Outsourcing Modular Exponentiation in Cryptographic Web Applications

Pascal Mainini and Rolf Haenni
Bern University of Applied Sciences (BFH)

FC18, VOTING'18 Workshop, 2018-03-02

Agenda

- Introduction
- Our protocols
- Evaluation
- Q&A

Introduction

Motivation

- Cryptographic voting protocols[1] requiring thousands of modular exponentiations (*modexps*)
- Limited performance for modexp calculation in mobile devices and web browsers
- Possible applications in the IoT domain
- Outsourcing calculation as-is not feasible due to secrecy requirements

[1] E.g. Haenni/Koenig/Dubuis: Cast-as-intended in electronic elections based on oblivious transfer.

Group Aspects

We restrict ourselves to the multiplicative group $\mathbb{Z}_p^* = \{1, \ldots, p-1\}$ of integers modulo p, or corresponding subgroups $\langle x \rangle \subseteq \mathbb{Z}_p^*$ of known order generated by x, denoted as $\mathbb{G}_q$.

Operations in the exponent are computed in the additive group $\mathbb{Z}_q = \{0, \ldots, q-1\}$ of integers modulo q.

Our algorithms generalize naturally to arbitrary groups, however we do not treat other groups such as elliptic curves explicitly as they are not applicable to the targeted electronic voting protocols.

Basic Principles

We want to perform *modular exponentiation*:

$\exp(x, y) = x^y \bmod p$

Depending on the application, base and/or exponent must be kept secret. We assume the result to be secret in general.

Basic Principles

Most protocols in literature perform blinding based on the homomorphic property of the exponentiation function.

Secret base:

$$\exp(x_1 x_2, y) = (x_1 x_2)^y = x_1^y x_2^y = \exp(x_1, y)\exp(x_2, y)$$

Secret exponent:

$$\exp(x, y_1 + y_2) = x^{y_1+y_2} = x^{y_1} x^{y_2} = \exp(x, y_1)\exp(x, y_2)$$

Calculation of the respective, blinded modexps is performed on different servers.

Server Assumptions

All outsourcing protocols involving more than one server require the servers to be *non-colluding*. Trivially, colluding servers can uncover the blinded parameters.

We further distinguish:

- *Semi-honest* servers: execute the protocol faithfully and always return correct results
- *Malicious* servers: may return correct or arbitrary results at any time

Related Work

Large amount of literature is available and can be classified along two lines: Number of required servers (1 or 2) and adversary model (mainly semi-honest and malicious).

Comprehensive analysis and a compilation of protocols for 1 server is given in Chevalier et al., alternative approaches are presented by Cavallo et al. and Kiraz et. al.

The main reference for 2-server protocols is the paper by Hohenberger and Lysyanskaya. Chen et al. and Ye et al. proposed similar protocols with improved efficiency. We found an alternative approach based on the subset-sum-problem in a paper proposed by Ma et al.

Our Protocols

Our Protocols (Semi-Honest Servers)

Secret Base: Outsourced calculation is performed by two servers with blinded base:

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

Decomposition of x is achieved by randomly choosing x_1 and letting $x_2 = x\, x_1^{-1} \bmod p$. This approach has the disadvantage that the client must calculate the multiplicative inverse $x_1^{-1} \bmod p$.

A better solution, which we have not seen so far, is to have $x_2 = x\, x_1 \bmod p$, from which follows $x = x_1^{-1} x_2 \bmod p$. We obtain:

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

Our Protocols (Semi-Honest Servers)

Secret Exponent: Outsourced calculation is performed by two servers with blinded exponent:

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

Decomposition of y is achieved by randomly choosing y_1 and letting $y_2 = y - y_1 \bmod q$, which is already very efficient and only requires a single addition.

Our Protocols (Semi-Honest Servers)

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

$x_1 \in_R \mathbb{G}_q$, $x_2 \leftarrow x\, x_1 \bmod p$;
$z_1 \leftarrow S_1.\mathrm{ModExp}(x_1, -y \bmod q, p)$;
$z_2 \leftarrow S_2.\mathrm{ModExp}(x_2, y, p)$;
return $z_1 z_2 \bmod p$

Our Protocols (Malicious Servers)

To detect malicious servers, the general approach in literature is to challenge each server with additional modexps. Checking is performed differently depending on the case:

For *secret base*, exponents must be the same. We check with identical bases for each server, product of results must be 1.

For *secret exponent*, base must be the same. We check with identical modexps for both servers, results must be the same.

With one challenge modexp, we obtain so-called β-*checkability* of $1/2$. This may be arbitrarily increased by sending more challenges, leading to a general β-checkability of $\beta = \frac{c}{c+1}$ for $c \geq 0$ challenges.

Our Protocols (Semi-Honest Servers)

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

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

Comparison of Protocols

Comparison of our algorithms for secret base / public exponent (1), public base / secret exponent (2) and the respective checke versions for malicious servers (C):

Paper	Protocol Name	Secret Base	Secret Exp.	Number of Servers	Number of ModExps	Number of Mult.	Number of Inv.	Number of Rand.	β
1	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	Alg. 1	yes	no	2	1	2	–	–	0
	Alg. 2	no	yes	2	1	1	–	–	0
	Alg. 1 (C)	yes	no	2	2	2	–	–	1/2
	Alg. 2 (C)	no	yes	2	2	1	–	–	1/2
2	*Exp*	yes	yes	2	4	9	5	6	1/2
3	**Exp**	yes	yes	2	3	7	3	5	2/3

Papers: 1) Chevalier et al., 2) Hohenberger and Lysyanskaya, 3) Chen et al.

Evaluation

Implementation

In order to support research in mentioned use cases, we provide *famodulus*, a PoC implementation:[2]

- Exponentiation client in JavaScript
- Exponentiation server in Java, using GMPLib
- A demonstrator application enabling measurements

[2] `https://github.com/mainini/famodulus`

Performance Analysis

Time for calculating modexps with different sizes (base / exponent / modulus) with GMPLib in our server implementation compared to browser-only calculation (identical hardware):

	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

Last column shows the advantage (factor) of the server compared to the browser.

Performance Analysis

Measurements for Algorithm 2 for different bitsizes. The last column shows the relative advantage over browser-only calculations for the same parameters.

	Algorithm 2			Adv.
ModExps	1024	2048	3072	3072
50	0.16s	0.88s	2.49s	12.58
100	0.29s	2.01s	4.86s	12.89
500	1.36s	8.11s	24.30s	12.79
1000	2.70s	16.21s	48.21s	13.00

	Algorithm 2 (Checked)			Adv.
ModExps	1024	2048	3072	3072
50	0.23s	1.40s	4.09s	7.68
100	0.46s	2.78s	8.11s	7.73
500	2.17s	13.32s	40.70s	7.64
1000	4.27s	26.59s	80.54s	7.78

Thanks! Questions?