1 z_2 z_3 z_4 \bmod p$

Algorithm 3: Four-server outsourcing protocol for secret base and secret exponent.

[7,2,1]. The idea consists in challenging each involved server with at least one additional modexp computation, but without letting the server know which one is the real task and which one the challenge. If x_i and y_i are the real parameters and x'_i and y'_i the challenges for server S_i, then this is achieved by randomizing the order of the calls $S_i.\mathsf{ModExp}(x_i, y_i, p)$ and $S_i.\mathsf{ModExp}(x'_i, y'_i, p)$. In the simplest possible case, the same random challenge is sent to multiple servers. The client then checks the consistency of the servers' responses and aborts the protocol in case of a mismatch. This general approach can be applied to every protocol from the previous subsection in slightly different forms.

To pass the above consistency check, a cheating server must respond correctly to the challenge, but if the challenge and the real task are indistinguishable for the server, the chance of identifying the challenge is $1/2$. If two servers are cheating simultaneously, the chance of guessing both challenges is $1/4$, and if four servers are cheating, the chance is $1/16$. Therefore, the chance that an attack by malicious servers remains undetected is always at most $1/2$, which implies that a protocol equipped with this technique offers $1/2$-checkability. Note that a higher value for β can be achieved by sending multiple challenges in random order to each server. Generally, we obtain $\beta = \frac{c}{c+1}$ for sending $c \geq 0$ challenges in random order to to each server.

Secret Base. If $x \in \mathbb{G}_q$ is secret and $y \in \mathbb{Z}_q$ is public, the parameters of the challenges sent to the servers must be indistinguishable from those of Algorithm 1. Therefore, while choosing the base $x' \in_R \mathbb{G}_q$ at random for making it indistinguishable from the random values x_1 and x_2, the same public exponents must be used, i.e., $-y \bmod q$ for S_1 and y for S_2. If we extend Algorithm 1 with corresponding calls $S_1.\mathsf{ModExp}(x', -y \bmod q, p)$ and $S_2.\mathsf{ModExp}(x', y, p)$, and randomize the order of the calls using a random bit $r \in_R \{0, 1\}$, the client obtains values $z'_1 = (x')^{-y} \bmod p$ and $z'_2 = (x')^y \bmod p$. Their consistency can be checked by $z'_1 z'_2 \bmod p = 1$ using a single additional multiplication. This whole procedure is shown in Algorithm 4.

Secret Exponent. If $x \in \mathbb{G}_q$ is public and $y \in \mathbb{Z}_q$ is secret, the situation is reversed. For making the challenge parameters indistinguishable, we must pick a random exponent $y' \in_R \mathbb{Z}_q$ while using the same public base x. This means that exactly the same challenge $S_i.\mathsf{ModExp}(x, y', p)$ is sent to both servers and that the consistency of their responses, $z'_i = x^{y'} \bmod p$, can be tested by verifying if they are identical. In Algorithm 5, we show the resulting protocol obtained as an extension of Algorithm 2.

Secret Base and Exponent. In the four-server protocol of Algorithm 3, where both x and y are secret, every servers receives two random values. Therefore, for making the challenge parameters indistinguishable from the real parameters, each server is challenged by $S_i.\mathsf{ModExp}(x', y', p)$ with the same two random values $x' \in_R \mathbb{G}_q$ and $y' \in \mathbb{Z}_q$. From getting four identical responses $z'_i = (x')^{y'} \bmod p$, the client concludes that they have all been computed correctly. Algorithm 6 depicts the protocol obtained from extending Algorithm 3 accordingly.

Algorithm: $\mathsf{ModExp}(x, y, p, q, S_1, S_2)$
Input: Secret base $x \in \mathbb{G}_q$
Public exponent $y \in \mathbb{Z}_q$
Prime modulus p
Group order q
Malicious servers S_1, S_2
$x_1 \in_R \mathbb{G}_q$, $x_2 \leftarrow x\, x_1 \bmod p$, $x' \in_R \mathbb{G}_q$
$r \in_R \{0, 1\}$
if $r = 0$ **then**
  $z_1 \leftarrow S_1.\mathsf{ModExp}(x_1, -y \bmod q, p)$
  $z'_1 \leftarrow S_1.\mathsf{ModExp}(x', -y \bmod q, p)$
  $z_2 \leftarrow S_2.\mathsf{ModExp}(x_2, y, p)$
  $z'_2 \leftarrow S_2.\mathsf{ModExp}(x', y, p)$
else
  $z'_1 \leftarrow S_1.\mathsf{ModExp}(x', -y \bmod q, p)$
  $z_1 \leftarrow S_1.\mathsf{ModExp}(x_1, -y \bmod q, p)$
  $z'_2 \leftarrow S_2.\mathsf{ModExp}(x', y, p)$
  $z_2 \leftarrow S_2.\mathsf{ModExp}(x_2, y, p)$
if $z'_1 z'_2 \bmod p = 1$ **then**
  return $z_1 z_2 \bmod p$
else
  return $\perp$

Algorithm 4: Two-server outsourcing protocol for secret base and public exponent with $\beta = 1/2$.

Algorithm: $\mathsf{ModExp}(x, y, p, q, S_1, S_2)$
Input: Public base $x \in \mathbb{G}_q$
Secret exponent $y \in \mathbb{Z}_q$
Prime modulus p
Group order q
Malicious servers S_1, S_2
$y_1 \in_R \mathbb{Z}_q$, $y_2 \leftarrow y - y_1 \bmod q$, $y' \in_R \mathbb{Z}_q$
$r \in_R \{0, 1\}$
if $r = 0$ **then**
  $z_1 \leftarrow S_1.\mathsf{ModExp}(x, y_1, p)$
  $z'_1 \leftarrow S_1.\mathsf{ModExp}(x, y', p)$
  $z_2 \leftarrow S_2.\mathsf{ModExp}(x, y_2, p)$
  $z'_2 \leftarrow S_2.\mathsf{ModExp}(x, y', p)$
else
  $z'_1 \leftarrow S_1.\mathsf{ModExp}(x, y', p)$
  $z_1 \leftarrow S_1.\mathsf{ModExp}(x, y_1, p)$
  $z'_2 \leftarrow S_2.\mathsf{ModExp}(x, y', p)$
  $z_2 \leftarrow S_2.\mathsf{ModExp}(x, y_2, p)$
if $z'_1 = z'_2$ **then**
  return $z_1 z_2 \bmod p$
else
  return $\perp$

Algorithm 5: Two-server outsourcing protocol for public base and secret exponent with $\beta = 1/2$.

3 Practical Implementation

Despite the large amount of literature on the subject of outsourcing modular exponentiation, we were not able to find practical and implemented solutions. However, for validating the use cases in cryptographic voting protocols from Section 1, we require such a practical implementation and we thus provide it as part of our contribution. We have defined two main objectives for the implementation:

– Providing a robust API for integration into cryptographic web applications.

Algorithm: $\mathsf{ModExp}(x, y, p, q, S_1, S_2, S_3, S_4)$

Input: Secret base $x \in \mathbb{G}_q$, secret exponent $y \in \mathbb{Z}_q$
Prime modulus p, group order q
Malicious servers S_1, S_2, S_3, S_4

$x_1 \in_R \mathbb{G}_q$, $x_2 \leftarrow x\, x_1 \bmod p$, $x' \in_R \mathbb{G}_q$
$y_1 \in_R \mathbb{Z}_q$, $y_2 \leftarrow y - y_1 \bmod q$, $y' \in_R \mathbb{Z}_q$
$r \in_R \{0, 1\}$
if $r = 0$ **then**
  $z_1 \leftarrow S_1.\mathsf{ModExp}(x_1, -y_1 \bmod q, p)$
  $z_1' \leftarrow S_1.\mathsf{ModExp}(x', y', p)$
  $z_2 \leftarrow S_2.\mathsf{ModExp}(x_1, -y_2 \bmod q, p)$
  $z_2' \leftarrow S_2.\mathsf{ModExp}(x', y', p)$
  $z_3 \leftarrow S_3.\mathsf{ModExp}(x_2, y_1, p)$
  $z_3' \leftarrow S_3.\mathsf{ModExp}(x', y', p)$
  $z_4 \leftarrow S_4.\mathsf{ModExp}(x_2, y_2, p)$
  $z_4' \leftarrow S_4.\mathsf{ModExp}(x', y', p)$
else
  $z_1' \leftarrow S_1.\mathsf{ModExp}(x', y', p)$
  $z_1 \leftarrow S_1.\mathsf{ModExp}(x_1, -y_1 \bmod q, p)$
  $z_2' \leftarrow S_2.\mathsf{ModExp}(x', y', p)$
  $z_2 \leftarrow S_2.\mathsf{ModExp}(x_1, -y_2 \bmod q, p)$
  $z_3' \leftarrow S_3.\mathsf{ModExp}(x', y', p)$
  $z_3 \leftarrow S_3.\mathsf{ModExp}(x_2, y_1, p)$
  $z_4' \leftarrow S_4.\mathsf{ModExp}(x', y', p)$
  $z_4 \leftarrow S_4.\mathsf{ModExp}(x_2, y_2, p)$
if $z_1' = z_2' = z_3' = z_4'$ **then**
  return $z_1 z_2 z_3 z_4 \bmod p$
else
  return $\perp$

Algorithm 6: Four-server outsourcing protocol for secret base and secret exponent with $\beta = 1/2$.

– Supporting performance measurements and comparisons of different protocols.

Our solution, which we call *famodulus*[4], fulfills both objectives. It consists of the following three logically distinct components:[5]

– *famodulus-client*, a JavaScript library for outsourcing modexp calculations to *famodulus-server* (in the current version, only Algorithms 2 and 5 are implemented),
– *famodulus-server*, an implementation of the exponentiation server,
– *famodulus-demo*, a comprehensive demonstrator application using *famodulus-client* and *famodulus-server*.

In Sections 3.1 and 3.2, we further describe the *famodulus-server* and *famodulus-client* components. *famodulus-demo*, which is a simple HTML5 web application used for test-

[4] *famodulus* is a combination of the Latin words *famulus* (servant) and *modulus* (measure), i.e., *famodulus* is a servant for modular exponentiation calculations.

[5] All three components have been released as open-source software under the MIT license, see https://github.com/mainini/famodulus.

ing and demos, consists of a user interface for outsourcing single or multiple modular exponentiations using *famodulus-client* to the servers, and provides support for parameter generation and execution time measurements. Communication between the client and the servers takes place over a minimal RESTful interface, which is also described in Section 3.2.

3.1 Client Library

In order to provide an API for cryptographic web applications to outsource modexp calculations, client code for the web browser currently has to be written in JavaScript. Our library focuses on clean and robust implementation as well as on performance and extensibility. Even if a considerable amount of deployed web browsers are still not supporting the full JavaScript ES6 specification, our library makes use of some of its advanced functionalities.[6] We expect an even broader adoption for ES6 soon. Where browser upgrades are not easily possible, so-called *polyfills* may be loaded by the application to support the missing language features.

Client-Side Technologies. Recently, JavaScript has also gained importance on the server side with the rise of *Node.js* in the last few years.[7] *Node.js* brings a widely adopted module system, which simplifies development and supports modularity of JavaScript code. For this reason, *famodulus-client* has been developed as a *Node.js* module, which gets transformed into a single file for the browser using *browserify*.[8] A side effect of this development model is the simplification of unit tests, which do not necessarily require a browser for execution. In principle, *famodulus-client* could thus also be used for outsourcing modexp calculations from a server running in JavaScript, even though calculating modexps through native bindings instead of sending them over the network would probably be a more sensible choice.

Outsourcing protocols require client-side calculations with big integers in JavaScript, which, as opposed to Java, has no built-in support for such types. We have thus conducted a small benchmark of libraries for big integer operations in JavaScript before starting development, focusing on performance for the required arithmetic operations. Based on the results, two libraries where considered for *famodulus-client*, the *BigInt* library by B. Leemon[9] and the *Verificatum JavaScript Cryptographic Library* (VJSC)[10] by D. Wikström, with the latter being slightly faster. We finally decided to use Leemon's library, mostly due to licensing concerns. While working with this library, we encountered critical efficiency problems with the bigInt2str and str2bigInt functions for converting big integers into strings and vice versa. By rewriting these functions, we improved their performance by an order of magnitude.

Modexp Computations. In an application of *famodulus-client* in the web browser, a global FamodulusClient object is exported, which is initialized with a list of exponentiation servers and additional configuration values. After initialization, functions for outsourcing modexps according to the different protocols can be invoked.

[6] See http://www.ecma-international.org/ecma-262/6.0/

[7] See https://nodejs.org

[8] See http://browserify.org

[9] See https://www.npmjs.com/package/BigInt

[10] See http://www.verificatum.com/html/product_vjsc.html

```
const servers = ['server_1', 'server_2'];
const checked = true;

let fam = new FamodulusClient(servers, checked);
fam.decExponent([{b: '2', e: '4', m: '5'},{b: '4', e:'2', m
    :'5'}]).then(result => {
   // do something with result
});
```

Listing 1: Outsourcing two simultaneous modexps to two servers using *famodulus-client* and Algorithm 5.

A code example of using *famodulus-client* for outsourcing a batch of two modexps, $2^4 \bmod 5$ and $4^2 \bmod 5$, is shown in Listing 1. The flag checked in the constructor of the FamodulusClient object indicates that servers are possibly malicious and that their responses need to be checked using the techniques from Section 2.2. The function decExponent performs a decomposition of the exponent in order to protect its secrecy. This setting corresponds to the outsourcing protocol of Algorithm 5. Switching the checked flag to false leads to invocation of the unchecked version of the protocol in Algorithm 2.

3.2 Exponentiation Server

The main objective of an exponentiation server according to our definition is to provide efficient modular exponentiation calculations. It should also provide a convenient interface for submitting calculation tasks and a secure channel for the transmission of the parameters and the responses. *famodulus-server* fulfills these requirements.

Server-Side Technologies. For ease of integration with current electronic voting projects at our institute, we decided to implement the exponentiation server in Java. This choice of platform has no influence on the functionality, and we consider porting the exponentiation server to another platform or programming language to be straightforward. While Java provides a reasonably efficient modexp implementation, we have decided to rely upon the native *GNU Multiple Precision Arithmetic Library* (GMPLib) for all server-side calculations.[11] A short series of benchmarks conducted during an initial evaluation phase indicates a performance gain of roughly a factor of four compared to Java's built-in BigInteger.modPow() method. We conducted our measurements using OpenJDK 1.8 on the Linux platform.

As of today, RESTful interfaces as defined by R. T. Fielding can be considered state-of-the-art for interaction between web applications and back-end services on the server side [4]. *famodulus-server* offers a very simple, yet flexible RESTful interface to submit modexp calculations and obtain corresponding results. Its implementation is based on JAX-RS, which specifies an API for RESTful services in Java. While multiple implementations for JAX-RS exist, we have chosen *Jersey*, the reference implementation.[12] Jersey applications offer greatest flexibility with support for deployment to various containers. By expecting that *famodulus-server* will almost always be deployed

[11] See https://gmplib.org

[12] See https://jersey.java.net

standalone on a server for optimal performance, we provide a configuration using the modern *Grizzly* standalone HTTP server.[13]

Modexp Computations. Modular exponentiations are submitted to *famodulus-server* over a secure HTTPS connection. Note that using TLS on top of HTTP is a critical precondition for protecting the secrecy of the parameters in outsourcing algorithms with multiple servers. The parameters (base, exponent, modulus) are encoded as JSON data enclosed in the body of the HTTP POST request. The JSON data format is widely used in RESTful interfaces. A single modexp is encoded as follows:

```
{"b": String, "e": String, "m": String}
```

The three attributes `"b"` (base), `"e"` (exponent), and `"m"` (base) are encoded as hexadecimal strings. The reason for this encoding is the missing data type for big integers in JavaScript, which makes parsing the JSON data impossible on the client side when the numbers exceed $2^{53} - 1$.

Each request submitted to the server must contain at least one single modexp in the JSON data format given above, it can however also contain many modexps at the same time. In practical applications, multiple modexps often share common parameters, for instance the prime modulus. For efficiency reasons, our JSON data format allows the definition of a common base, a common exponent, a common modulus, or a combination of common base, exponent, or modulus. These are the default values for modexps which do not provide the corresponding parameter. The complete message sent in a single HTTP Post request to the server then looks as follows:

```
{"b": String, "e": String, "m": String,
 "modexps": [modexp_1, ..., modexp_n],
 "brief": Boolean}
```

The first three lines are the default parameters, as described above, and may be omitted individually. The `"modexps"` attribute is a list of one or multiple modexps declarations, possibly with missing parameters. If parameters are missing, they are substituted in the calculations by the default values. The final attribute instructs the server to either return the results together with the full query (`"brief": false`) or the results only (`"brief": true`), depending on the client's needs.

4 Performance Analysis

This section describes the experimental performance analysis of *famodulus* that we have conducted. All test runs were conducted on a single machine with a Core i7 CPU (eight cores), running at 1.73 GHz with 8 GB of RAM, and with an installation of Debian GNU/Linux from the current testing branch. During the experiments, two *famodulus-server* instances were started by assigning corresponding processes to different CPU cores. Adherence to this setting was monitored. Processes which were not required for the experiments, for monitoring, or for the operating system itself have been stopped. Memory consumption during the experiments was monitored throughout. The experiments themselves were conducted with an off-the-shelf Firefox 50.1.0 web browser, with no specific configuration and with network communication taking place over the loop-back device. Conducting performance tests locally is a reasonable

[13] See https://grizzly.java.net

choice for our setting, given the fact that typical Internet network delays are several orders of magnitude smaller than the effective computing times spent on the servers. The Firefox process has been pinned to a separate CPU core.

4.1 Server-Only and Browser-Only Computations

The first series of experiments have been conducted on server-only and browser-only configurations in batches of 50, 100, 500, and 1000 modexps for modulus bit lengths of 1024, 2048, and 3072 bits.[14] The goal was to obtain an estimation of the performance difference of computing modular exponentiations using the native GMPLib and the JavaScript engine of the Firefox web browser. We selected the VJSC library for this purpose to obtain the best possible browser-only results. On the server side, we conducted the measurements using *famodulus-server*. Currently, no performance optimizations other than using GMPLib have been implemented, i.e., modexp computations are computed sequentially on a single CPU core upon receiving a batch of such tasks.

The results of our experiments are depicted in Table 2. They show that—depending on the bit lengths of the parameters—executing native code on the server is up to 18 times faster than corresponding JavaScript calculations in the web browser. The results also show that the browser-only running times become problematical from a user perspective for batch sizes of 100 modexps or more and bit lengths of 2048 bits or more. We get approximately 20 seconds for the 100/2048-setting and more than 10 minutes for the 1000/3072-setting. Batches of that size are necessary in the use cases mentioned in Section 1. In the server-only columns of Table 2, the 1000/3072-setting seems to be the only critical case with a running time of approximately 45 seconds. However, with better server hardware and by parallelizing the tasks onto different cores or multiple CPUs, the speed of the server computations can be increased arbitrarily.

	Server-Only			Browser-Only			Server Adv.		
ModExps	1024	2048	3072	1024	2048	3072	1024	2048	3072
50	0.09s	0.73s	2.26s	1.63s	11.02s	31.38s	18.45	15.19	13.87
100	0.18s	1.47s	4.48s	3.32s	22.14s	62.69s	18.89	15.02	13.98
500	0.88s	7.09s	22.57s	16.48s	103.19s	310.78s	18.71	14.55	13.77
1000	1.77s	14.26s	44.90s	33.04s	205.38s	626.62s	18.65	14.40	13.96

Table 2: Performance measurements of server-only (GMPLib) and browser-only (VJSC Library) modexp computations for different bit lengths. The last three columns show the relative advantage of server-only over browser-only computations.

4.2 Outsourcing Protocols

To evaluate the performance of the outsourcing protocols implemented in *famodulus*, we repeated the experiments from the previous subsection using the same batch sizes and bit lengths. We did the analysis for Algorithms 2 and 5, the two most efficient

[14] In all our experiments, we selected the smallest prime modulus p of the corresponding bit length. Base and exponent were picked at random from $\mathbb{Z}_p^*$ and $\mathbb{Z}_{p-1}$, respectively.

protocols from Table 1 with a single client-side modular multiplication each. All other algorithms of this paper require only two or four modular multiplications and are therefore not expected to perform much worse. Since Algorithm 5 requires each server to compute two modular exponentiations—the real one and the challenge—for each task in the batch, we expect a performance loss of up to 50% for each server running on a single core. This expectation gets confirmed by the measurement results shown in Table 3, especially for the batch size of 1000 modexps, where the relative overhead of both the client-side computations and the communication costs gets minimal in comparison with the costs of the necessary server-side computations. In all such cases, Algorithm 2 runs roughly 1.7 times faster than Algorithm 5.

The most interesting result of our experimental analysis is the performance of the outsourcing algorithms implemented in *famodulus* compared to browser-only computations. Relative values for 3072-bit parameters are shown in Table 3 (column 5 and 9). In case of Algorithm 2, the outsourcing protocol is approximately 13 times faster than browser-only computations. In comparison with the factor 14 obtained in the server-only setting for 3072 bits, we conclude that the overhead for the client and the communication is less than 8% of the total running time. In case of Algorithm 5, the outsourcing protocol is still between 7 and 8 times faster than client-only computations.

The absolute running times shown in Table 3 only get problematical for batch sizes of 500 modexps or more with 3072-bit parameters, for example approximately 80 seconds in the 1000/3072-setting of Algorithm 5. To obtain more acceptable running times in such extreme use cases, optimizations on the server side are mandatory. Such optimizations are also required to serve multiple users simultaneously. Nevertheless, we conclude from our experiments that even without such optimizations on the server side, the outsourcing protocols implemented in *famodulus* increase the overall computation time by approximately one order of magnitude.

	Algorithm 2			Adv.	Algorithm 5			Adv.
ModExps	1024	2048	3072	3072	1024	2048	3072	3072
50	0.16s	0.88s	2.49s	12.58	0.23s	1.40s	4.09s	7.68
100	0.29s	2.01s	4.86s	12.89	0.46s	2.78s	8.11s	7.73
500	1.36s	8.11s	24.30s	12.79	2.17s	13.32s	40.70s	7.64
1000	2.70s	16.21s	48.21s	13.00	4.27s	26.59s	80.54s	7.78

Table 3: Performance measurements of outsourcing modexp computations using Algorithms 2 and 5 for different bit lengths. Columns 5 and 9 show the relative advantage of the outsourcing protocols over browser-only computations for 3072-bit parameters.

5 Conclusion

In this paper, we presented our results from studying and implementing secure outsourcing protocols for modular exponentiations in the context of cryptographic web applications. The first conclusion is derived from the theoretical performance analysis of our protocols compared to existing protocols in the literature. In Table 1, by giving a summary of the relevant client-side operations, we have demonstrated that our protocols are much more efficient than comparable two-server protocols from the literature. For a secret base and a secret exponent, the advantage of the protocols from [7,2] compared to Algorithm 6 is the reduction from four to two servers, but the assumptions

of non-colluding servers and secure communication channels remain. In the light of the reduced workload on the client side in Algorithm 6, which consists of computing only four modular multiplications instead of several multiplicative inverses and random pairs $(r, g^r \bmod p)$, we are convinced that reducing the number of servers from four to two is not worthwhile in most settings. Similar conclusions can be drawn by comparing the client-side workload of our protocols with the one-server protocols from [3]. Their advantage, however, are the weaker underlying trust assumptions, which result from the public nature of the parameters sent to the server. Implementing these protocols, measuring corresponding running times, and comparing them to the results from this paper is left for future work.

The second conclusion of this paper results from the experimental performance analysis of our protocols in Section 4. In Section 1 we mentioned two use cases in the context of cryptographic voting protocols, in which a large amount of modexps need to be computed in the web browser. With our outsourcing protocols, we managed to reduce unacceptable in-browser running times by an order of magnitude. By optimizing or upgrading the server performance, further improvements of the overall running times are possible. We see at least three different approaches for server-side optimizations. The first is to execute the computations on high-performance server hardware, the second is to distribute the workload to all CPU cores or to a CPU cluster, and the third is to perform server-side precomputations for fixed-base or fixed-exponent modexps. The possibility of conducting modexp computations in parallel makes our whole approach highly scalable. Especially in scenarios with limited battery power (e.g., mobile devices), we consider this an important property. High scalability remains an important advantage even if client-side performance is further improved with new technologies such as WebAssembly. Setting up corresponding server infrastructure and conducting an experimental performance analysis for such a configuration is another topic left for future work.

References

1. Cavallo, B., Di Crescenzo, G., , Kahrobaei, D., Shpilrain, V.: Efficient and secure delegation of group exponentiation to a single server. RFIDsec'15, 11th International Workshop on Radio Frequency Identification. pp. 156–173. LNCS 9440, New York, USA (2015)
2. Chen, X., Li, J., Ma, J., Tang, Q., Lou, W.: New algorithms for secure outsourcing of modular exponentiations. IEEE Transactions on Parallel and Distributed Systems 25(9), 2386–2396 (2014)
3. Chevalier, C., Laguillaumie, F., Vergnaud, D.: Privately outsourcing exponentiation to a single server: Cryptanalysis and optimal constructions. ESORICS'16, 21st European Conference on Research in Computer Security. pp. 261–278. LNCS 9878, Heraklion, Greece (2016)
4. Fielding, R.T.: Architectural Styles and the Design of Network-Based Software Architectures. Ph.D. thesis, University of California, Irvine, USA (2000)
5. Galindo, D., Guasch, S., Puiggalí, J.: 2015 Neuchâtel's cast-as-intended verification mechanism. VoteID'15, 5th International Conference on E-Voting and Identity. pp. 3–18. LNCS 9269, Bern, Switzerland (2015)
6. Haenni, R., Koenig, R.E., Dubuis, E.: Cast-as-intended verification in electronic elections based on oblivious transfer. E-Vote-ID'16, 12th International Joint Conference on Electronic Voting. pp. 277–296. LNCS 10141, Bregenz, Austria (2016)

7. Hohenberger, S., Lysyanskaya, A.: How to securely outsource cryptographic computations. TCC'05, 2nd Theory of Cryptography Conference. pp. 264–282. LNCS 3378, Cambridge, USA (2005)
8. Kiraz, M.S., Uzunkol, O.: Efficient and verifiable algorithms for secure outsourcing of cryptographic computations. International Journal of Information Security 15(5), 519–537 (2016)
9. Locher, P., Haenni, R.: Verifiable Internet elections with everlasting privacy and minimal trust. VoteID'15, 5th International Conference on E-Voting and Identity. pp. 74–91. LNCS 9269, Bern, Switzerland (2015)
10. Ma, X., Li, J., Zhang, F.: Outsourcing computation of modular exponentiations in cloud computing. Cluster Computing 16(4), 787–796 (2013)
11. Mainini, P.: Efficient and Secure Outsourcing of Modular Exponentiation. Bachelor thesis, Bern University of Applied Sciences, Biel, Switzerland (2017)
12. Ye, J., Chen, X., Ma, J.: An improved algorithm for secure outsourcing of modular exponentiations. AINA'15, 29th International Conference on Advanced Information Networking and Applications Workshops. pp. 73–76. Gwangju, Korea (2015)