Efficient and Secure Outsourcing of Modular Exponentiation

Bachelor Thesis

Degree course:	Bachelor of Science BFH in Information Technology
Author:	Pascal Mainini
Advisor:	Prof. Dr. Rolf Haenni
Constituent:	BFH, Research Institute for Security in the Information Society, E-Voting Group
Experts:	Dr. Federico Flueckiger
Date:	2017-01-19

Versions

Version	Date	Status	Remarks
0.1	2016-10-25	Draft	Initial version
0.2	2016-11-03	Draft	Chapter Outsourced Modular Exponentiations, revision 1
0.3	2016-11-14	Draft	Chapter Outsourced Modular Exponentiations, revision 2
0.4	2016-11-25	Draft	Chapter Outsourced Modular Exponentiations, revision 3
0.5	2017-01-17	Draft	Main chapters complete
1.0	2017-01-19	Stable	Final version

Abstract

Many modern cryptographic techniques make extensive usage of modular exponentiation as underlying mathematical operation. Due to security requirements, they generally operate on very large numbers, making them impractical on typical devices of today, ranging from microcontrollers to smartphones. We present a thorough analysis of literature for outsourcing such calculations and provide a full implementation showcasing the technology and enabling benchmarks.

Modular exponentiation, called modexp for short, is the operation $x^y \bmod n$; or simply the remainder of an exponentiation modulo a given number. Even though efficient algorithms for calculating modexps exist, they remain computationally expensive operations. Applications become rapidly impractical, when a large number of modexps has to be calculated, possibly in a web browser, on a smartphone or an embedded system.

The E-Voting group of Bern University of Applied Sciences has developed two protocols for electronic voting which make extensive use of modexps (up to 300'000 per voter, with exponents ranging from 1024 bits upwards). While electronic voting is by far not the only application requiring such vast calculations, these two protocols are the primary motivation for our efforts in this thesis. They serve as use cases and as validation for our work.

One possible solution for the calculation of modexps on limited devices is the outsourcing to one or multiple, computationally strong peers. While sounding straightforward at first sight, problems emerge with a closer look: In the cryptographic context, at least some of the involved numbers have to be kept secret and it has to be ensured, that the result obtained from the server is correct (of course, without repeating the calculation).

Obvious and simple solutions for these problems can be found using only basic arithmetic; in general, however, these are not optimal. To find better algorithms, we have conducted a thorough analysis of the current state of the art and present unified algorithm descriptions and a comparative overview.

For our use cases, practical application and benchmarking of modexp outsourcing, are very important objectives. We have thus developed a fully working system for outsourcing modular exponentiations, with the properties required by the use cases, which enables direct performance comparison between outsourced and locally calculated modexps. We release the full implementation of this system, consisting of three self-contained components, under the permissive MIT open-source license.

Contents

1. Introduction

Dear reader,

you are holding in your hand, or more probably are looking at on your screen, my Bachelor's thesis in computer science.

This document, as well as the other parts of my work, are the result of countless hours spent during the last four months. Countless hours, investigating issues in the practical application of what looks just like one of the most simple mathematical formulas one could think of. To me, these hours where a fantastic journey into the unknown; I hope that this document is able to convey part of the fascination and some of the insights earned, while living through this journey.

Let us start the voyage with a quick look at the itinerary.

1.1. Document Structure

The first chapter, which follows directly after this introduction, will present the landscape of the journey. It introduces two use cases which were the motivation for research in this particular domain. It will also present our main contributions, the new areas in that landscape that have been mapped with our work.

Chapter 3 presents the timetable, our project plan, and gives an account of the steps that we went through on our way.

In Chapter 4, we introduce the first of our major two contributions: an in-depth review of relevant literature, with carefully unified algorithms in pseudo-code and a comparative summary.

Chapter 5 then presents our second contribution, the software components which have been developed based on the insights gained in Chapter 4.

There would be no interesting journey without at least a single day with some clouds on the horizon: Chapter 6 presents some unexpected results in the running time performance of our implementation.

Finally, already on the way back home, we revisit our impressions and conclude our work in Chapter 7.

Following that, in Appendix A, we provide technical details for installing our software, and Appendix B gives a short account of relevant decisions taken during the meetings with the advisor.

1.2. Acknowledgements

For me personally, this work is a very important and completely unforeseen milestone in my life. I would like to wholeheartedly thank, amongst many others which I'll unwittingly forget to mention here, the following people for supporting me and making this possible:

My parents, for all the freedom and support, Sheila for her love and endless patience. Peter Zankl, long-term mentor and engineering inspiration, Guy Zürcher for providing the starting point in my professional life, and Urs Sauter, for somewhat forcing me into studies. Furthermore, Annett Laube-Rosenpflanzer and Gerhard Hassenstein for being very kind and supportive superiors. For initial motivation, four years ago, Andreas Mosimann, and for picking up the phone late night, Laura Kistler.

During my work on this thesis, Thomas Bergwinkl and Reto Koenig contributed very valuable input. Rolf Haenni invested a large amount of time in deep, insightful discussions with me, for which I am deeply grateful.

Special thanks go to the fine people of my class: without all those endless hours of utter confusion and yellow cables, I would probably not have made it. Immer Stop.

If I have seen a little further, it is by standing on the shoulders of giants. – Isaac Newton

2. Motivation and Contribution

Today, many modern cryptographic techniques make extensive usage of modular exponentiation (introduced in detail in Chapter 4 , p. 9) as underlying mathematical operation. Most often, due to security requirements, they operate on very large numbers, making them impractical on many typical devices found nowadays, ranging from microcontrollers to smartphones.

While being broadly used, one particular application domain has motivated the research and development in this thesis: applications in modern, cryptographic protocols for electronic voting (e-voting), which is becoming an increasingly important topic in many countries. In Switzerland, for instance, strong focus on e-voting has just recently been reconfirmed as an important part of the federal electronic government strategy [31]. E-voting is a broad topic on its own which we do not further cover in this thesis; the interested reader is pointed out to Essex and Jonkers [23, 36] which provide a good overview of e-voting in the introductory chapters of their theses, and to Caarls [24] which outlines considerations in practical deployment of e-voting systems.

We would like to point out, that many societal and technical challenges remain open from our point of view for the time being. For this, we personally do not endorse e-voting. However, we understand the rising demand in electronic government, which includes e-voting, and we do support fundamental research in these areas.

2.1. Use Cases Motivating our Work

Two modern protocols in e-voting have recently been developed at the *E-Voting Group*[1] of BFH's Research Institute for Security in the Information Society (RISIS), both of them make extensive usage of modular exponentiation.

First, a protocol for so-called cast-as-intended verification in remote electronic voting by Haenni, Koenig and Dubuis [34], requires the calculation of up to 400 modular exponentiations with a modulus size starting at 2048 bits. Second, another protocol by Locher and Haenni [41], presents a new cryptographic Internet voting protocol with everlasting privacy. Here, the requirements are at about 300'000 modular exponentiations with a modulus size starting at 1024 bits. These numbers apply to calculations which have to be carried out on the side of the voter, for which a computationally weak environment has to be assumed.

Work on these protocols will, in the future, validate our research in this thesis. The protocols themselves will in turn be used in a proof of concept for the e-voting solution developed by the canton of Geneva. This solution is targeting e-voting in multiple cantons of Switzerland.

[1] `https://www.ti.bfh.ch/de/forschung/research_institute_for_security_in_the_information_society/e_voting_group.html`

2.2. Our Contributions

In this thesis, we provide two main contributions for *outsourced modular exponentiation*:

- A thorough analysis of state of the art outsourced modular exponentiation leading to a comparative overview of various protocols
- The implementation of a fully working system for research in outsourced modular exponentiations

In Chapter 4 (p. 9) we present our analysis of state of the art publications of protocols for outsourcing modular exponentiations. Outsourcing is considered to be a possible solution for supporting calculation on computationally weak devices. To improve understanding and comparability, we have unified all algorithm descriptions provided by the publications in pseudo-code. We conclude our analysis with a comparative overview of all protocols depending on possible requirements like, for instance, restrictions on the number of possible servers.

While researching outsourcing protocols, we have found additional mathematical problems, namely in randomization and calculation of modular inverses, which may be interesting for outsourcing as well. For these, descriptions and pseudo-code representations are also provided in the same chapter.

Based on our research, we have then implemented a complete and working system for outsourcing modular exponentiation, fulfilling the requirements of the use cases. This system consists of three self-contained components, which cover specific aspects of the problem:

- A stand-alone server, implemented in Java, offering a RESTful interface to clients for submitting calculations
- A JavaScript library for client-side development, targeted at researchers and application developers
- A demonstrator application which can be used for testing the functionality of the former components and for benchmarking

The whole system, which we release under the permissive MIT open-source license [19], is described in full detail in Chapter 5 (p. 29).

3. Objectives and Project Organization

Having presented the motivation for our work and the environment, in which it takes place, this chapter first addresses main objectives of the developed solution. The second part of the chapter then briefly presents the project plan and an account of the work performed during the project.

3.1. Objectives

In an initial meeting together with the advisor, we have defined the main objectives which should be accomplished by the research and development conducted in this thesis.[1] These objectives are:

- Implementation of a demonstrator application including a server and corresponding client library for JavaScript.
- Definition of a RESTful interface for communication between client and server.
- The implementation should be efficient, using state of the art algorithms where possible.
- Installation and usage of the demonstrator application should be easy and straightforward.
- Code quality requires special attention, the system will serve for further research and possibly as cryptographic library.

We will revisit these objectives during the following chapters; especially in Chapter 5 (p. 29) which describes the implemented demonstrator application, as well as in Appendix A (p. 63) which documents its installation and usage. In our final conclusions given in Chapter 7 (p. 47), we assess the fulfillment of the objectives.

3.2. Project Plan and Activity

Work on this thesis mainly occurred during three distinct phases which followed each other. Within those phases, different tasks where accomplished, mostly in sequence with some overlap and with the work on this document being done in parallel. Figure 3.1 shows our project plan and details phases and tasks.

The project officially started at the beginning of the semester in an initial kick-off meeting with the advisor on September 19, 2016.

After this, project setup took place and the research phase started. In this phase, which lasted up to the end of October (with some additional work being done also in November), thorough research of current state of the art has been conducted, which led to the results presented in Chapter 4 (p. 9). During this time, four meetings with the advisor took place (see Appendix B, p. 73 for a summary of all meetings).

[1] Meeting held 2016-09-19.

Efficient and Secure Outsourcing of Modular Exponentiation

Bachelorthesis Pascal Mainini – Fall Semester 2016

Week Semester	1	2	3	4	5	6	7	8	9	10	11	12	13	14	UF	UF	15	16	UF	PR	PR
Week Calendar	38	39	40	41	42	43	44	45	46	47	48	49	50	51	52	1	2	3	4	5	6
Date (Monday)	19.09.16	26.09.16	03.10.16	10.10.16	17.10.16	24.10.16	31.10.16	07.11.16	14.11.16	21.11.16	28.11.16	05.12.16	12.12.16	19.12.16	26.12.16	02.01.17	09.01.17	16.01.17	23.01.17	30.01.17	06.02.17
Project Phase	**Research**							**Development**										**Delivery**			
Initialization	**K**		**S1**	**S2**	**H**																
Project Setup																					
Research																					
Specification						**S3**		**S4 / M1**													
Specification REST-API																					
Software Design + Technology																					
Development								**E1**		**S5**			**S6**		**H**	**H**					
Implementation Server																					
Implementation Client-Library																					
Testing																					
Delivery																	**S7 / M2**	**M3**			
Packaging																					
Deployment																					
Documentation																		**M4**			
Report																					
Conclusion																					
Presentation																		**P: 20.1.17**			
Defense																				**D: 31.1.17**	

Updated: 03.01.2017

Legend

K = Kickoff
S = Status Meeting
M = Milestone
H = Holidays
P = Presentation / Final Day
D = Defense

Meetings

S1	Discussion of Algorithms
S2	Discussion of Algorithms
S3	Algorithm Choice, Review Report Structure
S4	Review REST-API, Software Design + Technology
S5	Intermediate Development Status Meeting
S6	Intermediate Development Status Meeting
S7	Final Development Status Meeting
E1	Initial Expert Meeting

Milestones

M1	REST-API and Software-Architecture specifed, Technology defined
M2	Implementation finished and tested
M3	Running Server(s) deployed
M4	Final Version of Report

Figure 3.1.: Project plan

In November, software development started with some initial work on the server side. After this, work on all three components (server, client and demonstrator application) took place more or less in parallel. A first milestone was reached just before Christmas: the initial release of all components as a fully working system, which supported DEC2 as initial outsourcing protocol (Protocol 4.1.2, described on page 13).

During Christmas break, intensive refactoring of the components for usability and performance took place, initial benchmarks where conducted early in January. During those benchmarks, a significant discrepancy between expected performance values for the DEC2 protocol and values effectively measured in the system under test was noticed.

The implementation of further protocols at this point would not have made sense without prior investigation of the noticed performance degradation. The root cause for the bad performance has been determined in the following two weeks, its analysis is presented in Chapter 6 (p. 39).

4. Outsourced Modular Exponentiations

In this chapter, we introduce the concept of *outsourced modular exponentiations* and the difficulties in implementing them. A first section gives a short overview of the required, mathematical background. It is followed by the description of an intuitive protocol we have devised, which highlights key concepts used in state of the art protocols for secure outsourcing modular exponentiations. These are then introduced in subsequent sections and we will also take a look at supporting algorithms which may further improve performance of our applications or simplify their implementation. Finally, we conclude this chapter with an overview of related work and the corresponding scenarios of application.

4.1. Introducing Outsourced Modular Exponentiations

Mathematically, a modular exponentiation c, generally abbreviated as *modexp*, is defined as

$$c \equiv b^e \pmod{m}$$

with b being the base, e the exponent and m the modulus. Thus, c is the remainder of the e-th power of b divided by m. In this thesis, we use the notation used in the majority of the consulted literature and write $u^a \bmod p$ for a modular exponentiation of base u to the power of a modulo either a prime modulus p or keeping m if the modulus is not prime.

Modular exponentiation is a computationally expensive operation. Assuming a naive algorithm as used for 'classic' exponentiation:

$$b^e = b \cdot b \cdot b \cdots b \ (e \text{ times})$$

one quickly sees that the number of multiplications corresponds to the exponent. As nowadays, cryptographic applications in general work on very large exponents (up to the order of 1200 digits), using such an algorithm is not applicable in practice.

For modular exponentiation, there are well known algorithms (see [43, Chapter 14.6, p. 613] and [37, Appendix B, p. 553] for reference) which reduce the amount of required multiplications to the order of $O(\log_2 e)$ in average, which still can mean 2048-4096 modular multiplications in practice. Additionally, often not a single modular exponentiation, but many (possibly with changing exponent or base) are required by a cryptographic application; further increasing computational complexity.

Cryptographic primitives (algorithms) take place in specific settings, more precisely they work in specific mathematical groups. Giving an introduction to cryptography or group theory would go beyond the scope of this document. Instead, in the following, we only shortly characterize groups of integers modulo p as far as is needed to support understanding of the presented algorithms and protocols. For further reading, Katz and Lindell [37] (especially Chapter 8, p. 285) provide a modern introduction to cryptography; Lang [39] and Childs [29] may serve as additional literature for underlying algebra and group theory.

4.1.1. Background: The Multiplicative Group $\mathbb{Z}_p^*$

Today, most of cryptography takes place in either groups of integers modulo p or in elliptic curve groups; the latter gaining popularity nowadays due to computational advantages. *By decision, all work relevant to this thesis takes place only in multiplicative groups of integers modulo prime p which we denote as* $\mathbb{Z}_p^*$.[1] Note also, that in our case, p is in general a very large prime number, meeting the required cryptographic security level (e.g. as of today in the range of 2048-4096 bits).

A group is defined mathematically as a set with a binary operation $\circ$ operating on that set. Furthermore, some additional constraints must be fulfilled, in the case of $\mathbb{Z}_p^*$ these are:[2,3]

- *Closure*: For all $x, y \in \mathbb{Z}_p^*$, $x \circ y \in \mathbb{Z}_p^*$ (result of the operation lies again in the group).
- *Existence of an identity*: For all $x \in \mathbb{Z}_p^*$, there is a $e \in \mathbb{Z}_p^*$ such that $x \circ e = e \circ x = x$.
- *Existence of an inverse*: For all $x \in \mathbb{Z}_p^*$, there is a $y \in \mathbb{Z}_p^*$ such that $x \circ y = y \circ x = e$.
- *Associativity holds*: For all $x_1, x_2, x_3 \in \mathbb{Z}_p^*$, $(x_1 \circ x_2) \circ x_3 = x_1 \circ (x_2 \circ x_3)$ (order is irrelevant).

For $\mathbb{Z}_p^*$, the set consists of the integers $\{1, \ldots, p-1\}$ and the operation is the multiplication ($\times$); furthermore, the identity element is 1 and the inverse for each element x is denoted as x^{-1}. Inverses are not intuitive (as for the addition), but they can be found efficiently using the extended Euclidean algorithm (see [37, p. 551]).

Theoretically, we could also work in an additive group with the addition ($+$) as operation; such groups however do not provide any computationally hard problems and are thus not interesting for cryptographic applications. By using the multiplicative group, we find the problem of *discrete logarithm* (see [37, p. 319]), for which no efficient solution has been found so far and which is widely used in cryptography today.

Another important property of a group is its *order*, which corresponds to the number of elements in the group. For multiplicative groups over the integers in general, the order can be calculated using Euler's $\phi()$ function; for groups with prime modulus, $|\mathbb{Z}_p^*| = \phi(p) = p - 1$ holds for all moduli. *We often denote the order of a group as q.*

Note that we obtain the identity element of a group by exponentiating every member of the group with the group order: $x \in \mathbb{Z}_p^* : x^q = x^{p-1} = 1$.

As last important part, *generators* of groups have to be introduced. A generator g is an element of the group which 'generates' (not necessarily in order) all elements of the group by exponentiation up to a certain exponent. For $\mathbb{Z}_p^*$, exponentiation up to q is sufficient as higher exponents lead to cyclic repetitions of the group members and restricting exponents up to q provides additional benefits not detailed here. Thus, for a given generator $g \in \mathbb{Z}_p^*$: $\mathbb{Z}_p^* = \{1, g, g^2, \ldots, g^{q-1}\}$.

4.1.2. Issues in Outsourcing

After introducing modular exponentiation, which looks simple enough, where exactly lies the challenge of outsourcing such computations to a computationally strong server as required by our applications described in Chapter 2? Would not simply sending the values for u, a and p to the server and having it return its result to the client already solve the issues of the latter? For most

[1] Meeting with advisor, 27th October 2016.
[2] A binary operation is an operation taking two operands.
[3] See [37, Groups, p. 291] for more details.

cryptographic applications, unfortunately, such a simple solution is not applicable: Usually, some of the values must be kept secret and thus be known only by the client.

Introducing variables x, y and z for secret base, exponent and result, we address three outsourced modular exponentiations which are required by our applications:

Table 4.1.: Relevant outsourced modular exponentiations

Modexp	Base	Exponent	Result
$u^y = z$	Public	Secret	Secret
$x^a = z$	Secret	Public	Secret
$x^y = z$	Secret	Secret	Secret

Of course, $x^y = z$ also covers the two other cases; there are however significant performance gains if not all parameters need to be kept secret and specially adapted protocols can be used for these. For the same reason, some protocols make further distinction in cases with fixed or variable base.

As can be seen in Table 4.1, a protocol for outsourcing modular exponentiation must hide (or more specifically *blind*) either the base, the exponent or both from the server performing the modexp and the server must not be able to learn anything about the result. Depending on the requirements of the application however, this is not the only issue to be solved:

We assume two types of servers: *honest but curious* (or *semi-honest*) and malicious servers. A semi-honest server follows the protocol and returns correct results, however possibly snooping on the exchanged values. Opposed to that, a malicious server interacts with the values and may return manipulated results.[4] If the application has no way to detect such malicious behavior, the protocol needs to ensure that it cannot occur or only with insignificant probability.[5] This requirement is called *checkability*; protocols which support it are in general more complex and less performant.

4.1.3. Basic Decomposition Protocols (DEC)

We now describe an intuitive approach for solving the given issues without any formal security proof. For this, we have devised three simple protocols subsumed as *Basic Decomposition* or *DEC*. These are straightforward to implement, and serve well as illustration for techniques used in protocols found in the literature, which in turn are described in section 4.2.

Basic Algorithm

It can easily be seen, that a given number $n \in \mathbb{Z}_p^*$ can be blinded using only basic arithmetics and a random number generator: We can write $x = n/r$ where $r \in \mathbb{Z}_p^*$ is chosen at random.[6] Assuming that an adversary does not know r, n cannot be deduced from x. This is the basic idea behind blinding.

[4] Not necessarily maliciously, this could also occur by malfunction of the software or other errors.

[5] Detection is not trivial, as the client would have to perform the calculation itself for cross-checking, leading back to the initial problem. However applications may distribute calculations to multiple peers for instance, and thus discover errors on their own.

[6] Here, the division is an operation of the underlying group. For $\mathbb{Z}_p^*$, this corresponds to a multiplication with the inverse!

In order to fully hide the (modular) exponentiation u^a, both u and a need to be blinded. While u lies in multiplicative group $\mathbb{Z}_p^*$ and can be blinded as described, a is in additive group $\mathbb{Z}_q$ and has to be blinded by addition of a random variable. We proceed as follows:

First, using randomly chosen $v \in \mathbb{Z}_p^*$, we divide the base into two parts v and w:

$$u^a = (vw)^a \text{ , where } w = u/v.$$

Second, using randomly chosen $b \in \mathbb{Z}_q$, the exponent is split into parts b and c using a similar approach:

$$(vw)^a = (vw)^{b+c} \text{ , where } c = a - b.$$

After splitting up, the modular exponentiation can be written as follows:

$$u^a = (vw)^{b+c} = v^b v^c w^b w^c.$$

We have now created four individual and randomized modular exponentiations v^b, v^c, w^b and w^c! These can be transmitted to four different servers which perform the calculation. After obtaining the results, we simply have to multiply them to obtain the result for u^a:

$$v^b v^c w^b w^c = (vw)^{b+c} = u^a.$$

There are two important restrictions to this approach:

First, it has to be ensured that the servers are *non-colluding*. If any two of them cooperate, they can trivially reconstruct either the base or the exponent, depending on the parts they have obtained.

Furthermore, communication to the servers must occur over a secure channel.[7] If the parts would be transmitted without protection, any outside adversary or possibly any of the servers could intercept the parts and reconstruct the base and/or the exponent.

Having introduced the basic approach mathematically, we now give protocols in pseudo-code for implementation. We denote the calculation of a modexp $u^a \bmod p$ on a remote server as a call to the function $\mathbf{S}_i(u, a, p)$, with i being the number of the server invoked. Furthermore, we use $\in_R$ to indicate uniformly picking at random an element of the specified set.

[7] See Glossary (page 51) for a description of secure channels.

The first protocol implements secret base and result exponentiation $x^a = z$:

Protocol: DEC1(u, a, p)
Input: Base $u \in \mathbb{Z}_p^*$, exponent $a \in \mathbb{Z}_q$, prime modulus p
Output: $u^a \in \mathbb{Z}_p^*$

$v \in_R \mathbb{Z}_p^*$
$w \leftarrow u/v \mod p$

$s_1 \leftarrow \mathbf{S}_1(v, a, p)$
$s_2 \leftarrow \mathbf{S}_2(w, a, p)$

return $s_1 \cdot s_2 \mod p$

Protocol 4.1.1: DEC1: Outsourced modular exponentiation $x^a = z$

The next protocol hides the exponent and the result ($u^y = z$), while working with a public base:

Protocol: DEC2(u, a, p)
Input: Base $u \in \mathbb{Z}_p^*$, exponent $a \in \mathbb{Z}_q$, prime modulus p
Output: $u^a \in \mathbb{Z}_p^*$

$b \in_R \mathbb{Z}_q$
$c \leftarrow a - b \mod q$

$s_1 \leftarrow \mathbf{S}_1(u, b, p)$
$s_2 \leftarrow \mathbf{S}_2(u, c, p)$

return $s_1 \cdot s_2 \mod p$

Protocol 4.1.2: DEC2: Outsourced modular exponentiation $u^y = z$

Finally, the third protocol hides base, exponent and modulus ($x^y = z$):

Protocol: DEC3(u, a, p)
Input: Base $u \in \mathbb{Z}_p^*$, exponent $a \in \mathbb{Z}_q$, prime modulus p
Output: $u^a \in \mathbb{Z}_p^*$

$v \in_R \mathbb{Z}_p^*$
$w \leftarrow u/v \mod p$
$b \in_R \mathbb{Z}_q$
$c \leftarrow a - b \mod q$

$s_1 \leftarrow \mathbf{S}_1(v, b, p)$
$s_2 \leftarrow \mathbf{S}_2(v, c, p)$
$s_3 \leftarrow \mathbf{S}_3(w, b, p)$
$s_4 \leftarrow \mathbf{S}_4(w, c, p)$

return $s_1 \cdot s_2 \cdot s_3 \cdot s_4 \mod p$

Protocol 4.1.3: DEC3: Outsourced modular exponentiation $x^y = z$

Adding Checkability

The previously presented basic protocols do not offer any security against wrong or manipulated results; any of the involved servers could return an incorrect result without the client noticing.

A possible solution to this problem is to test each server using a special *test query*, of which the result is known, without the server being aware that it is being tested. Servers returning incorrect results would be uncovered and the protocol could abort with an error. A simple implementation of such a test query would be choosing a random base and exponent and sending it to each of the four servers. If the client does not obtain the same result four times, something went wrong.

The following is a new version of DEC3 (protocol 4.1.3) with added checkability. A similar approach also works for DEC1 and DEC2, we omit the details here for brevity.

Protocol: DEC4(u, a, p)
Input: Base $u \in \mathbb{Z}_p^*$, exponent $a \in \mathbb{Z}_q$, prime modulus p
Output: $u^a \in \mathbb{Z}_p^*$

$v \in_R \mathbb{Z}_p^*$
$w \leftarrow u/v \mod p$
$b \in_R \mathbb{Z}_q$
$c \leftarrow a - b \mod q$

$u_t \in_R \mathbb{Z}_p^*$
$a_t \in_R \mathbb{Z}_q$

// Execute in random order!
$s_{11} \leftarrow \mathbf{S}_1(v, b, p)$
$s_{12} \leftarrow \mathbf{S}_1(u_t, a_t, p)$

// Execute in random order!
$s_{21} \leftarrow \mathbf{S}_2(v, c, p)$
$s_{22} \leftarrow \mathbf{S}_2(u_t, a_t, p)$

// Execute in random order!
$s_{31} \leftarrow \mathbf{S}_3(w, b, p)$
$s_{32} \leftarrow \mathbf{S}_3(u_t, a_t, p)$

// Execute in random order!
$s_{41} \leftarrow \mathbf{S}_4(w, c, p)$
$s_{42} \leftarrow \mathbf{S}_4(u_t, a_t, p)$

if $s_{12} = s_{22} = s_{32} = s_{42}$ **then**
    return $s_{11} \cdot s_{21} \cdot s_{31} \cdot s_{41} \mod p$
else
    return *ERROR*
end

Protocol 4.1.4: DEC4: Checkable DEC3

4.2. State of the Art Protocols and Further Related Work

In an initial phase of this thesis, we have extensively researched the scientific literature and on-line resources about the subject of outsourced modular exponentiations. We have done so to the best of knowledge to ensure the correctness of the protocols and to become aware of their security properties. As the motivation of our work lies in improving performance of the use cases, choosing efficient protocols was another important aspect of research.

As pointed out before, distributing to multiple servers may be a viable approach for securing calculations. This option was also chosen by some authors of the discussed protocols, while others rely on a single server for outsourcing. We have thus classified the protocols in two groups, one (**S**) applying to the single server case and the other (**M**) to multiple servers. When considering multiple servers, only cases with two or four (depending on the protocol) make sense; adding more servers is possible but it will improve neither security nor efficiency.

4.2.1. Hohenberger and Lysyanskaya (M1)

Contributions

In 2005, Hohenberger and Lysyanskaya [35] published their protocol for securely outsourcing modular exponentiations to two servers. Besides that, the paper made two important contributions: Providing thorough *security definitions* as well as introducing the *two untrusted program model*. With their work, the authors established standards for subsequent research in distributed, cryptographic protocols. For this reason, we provide a more in-depth description of relevant aspects of the paper in the following.

Briefly, security definitions introduced by the authors define a so-called (α, β)-*outsource-secure algorithm*, which could for instance be the protocol for outsourced modexps. For this algorithm, a definition of *outsource-security* is given. This security definition proves that any outside adversary as well as the program performing the computation cannot learn anything about the calculation. Furthermore, the definition of so-called α-*efficiency* mandates that the running time of such an algorithm has to be smaller or equal to an α-multiplicative factor of the original, un-outsourced algorithm. This clearly makes sense as outsourcing will not help in the case that the outsourced algorithm is asymptotically worse. Finally, the definition of β-*checkability* gives a precise definition of the aforementioned checkability. It mandates that a deviation from the intended function of an algorithm must be detected with a probability larger than a given β.

In the two untrusted program model, an adversary provides two implementations of the outsourced part of an algorithm. These implementations can be different, however this is not a requirement (e.g. in practice, two different vendors could be considered when buying the software). Even though the adversary controls the implementations, he is not allowed to directly control running instances of them. All communication must be made over the operator running the algorithm. The *one-malicious*-version of this model states that only one of the two implementations may deviate from the assumed functionality at any time but the operator does not know which of them.

Using the one-malicious, two untrusted program model, the paper proves that the β-checkability of the presented algorithm is $1/2$, meaning that erroneous or malicious runs of it can be detected in every second case on average. In general, this can be considered as enough security for applications which need to do a lot of outsourced modexps, as it is usually the case.

Protocol Discussion

At first sight, our protocol DEC4 (protocol 4.1.4) looks quite similar to the protocol given by the authors, however there are some important differences.

The protocol by the authors introduces a generator for so-called *blinding pairs* (r, g^r) where r is chosen at random on every invocation. We will subsequently refer to such a generator by the name of RandomPair() and assume it to be available and efficient.

Using such a generator leads to improved blinding, which now only requires two instead of four servers for outsourcing secret base *and* exponent exponentiations. Also, checkability is now ensured using test queries with a known, correct result. As each server receives two queries, one containing the required exponentiation and the other the test pair, incorrect behavior per server can be detected with a probability of $1/2$.

The protocol first starts by obtaining two blinding pairs $(r_{b_1}, g^{r_{b_1}})$ and $(r_{b_2}, g^{r_{b_2}})$ from the generator. Using these two pairs, a third *fully* randomized pair (v, v^b) is deduced:

$$v = g^{r_{b_1}};$$
$$v^b = g^{r_{b_2}} \text{ , where } b = r_{b_2}/r_{b_1};$$

The next two logical divisions split the base and exponent in a similar way as DEC4. Note that $d \in \mathbb{Z}_q$ and $f \in \mathbb{Z}_p^*$ are chosen randomly:

$$u^a = (vw)^a = v^a w^a = v^b v^c w^a \text{ , where } w = u/v \text{ and } c = a - b;$$
$$v^b v^c w^a = v^b (fh)^c w^{d+e} = v^b f^c h^c w^d w^e \text{ , where } h = v/f \text{ and } e = a - d.$$

With those divisions, the modexps w^d, w^e, f^c and h^d have been obtained. As opposed to DEC4, it is possible to create two pairs with different base and exponent, which don't leak any information about the given inputs: (w^d, f^c) and (w^e, h^c). Therefore, only two servers are required to perform the calculations.

However, to obtain a checkability of $1/2$, also two test modexps per server are required. These are generated by invoking RandomPair four times to obtain pairs $(r_{t_1}, g^{r_{t_1}})$, $(r_{t_2}, g^{r_{t_2}})$, $(r_{t_3}, g^{r_{t_3}})$ and $(r_{t_4}, g^{r_{t_4}})$. By similar combination as for (v, v^b), these form the two fully randomized test pairs $(r_{t_1}/r_{t_3}, g^{r_{t_3}})$ and $(r_{t_2}/r_{t_4}, g^{r_{t_4}})$.

Each server then calculates four modexps which he has obtained in *randomized* order: two of the modexp-pairs given above, as well as two test modexps. After obtaining the results for the outsourced modexps, the client simply needs to multiply them together again to obtain u^a:

$$v^b f^c h^c w^d w^e = v^{b+c} w^{d+e} = v^a w^a = (vw)^a = u^a.$$

It is important to note, that even if v^b looks like an overall randomization which is not transmitted, servers must not collude and secure channels are also required. Considering the final multiplication, $v^b f^c h^c w^d w^e = u^a$ where $e = a - d$, one notices that it is easy for any adversary obtaining the two modexps w^d and w^e to obtain a by simple addition: $a = d + e = d + (a - d)$. Similar reasoning can be applied for f^c and h^c to obtain u.

We give a pseudo-code representation of the protocol in the following:

Protocol: M1(u, a, p)
Input: Base $u \in \mathbb{Z}_p^*$, exponent $a \in \mathbb{Z}_q$, prime modulus p
Output: $u^a \in \mathbb{Z}_p^*$

$(r_{b_1}, g^{r_{b_1}}) \leftarrow$ RandomPair()
$(r_{b_2}, g^{r_{b_2}}) \leftarrow$ RandomPair()
$(r_{t_1}, g^{r_{t_1}}) \leftarrow$ RandomPair()
$(r_{t_2}, g^{r_{t_2}}) \leftarrow$ RandomPair()
$(r_{t_3}, g^{r_{t_3}}) \leftarrow$ RandomPair() // Attention: $r_{t_3} \in \mathbb{Z}_q^*$!
$(r_{t_4}, g^{r_{t_4}}) \leftarrow$ RandomPair() // Attention: $r_{t_4} \in \mathbb{Z}_q^*$!

$d \in_R \mathbb{Z}_q$
$f \in_R \mathbb{Z}_p^*$

$v \leftarrow g^{r_{b_1}}$
$b \leftarrow r_{b_2}/r_{b_1} \mod p$
$c \leftarrow a - b \mod q$
$e \leftarrow a - d \mod q$
$w \leftarrow u/v \mod p$
$h \leftarrow v/f \mod p$

// Order of the following queries must be randomized!
$s_{11} \leftarrow \mathbf{S}_1(w, d, p)$
$s_{12} \leftarrow \mathbf{S}_1(f, c, p)$
$s_{13} \leftarrow \mathbf{S}_1(g^{r_{t_3}}, r_{t_1}/r_{t_3} \mod q, p)$
$s_{14} \leftarrow \mathbf{S}_1(g^{r_{t_4}}, r_{t_2}/r_{t_4} \mod q, p)$

// Order of the following queries must be randomized!
$s_{21} \leftarrow \mathbf{S}_2(w, e, p)$
$s_{22} \leftarrow \mathbf{S}_2(h, c, p)$
$s_{23} \leftarrow \mathbf{S}_2(g^{r_{t_3}}, r_{t_1}/r_{t_3} \mod q, p)$
$s_{24} \leftarrow \mathbf{S}_2(g^{r_{t_4}}, r_{t_2}/r_{t_4} \mod q, p)$

if $s_{13} \neq g^{r_{t_1}}$ **or** $s_{14} \neq g^{r_{t_2}}$ **or** $s_{23} \neq g^{r_{t_1}}$ **or** $s_{24} \neq g^{r_{t_2}}$ **then**
    return *ERROR*
else
    return $g^{r_{b_2}} \cdot s_{11} \cdot s_{12} \cdot s_{21} \cdot s_{22} \mod p$
end

Protocol 4.2.1: M1: Hohenberger and Lysyanskaya from [35]

Summary

The authors have presented the first protocol for outsourcing modular multiplication to two servers of which only one has to act honestly. Both servers learn nothing out of this calculation and a malicious server can be detected with probability $1/2$. The protocol needs 9 modular multiplications

and 5 modular inversions as well as 6 invocations of RandomPair. Each server has to calculate 4 modular exponentiations.

4.2.2. Chen et al. (M2)

Contributions

In 2014, Chen et al. [26] published a paper containing an improved version of the protocol presented by Hohenberger and Lysyanskaya (4.2.1), based on the same security definitions as the original authors.

Additionally, their work also introduces a first protocol for outsourced, *simultaneous modular exponentiations*, however in a currently unpublished paper from 2016, Lin et al. [40] state that this protocol leaks sensitive data.[8] We did not thoroughly verify this claim as simultaneous modexps are not a primary requirement for this thesis. Furthermore, our implementation enables the client to transmit multiple but isolated modexps to the server, which is not resulting in the same performance gain, but can be tolerated for the our applications.

Protocol Discussion

The improved protocol for outsourcing modexps presented by Chen et al. looks quite similar to protocol M1 (4.2.1). After creating an initial, fully randomized blinding pair by using the same technique as M1, they propose slightly different logical splits:[9]

$$u^a = (vw)^a = g^{r_{b_1}a}w^a = g^{r_{b_2}}g^{\gamma}w^a = \underline{v^b g^{\gamma} w^a}, \text{ where } w = u/v \text{ and } \gamma = ar_{b_1} - r_{b_2};$$
$$v^b g^{\gamma} w^a = v^b g^{\gamma} w^{k+l} = v^b g^{\gamma} w^k w^l, \text{ where } l = a - k.$$

As can be seen immediately, the amount of randomized modexps is reduced from four to three (w^k, w^l and g^{γ}). By obtaining pairs $(r_{t_1}, g^{r_{t_1}})$, $(r_{t_2}, g^{r_{t_2}})$ from RandomPair, a fully randomized test pair $(r_{t_2}/r_{t_1}, g^{r_{t_1}})$ is deduced as before. With a third blinding pair $(r_{t_3}, g^{r_{t_3}})$, g^{γ} is blinded as $(\gamma/r_{t_3}, g^{r_{t_3}})$ and can then be used as a second test-pair which directly returns the result for the needed g^{γ}! The number of modexps sent to the servers is thus reduced to three and checkability is improved to $2/3$.

To obtain the result for the outsourced modexp, a similar multiplication as in M1 is performed:

$$v^b g^{\gamma} w^k w^l = g^{r_{b_2}} g^{\gamma} w^k w^l = v^b g^{\gamma} w^k w^l = u^a.$$

[8] A simultaneous modular exponentiation, used in many cryptographic applications, is a product of two modexps with different bases and exponents: ${u_1}^a {u_2}^b \bmod p$.

[9] The authors denote v^b as $g^{r_{b_2}}$. To ease comparison with protocol M1, we have added a representation using v^b.

We give a pseudo-code representation of the protocol in the following:

Protocol: M2(u, a, p)
Input: Base $u \in \mathbb{Z}_p^*$, exponent $a \in \mathbb{Z}_q$, prime modulus p
Output: $u^a \in \mathbb{Z}_p^*$

$(r_{b_1}, g^{r_{b_1}}) \leftarrow$ RandomPair()
$(r_{b_2}, g^{r_{b_2}}) \leftarrow$ RandomPair()
$(r_{t_1}, g^{r_{t_1}}) \leftarrow$ RandomPair() // Attention: $r_{t_1} \in \mathbb{Z}_q^*$!
$(r_{t_2}, g^{r_{t_2}}) \leftarrow$ RandomPair()
$(r_{t_3}, g^{r_{t_3}}) \leftarrow$ RandomPair() // Attention: $r_{t_3} \in \mathbb{Z}_q^*$!

$k \in_R \mathbb{Z}_q$

$v \leftarrow g^{r_{b_1}}$
$\gamma \leftarrow a \cdot r_{b_1} - r_{b_2} \mod q$
$l \leftarrow a - k \mod q$
$w \leftarrow u/v \mod p$

// Order of the following queries must be randomized!
$s_{11} \leftarrow \mathbf{S}_1(w, l, p)$
$s_{12} \leftarrow \mathbf{S}_1(g^{r_{t_3}}, \gamma/r_{t_3} \mod q, p)$
$s_{13} \leftarrow \mathbf{S}_1(g^{r_{t_1}}, r_{t_2}/r_{t_1} \mod q, p)$

// Order of the following queries must be randomized!
$s_{21} \leftarrow \mathbf{S}_2(w, k, p)$
$s_{22} \leftarrow \mathbf{S}_2(g^{r_{t_3}}, \gamma/r_{t_3} \mod q, p)$
$s_{23} \leftarrow \mathbf{S}_2(g^{r_{t_1}}, r_{t_2}/r_{t_1} \mod q, p)$

if $s_{12} = s_{22}$ **and** $s_{13} = s_{23} = g^{r_{t_2}}$ **then**
    return $g^{r_{b_2}} \cdot s_{11} \cdot s_{12} \cdot s_{21} \bmod p$
else
    return *ERROR*
end

Protocol 4.2.2: M2: Chen et al. from [26]

Summary

Chen et al. present an improved version of protocol M1, also for two servers. This protocol, adhering to the same security definitions, reduces the required modular multiplications from 9 to 7 and modular inverses from 5 to 3. Also, 5 instead of 6 blinding pairs are needed and each server calculates 3 instead of 4 modular exponentiations. Additionally, the β-checkability increases from $1/2$ to $2/3$.

4.2.3. Chevalier et al. (S1)

Contributions

Significant research regarding outsourcing exponentiation to a single server has been provided by Chevalier et al. in 2016 [27, 28]. The authors provide a taxonomy of outsourcing protocols for fixed and variable base for all combinations of secret base, exponent and result. For all protocols, except for the trivial and useless cases, pseudo-code descriptions and a complexity analysis is given. All protocols do not provide checkability. There is some overlap between the different protocols for variable base exponentiation; the authors provide versions with different computational complexity depending on the complexity of implementation. We further detail this point in the protocol discussion below.

Besides providing a valuable classification and analysis of protocols, the paper also presents an attack on a protocol given by Wang et al. [46], neither their protocol nor the attack by Chevalier et al. is relevant to this thesis.

Additionally, the full version ([28, Appendix C, p. 23]) provides protocols for simultaneous modular exponentiations, based on the protocols for single exponentiation given in the same paper. As already stated in section 4.2.2, simultaneous modexps are not a primary requirement for this thesis, we do not include a description of these protocols.

Protocol Discussion

In [27], not a single protocol is introduced, but a whole family of protocols depending on the required secrecy of the parameters and if the base is fixed or variable. We use the same notation as before, using $\mathbf{S}_1(u, a, p)$ as remote call to the (single) server to obtain $u^a \bmod p$ and assuming a function RandomPair() returning randomized pairs (r, g^r). Where required by the protocols, we will describe additional algorithms or provide references to the literature.

First, we start the discussion with protocols for fixed base and the given Protocol 1, which applies to the case $x^y = z$ (everything secret) and which is also optimal for $x^a = z$ (public exponent):

Protocol: S1P1(u, a, p)
Input: Base $u \in \mathbb{Z}_p^*$, exponent $a \in \mathbb{Z}_q$, prime modulus p
Output: $u^a \in \mathbb{Z}_p^*$

$(r_1, u^{r_1}) \leftarrow$ RandomPair() // Attention: $r_1 \in \mathbb{Z}_q^*$!
$(r_2, u^{r_2}) \leftarrow$ RandomPair()

$t \leftarrow (a - r_2)/r_1 \mod q$
$s_1 \leftarrow \mathbf{S}_1(u^{r_1} \mod p, t, p)$

return $s_1 \cdot u^{r_2} \bmod p$

Protocol 4.2.3: S1P1: Fixed base $x^y = z$ and $x^a = z$ outsourcing from [27]

As can be immediately seen, similar blinding techniques as in the previous protocols are applied but there is no checkability given. Another important observation is, that the base u has to be the same as the generator element returned as base by RandomPair(), hence the fixed base of the protocols. Protocol 2 and Protocol 3 provide simplifications of Protocol 1 for improved efficiency. Protocol 2

can be used in cases where the result or the base and the result can be public, Protocol 3 where the base alone can be public. *Note that the Protocol 3 given in the paper returns $h \cdot g^k$. This is a typo which got confirmed by the authors and which we have corrected in the given pseudo-code.*

Protocol: S1P2(u, a, p)
Input: Base $u \in \mathbb{Z}_p^*$, exponent $a \in \mathbb{Z}_q$, prime modulus p
Output: $u^a \in \mathbb{Z}_p^*$

$(r, u^r) \leftarrow$ RandomPair() // Attention: $r \in \mathbb{Z}_q^*$!
$s_1 \leftarrow \mathbf{S}_1(u^r, a/r \mod q, p)$

return s_1

Protocol 4.2.4: S1P2: Fixed base $x^y = n$ and $u^y = n$ outsourcing from [27]

Protocol: S1P3(u, a, p)
Input: Base $u \in \mathbb{Z}_p^*$, exponent $a \in \mathbb{Z}_q$, prime modulus p
Output: $u^a \in \mathbb{Z}_p^*$

$(r, u^r) \leftarrow$ RandomPair()
$s_1 \leftarrow \mathbf{S}_1(u, a - r \mod q, p)$

return $s_1 \cdot u^r \mod p$

Protocol 4.2.5: S1P3: Fixed base $u^y = z$ outsourcing from [27]

The first protocol for a (public) variable base, Protocol 5, can be used for $u^y = z$ and $u^y = n$ outsourcing. It is a configurable protocol, whose computational complexity depends on a parameter s, which, depending on the given value results in (besides other slight differences) more or less calls to $\mathbf{S}_1()$. For a better understanding of the protocol, we give Protocol 4 from [27], which corresponds to Protocol 5 with $s = 1$, leading to a single invocation of $\mathbf{S}_1()$:

Protocol: S1P4(u, a, p)
Input: Base $u \in \mathbb{Z}_p^*$, exponent $a \in \mathbb{Z}_q$, prime modulus p
Output: $u^a \in \mathbb{Z}_p^*$

$m \leftarrow \lceil \sqrt{p} \rceil$
$a_0 \leftarrow a \bmod m$
$a_1 \leftarrow a \operatorname{div} m$ // Euclidean division: $a = a_1 \cdot m + a_0$

$s_1 \leftarrow \mathbf{S}_1(u, m, p)$

return $u^{a_0} s_1^{a_1} \mod p$ // Using TwoWary()

Protocol 4.2.6: S1P4: Variable base $u^y = z$ and $u^y = n$ with $s = 1$ outsourcing from [27]

The main point here is a (configurable) split of the exponent into parts with about the same amount of bits (achieved with $\lceil \sqrt{p} \rceil$); half of the calculation is performed by the server and the other half by

the client. By calculating the multi-modexp $u^{a_0}h^{a_1}$, the client joins the two parts together again.[10] By splitting the exponent in smaller parts (resulting in more calls to the server), the amount of group operations performed on the client can be reduced.

Finally, the last two protocols discussed, apply again similar blinding techniques as used before, we omit further discussion. It is however important to note, that they rely on either Protocol 4 or Protocol 5 to outsource (and thus speed up) parts of their calculation. Protocol 6 outsources $x^y = z$ and $x^y = n$ (public or private result with secret base and secret exponent) calculations:

Protocol: S1P6(u, a, p)
Input: Base $u \in \mathbb{Z}_p^*$, exponent $a \in \mathbb{Z}_q$, prime modulus p
Output: $u^a \in \mathbb{Z}_p^*$

$(r_1, g^{r_1}) \leftarrow$ RandomPair()
$(r_2, g^{r_2}) \leftarrow$ RandomPair()

$v \leftarrow u \cdot g^{r_1} \mod p$
$h \leftarrow v^a \mod p$ // Delegated with Protocol 4 or 5

$s_1 \leftarrow \mathbf{S}_1(g, -ar_1 - r_2 \mod q, p)$

return $h \cdot s_1 \cdot g^{r_2} \mod p$

Protocol 4.2.7: S1P6: Variable base $x^y = z$ and $x^y = n$ outsourcing from [27]

If the exponent is not required to be kept secret ($x^a = z$), there are two optimizations to Protocol 6. Protocol 8 from [27] uses an advanced method based on Gallant, Lambert and Vanstone's decomposition algorithm [33], which is beyond the scope of this thesis. We thus introduce as last protocol Protocol 7, which does not require this algorithm and improves Protocol 6 for the client by only requiring a constant number of group operations:

Protocol: S1P7(u, a, p)
Input: Base $u \in \mathbb{Z}_p^*$, exponent $a \in \mathbb{Z}_q$, prime modulus p
Output: $u^a \in \mathbb{Z}_p^*$

$(r_1, g^{r_1}) \leftarrow$ RandomPair()
$(r_2, g^{r_2}) \leftarrow$ RandomPair() // Attention: $r_2 \in \mathbb{Z}_q^*$!
$(r_3, g^{r_3}) \leftarrow$ RandomPair()

$k \leftarrow (r_3 - r_1 \cdot a)/r_2 \mod q$

$s_1 \leftarrow \mathbf{S}_1(u \cdot g^{r_1} \mod p, a, p)$
$s_2 \leftarrow \mathbf{S}_1(g^{r_2} \mod p, k, p)$

return $s_1 \cdot s_2 \cdot g^{r_3} \mod p$

Protocol 4.2.8: S1P7: Variable base $x^a = z$ outsourcing from [27]

[10] An efficient method, the so-called *simultaneous* 2^w*-ary method*, denoted as TwoWary() in the algorithm, is suggested by the authors in [27]. It was introduced by Straus [45].

Summary

Chevalier et al. provide a valuable taxonomy of protocols for outsourcing modular exponentiations to a single server. A whole family of protocols for different applications with fixed or variable base and varying secrecy requirements is introduced in their work. We have given an overview of those protocols which could be relevant for implementation in this thesis, omitting protocols requiring too advanced techniques. We refer to the original work [27, 28] for further informations, especially regarding computational complexity considerations.

4.2.4. Kiraz and Uzunkol (S2)

Also in 2016, Kiraz and Uzunkol [38] published a protocol for fixed, secret base and secret exponent exponentiation to a single server which *provides verifiability*.

This protocol is by far the most complex in the literature discussed in this chapter; we did not further investigate a possible implementation for that reason and also, because the protocol does not provide any benefits to our applications.

For completeness, we include the protocol in the conclusive overview (section 4.5).

4.2.5. Further Related Work

Besides the papers that were discussed in this chapter, others have been consulted but not considered for this thesis. In this section, we briefly describe each paper and state, why we did not include it in our work.

In 2013, a completely different method, whose security relies on the subset sum problem, has been published by Ma et al. [42]. Protocols for outsourcing to a single server and to multiple servers are given by the authors. The basic idea is to hide the effective computation in a large set of random modexps, making it hard for the adversary to find the relevant one. Being a very interesting approach, we have dismissed it for our purposes due to the fact that every outsourced modexp needs a large amount (>200) of randomized modexps for blinding. Furthermore, Kiraz and Uzunkol [38] describe a checkability issue in this paper which we did not further investigate.

Another publication by Cavallo et al. [25] in 2015 presents two protocols for outsourcing private variable base and public exponent exponentiation to a single server. These protocols have been analyzed by Chevalier et al. in [27, 28], which noted performance issues in the first protocol and stronger requirements for the RandomPair() function in the second. We dismissed the paper due to these issues. However, a very simple protocol for outsourcing of modular inverses to a single server is also given by Cavallo et al. which we considered to be useful for our purposes, we describe it in section 4.3.1.

Lin et al. [40] proposed in 2016 an even more efficient variant of protocol M2, most notably reducing the number of modular inversions from 3 to 1 and having only 3 invocations of RandomPair (instead of 5). Furthermore, they claim to break the protocol for simultaneous modular exponentiation by Chen et al. (see section 4.2.2 for details). As of date of writing, no peer reviewed version of this paper could be found; for this reason, we did not further consider it.

4.3. Outsourcing Supporting Functions

Besides outsourcing modular exponentiation, for certain applications of the protocols, it may also make sense to have outsourced versions of additional and supporting functions. In this section we describe three such functions and how to outsource them.

4.3.1. Modular Multiplicative Inverse (INV)

We have already introduced inverse elements in the introduction to this chapter (see page 9). Finding inverses using the extended Euclidean algorithm has a polynomial time complexity of $O\left((log_2 n)^2\right)$ operations for a given input n (see [43, Chapter 2, p. 62]). Even being solvable computationally efficient, outsourcing such calculations may still be desirable if computational resources are limited.

Cavallo et al. [25] provide a generic outsourcing protocol for inverses which is not restricted to modular multiplicative groups. We give a pseudo-code representation of their protocol for $\mathbb{Z}_p^*$; we denote the calculation of a multiplicative modular inverse $d^{-1} \bmod p$ on a remote server as a call to the function $\mathbf{S}_1(d, p)$:

Protocol: INV(n, p)

Input: Element $n \in \mathbb{Z}_p^*$, prime modulus p
Output: $n^{-1} \in \mathbb{Z}_p^*$

$r \in_R \mathbb{Z}_p^*$
$v \leftarrow r \cdot n \mod p$

$s_1 \leftarrow \mathbf{S}_1(v, p)$

if $v \cdot s_1 \mod p \neq 1$ **then**
    return *ERROR*
else
    return $r \cdot s_1 \mod p$
end

Protocol 4.3.1: INV: Outsourced inverse in $\mathbb{Z}_p^*$ based on [25]

4.3.2. Outsourced Random Modular Exponentiation (RANDEXP)

Generation of randomness is often also an issue on computationally limited devices. Based on ideas taken from protocols for outsourcing of modular exponentiation, we introduce, without any security proof, a simple protocol for the outsourcing of the RandomPair() algorithm. This protocol could be useful for instance to obtain the blinding pairs required by the various outsourcing protocols.

Our protocol RANDEXP has the same prerequisites as given before: We require two non-colluding servers as well as secure channels for transmission. In the following pseudo-code representation, a

call to the server ($\mathbf{S}_i(g, p)$) returns a pair $(r, g^r) \in \mathbb{Z}_p^*$ with r being randomly chosen:

Protocol: RANDEXP(g, p)
Input: Generator $g \in \mathbb{Z}_p^*$, prime modulus p
Output: $(r, g^r) \in \mathbb{Z}_p^*$

$(r_{s_1}, g^{r_{s_1}}) \leftarrow \mathbf{S}_1(g, p)$
$(r_{s_2}, g^{r_{s_2}}) \leftarrow \mathbf{S}_2(g, p)$

$r \leftarrow r_{s_1} + r_{s_2} \mod q$
$g^r \leftarrow g^{r_{s_1}} \cdot g^{r_{s_2}} \mod p$

return (r, g^r)

Protocol 4.3.2: RANDEXP: Outsourced public base random modular exponentiation $(r, g^r) \bmod p$

Obviously, this protocol is uncheckable, as the client cannot know if a pair is correct without further validation. Such a validation could be implemented in different ways which we do not further detail; one possibility would be for instance to rely on protocols for outsourcing modexps.

4.3.3. Outsourced Random Multiplicative Inverse (RANDINV)

Based on the same motivation as the aforementioned RANDEXP protocol (4.3.2), we also briefly introduce the RANDINV protocol:

RANDINV returns pairs $(r, r^{-1}) \bmod p$, with r^{-1} being the multiplicative modular inverse of r, a (pseudo-)random number generated on every invocation independently of the previously generated one. Checkability is provided analogously to INV by using the fact that $r \cdot r^{-1} \equiv 1 \pmod p$.

In the following pseudo-code representation, a call to the server ($\mathbf{S}_i(p)$) returns a pair $(r, r^{-1}) \in \mathbb{Z}_p^*$:

Protocol: RANDINV(p)
Input: Prime modulus p
Output: $(r, r^{-1}) \in \mathbb{Z}_p^*$

$(r_{s_1}, r_{s_1}^{-1}) \leftarrow \mathbf{S}_1(p)$
$(r_{s_2}, r_{s_2}^{-1}) \leftarrow \mathbf{S}_2(p)$

if $r_{s_1} \cdot r_{s_1}^{-1} \mod p = r_{s_2} \cdot r_{s_2}^{-1} \mod p = 1$ **then**
    return $(r_{s_1} \cdot r_{s_2} \bmod p, r_{s_1}^{-1} \cdot r_{s_2}^{-1} \bmod p)$
else
    return *ERROR*
end

Protocol 4.3.3: RANDINV: Outsourced random multiplicative inverse $(r, r^{-1}) \mod p$

4.4. The RandomPair() Algorithm

Many of the presented protocols rely on an algorithm for finding pairs (r, g^r), with r being random and different on every invocation. In [35], the authors propose two possible options:

1. Initialize the client with a large enough list of such pairs originating from a trusted source. The client then chooses a pair (or a combination of pairs) at random every time.
2. Use a specialized algorithm for generating such pairs on the fly.

The first option is straightforward: A list containing random pairs is precomputed and given to the client before any computation on the client has to occur. The source of these pairs must be trustworthy and the transmission of the pairs must occur over a secure channel. After the client has been initialized, a single or multiple pairs can be chosen out of the list at random every time a pair is required. The pairs can possibly be combined together to form new pairs by different methods which we won't detail further. While being simple, this method has the drawback of strongly limiting the amount and distribution of available random pairs.

There are efficient algorithms available which provide a stream of random pairs with a better distribution; the algorithm proposed in [35] is the EBPV generator by Nguyen et al.[44]. A thorough investigation of such algorithms is out of scope for our work and we will thus not go into more details here.[11]

4.5. Conclusive Overview

We conclude this chapter by giving an overview of all previously discussed protocols in Table 4.2.

The first three rows indicate, which protocol is applicable for the desired outsourced modular exponentiation depending on the amount of servers given (1, 2 or 4) and if checkability is required or not. For the cases with two servers, the optimal protocol is underlined; for the one server protocols, the optimal protocol depends parametrization and if the base is fixed or not.

For one server without checkability, the appropriate protocol from Chevalier et al. is indicated for both, fixed and variable base. If checkability is required, only Kiraz and Uzunkol (S2) with secret base and exponent can be applied to the single server case.

With two servers, either our DEC protocols for secret base or secret exponent are applicable without checkability; otherwise Hohenberger / Lysyanskaya (M1) and Chen et al. (M2) may be considered. Clearly, M2 is favorable due to improved efficiency.

Finally, DEC protocols 3 and 4 can be used if distribution to four servers is desired with or without checkability.

For the supporting protocols, INV runs with a single server and provides checkability for the cost of a single, modular multiplication (which could also be dismissed if really required). Running the protocol with more than one server provides no benefits in terms of secrecy or performance.

RANDEXP and RANDINV both require two servers. RANDEXP is by definition not checkable without further outsourcing, while RANDINV, comparable to INV, provides checkability at the cost of a single, modular multiplication. Again, adding more servers leads to no improvements.

[11] Meeting with advisor, 12th December 2016.

Table 4.2.: Overview of protocols and application

Protocol	**1 Server**		**2 Server**		**4 Servers**	
Checkable	no	yes	no	yes	no	yes
$u^y = z$	S1P3, S1P4, S1P5		DEC2, M1[3], M2[3]	M1, M2, DEC2[2]		
$x^a = z$	S1P1, S1P7, S1P8		DEC1, M1[3], M2[3]	M1, M2, DEC1[2]		
$x^y = z$	S1P1, S1P6	S2	M1[3], M2[3]	M1, M2	DEC3	DEC4
$x^{-1} = z$	INV[1]	INV				
(r, g^r)			RANDEXP			
(r, r^{-1})			RANDINV[1]	RANDINV		

[1] The single modular multiplication for checkability could also be dismissed.
[2] By adding checkability as described for DEC4.
[3] By removing relevant parts for checkability, without further proof.

5. Implemented Solution

The previous chapters described the motivation for outsourcing modular exponentiations (Chapter 2), the objectives an implemented solution should fulfill (Chapter 3) and finally laid out the theoretical foundations necessary (Chapter 4).

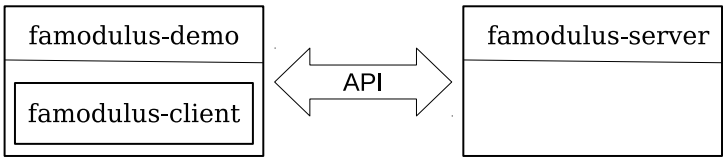
In this chapter, we will describe relevant aspects of the solution we have implemented. We focus only on important parts of the software, its source code and underlying ideas, not on installation and usage, which is detailed in Appendix A (p. 63).

While following this chapter, we suggest to study the source code in parallel for reference; a description of accessing and retrieving it can be found as well in Appendix A, in Section A.1.2 (p. 64).

5.1. Introducing famodulus

As part of this thesis, we have developed a fully working system corresponding to the objectives given in Chapter 3. This system, which we call famodulus, is depicted in Figure 5.1 and consists of three main components:[1]

Figure 5.1.: Overview of famodulus components



The three components, which will be described in greater detail in the following sections, are:

- famodulus-server, back-end server for performing calculations
- famodulus-client, JavaScript library for outsourcing calculations to famodulus-server
- famodulus-demo, a comprehensive demonstrator application using famodulus-client

famodulus-client interacts with famodulus-demo using a RESTful API based on HTTP, which is described in Section 5.5 below. There is no other coupling between these components and they can also be used independently or replaced with other implementations. famodulus-demo, which is a demonstrator application for famodulus-client, naturally depends on the former.

[1] famodulus is a combination of the Latin words *famulus*, signifying servant, and *modulus* (measure): famodulus is the servant for modular exponentiation calculation.

5.2. famodulus-server

famodulus-server can be considered the smallest component in terms of complexity and size of code. Its only purpose is to provide any client (most likely running on a different machine) access to the computing power of the host it is running on. For the system developed during this thesis, only the function for modular exponentiation on big integers has to be provided.

As the use cases given in Section 2.1 (Chapter 2, p. 3) are targeting a Java environment, implementing the server in Java as well has been considered the most sensible solution. Using *JAX-RS* [10], the Java API for RESTful services, is considered best current practice when implementing RESTful interfaces in Java. Multiple implementations for JAX-RS exist, we have chosen its reference implementation, *Jersey* [12]. Jersey applications support deployment to different containers, ranging from simple HTTP servers up to Java EE and OSGi. Discovery of resources is accomplished during run time with annotations in the source code.

As container, we are using the *Grizzly* [17] standalone HTTP server. We expect that famodulus-server will almost always be running standalone and should not come with additional overhead; efficiency being one of the key objectives. Furthermore, Grizzly makes bundling of the container with the application straightforward and deployment and usage simple for the user. Still, deploying famodulus-server to other containers should be possible without hassles.

5.2.1. Overview of Source Code

famodulus-server is built as Maven [2] project with the top-level `pom.xml` file containing all relevant information. We only require three external dependencies (besides JUnit for testing): the Grizzly2 container for Jersey, Jersey Moxy for JSON parsing and serialization as well as jnagmp [13], a native binding to the GNU multi precision library GMP [18], for improved modular exponentiation performance.

All source code can be found in the `src` directory, which contains a Java class hierarchy for the main code as well as for the unit tests. Furthermore, a static HTML file is located in the source tree at `src/main/resources`, it serves as homepage for the server if accessed by a browser and informs about the service running.

The main Java package for the server is `ch.mainini.famodulus.server`, which contains two classes. Class `Server` provides the main method which is used to start the server. It performs automatic resource discovery for the JAX-RS resources and adds a handler for serving the static page.

Class `CORSResponseFilter` interacts with the HTTP response and adds so-called CORS [22] headers to it, before it is sent back to the client. The reason for this is, that clients normally run in another origin relative to the server (i.e. have not originally been served by the same server). For security reasons, web standards mandate CORS before allowing such *cross origin requests*, which in our case are used for transmitting the modexp calculations to the server.

`BigIntegerStringAdapter`, located in the `util` sub package, performs the transformation of internally used BigIntegers into strings for the JSON serialization and vice versa. This step is required, as otherwise, BigIntegers would be serialized as JSON numbers, which cannot be handled by the JavaScript on the client side.

5.2.2. Calculation of Modular Exponentiations

All code relevant to the implementation of the RESTful API described in Section 5.5 can be found in the package `ch.mainini.famodulus.server.modexp`.

The JSON messages, which are exchanged by the API, are modeled as two Java Beans internally: the `ModExpQueryBean` encapsulates a full modexp query, containing individual modexps which in turn are encapsulated in the `ModExpBean`. Both beans are simple POJOs without any further functionality.

The actual calculation of the modexps occurs in the `query()` method of class `ModExpResource.java`; as can be seen in the source code (Figure 5.2), implementation is fairly simple.

The class exposes a JAX-RS resource supporting HTTP POST requests for modexp queries in JSON, annotations found in lines 1-3 are used for configuring the resource. Lines 6-9 assign defaults, which where possibly submitted with the request, to local variables for speeding up the main calculation loop. This loop, which starts in line 11, iterates over all modexps found in the query, applies default values where needed (lines 12-14), performs modular exponentiation (line 16) and creates the response (lines 17-25). The response is appended to the original data of the query, with parameters being removed if a brief response has to be returned (parameters are explicitly set to `null` in that case).

The initial release of famodulus-server relied on `BigInteger.modPow()` for modexp calculation. With version 1.1.0, the code now uses jnagmp [13], which provides a Java native binding to the GMP library [18] and offers a performance increase of roughly four times. [8]

5.2.3. Quality Assurance

Quality of the source code has been defined as important objective (3.1, p. 5). Besides using standardized components and following best practices, the code is fully documented using Javadoc.

Rigorous unit testing has been a key point from start, unit tests for all classes are included in the separate class hierarchy found in `src/test`. They cover basic API functionality as well as specifically designed edge cases.

Coverage of code by unit tests has been measured throughout development using Cobertura [5], see Appendix A (p. 63) for details on generating the reports. Except for the `Server` class, all classes are fully covered by unit tests; as `Server` contains the main method, testing it would require special measures to be taken (and furthermore, the code in the main class is trivial and failure would be detected immediately).

Figure 5.2.: Method query() from ModExpResource.java

```
1   @POST
2   @Consumes(MediaType.APPLICATION_JSON)
3   @Produces(MediaType.APPLICATION_JSON)
4   public ModExpQueryBean query(ModExpQueryBean query) {
5       final long startTime = System.nanoTime();
6       final BigInteger defaultModulus = query.getModulus();
7       final BigInteger defaultBase = query.getBase();
8       final BigInteger defaultExponent = query.getExponent();
9       final boolean briefResponse = query.getBrief();
10
11      for(ModExpBean modexp: query.getModexps()) {
12          final BigInteger m = modexp.getModulus() != null ? modexp.getModulus() : defaultModulus;
13          final BigInteger b = modexp.getBase() != null ? modexp.getBase() : defaultBase;
14          final BigInteger e = modexp.getExponent() != null ? modexp.getExponent() : defaultExponent;
15
16          final BigInteger r = Gmp.modPowSecure(b, e, m);
17          modexp.setResult(r);
18          LOG.finest(String.format("Calculated modexp, m: %s, b: %s, e: %s, r: %s ...",
19                  m.toString(16), b.toString(16), e.toString(16), r.toString(16)));
20
21          if(briefResponse) {
22              modexp.setModulus(null);
23              modexp.setBase(null);
24              modexp.setExponent(null);
25          }
26      }
27
28      if(briefResponse) {
29          query.setModulus(null);
30          query.setBase(null);
31          query.setExponent(null);
32          query.setBrief(null);
33      }
34
35      LOG.fine(String.format("Calculation took %f ms.", (System.nanoTime() - startTime) / 1000000.0));
36      return query;
37  }
```

5.3. famodulus-client

The famodulus-client library has been designed with an application developer in mind, who wants to use the library for outsourcing calculations. The main objectives for implementation where:

- Consistent and simple API for the developer
- Clean and robust implementation
- Performance
- Simple to extend with new protocols

To achieve these objectives, a careful choice of development method, required libraries and design of APIs had to be made, which will now be described.

5.3.1. JavaScript Development Methods

As the library targets the browser, it had to be implemented in JavaScript. With the release of version 6 of ECMAScript (ES6 for short) in 2015 [7] , a major rework of the language has occurred and it now provides many modern features and lean semantics.[2] While the adoption of ES6 by browser vendors took some time, most of the features are now supported by current browsers, and if not, so-called *polyfills* may be used, which substitute a given functionality using ES5 code.

With the release of the *Chrome V8* JavaScript [4] engine by Google in 2008 and the introduction of of the *node.js* [15] JavaScript runtime in 2009, JavaScript applications have become increasingly widespread outside of the browser. Often nowadays, hybrid code, which works in both environments, is developed. *CommonJS* [6], a module system adopted by node.js and its package manager *npm* counts roughly 350'000 modules [16] today.

famodulus-client has been developed as npm module, even though its main field of application is the browser. There are multiple reasons for this decision:

- Modularity during development, separation of concerns
- More structured development and build cycle
- Unit testing can be done without browser
- The library is more versatile and can be used in non-browser environments as well[3]

Using npm as minimal build tool, a build life cycle has been established, which allows to run unit tests, code coverage and linting in a similar fashion as with Maven for the server side.

For usage in the browser, the library is assembled out of its individual parts by *Browserify* [3], which creates a single file including all dependencies, to be loaded in a single `<script>` tag by the application developer.

[2] JavaScript is standardized by the ECMA organization in standard ECMA-262.

[3] There are microcontrollers (having far less computing power for modular exponentiation than even a tablet) running node.js today

5.3.2. Evaluation of Big Integer Libraries for JavaScript

Outsourcing protocols need to make some calculations with big integers in JavaScript and on the client side. Opposed to Java, JavaScript has no built-in support for working with big integers. At the beginning of development, a small benchmark of various libraries for big integer operations in JavaScript has been conducted in order to find the fastest one.

Based on the results, two libraries where considered for famodulus-client: the Library by Leemon [11][4] and the Verificatum library [20], with the latter being even slightly faster. Finally, the decision to use Leemon's library has been taken, based on licensing concerns: Leemons library is in the public domain, whereas Verificatum is explicitly declared as non free software with a proprietary license (still allowing free usage for research purposes, though). For benchmarking, famodulus-demo described in the next section offers the possibility to use Verificatum instead of Leemon for reference calculations.

5.3.3. Overview of Source Code

Source code for famodulus-client consists of four JavaScript files in the subdirectory `lib`; the corresponding unit tests are placed in the `test` directory.

The API of the famodulus-client library is found in file `client.js`. This file only provides wrapper functions performing argument checking and offering a more comfortable interface to the application developer.

Outsourcing protocols, currently found in the files `direct.js` for direct outsourcing (which simply sends the modexps directly and without any blinding to the server) and `dec.js` for protocols in the DEC family, have to follow internal conventions. A protocol is implemented as JavaScript function with defined parameters `modexps`, `defaults` and `options`. Following that convention, unit tests can be generalized, and the protocols may be exchanged or used directly by external applications. `options` is a JavaScript object with at least the attribute `server` (if the protocol only supports a single server) or `servers` (array of servers, in case multiple servers are required or supported). Refer to Section 5.5 below for a description of `modexps` and `defaults`.

File `util.js` contains various helper functions which can be used for implementing outsourcing protocols. Most importantly, functions for generating random big integers (based on the Web Cryptography API [21]) as well as for random shuffling of lists are provided. Also, a function used for sending requests to famodulus-server provides a uniform API independent of the environment (browser or node.js).

5.3.4. Quality Assurance

The same objectives in terms of quality as for the server also apply to famodulus-client, the same measures where taken to assure their fulfillment. Implementation has been done using best current practices and with only a minimal set of external dependencies. All code is fully documented using JSDoc [14].

[4] During the project, the library has vanished from the URL given in the references. An inquiry was sent to the author about the reason for this, however no response has been obtained so far. The library can however still be found on github and npm.

Unit tests, which may be run using node.js or in the browser without difference provide a rigorous assessment of outsourcing protocols and of the famodulus-client API. Code coverage has been ensured throughout development using Istanbul [9], which provides reports similar to Cobertura.

Please refer again to Appendix A (p. 63) for details on running the unit tests or generating the coverage report.

5.4. famodulus-demo

The last component of the famodulus system is famodulus-demo. Its main purpose is to provide a demonstrator application for famodulus-client and to serve as a tool for benchmarking outsourced modular exponentiations. famodulus-demo is a web application which relies on the famodulus-client library to outsource calculations to one or more famodulus-server instances. For screen shots of significant parts of the web application, please refer to Appendix A (p. 63).

To ease its use, the web application is also bundled together with a Grizzly web server, which takes care of serving all relevant resources. This server part will not be further detailed, it mainly corresponds to the famodulus-server implementation with all parts specific to the modexp API removed.

As for famodulus-server, the source is also built as a Maven project. All resources of the web application are located in the folder `/src/main/resources/ch/mainini/famodulus/demo`, with subdirectories for individual resource types. The file `demo.html` contains the entry form in HTML5 (Figure A.3, p. 69). The form requires various parts of JavaScript to fulfill its function, a list containing all loaded scripts with a short description is given in Table 5.1.

Table 5.1.: Scripts required by demo.html in loading order

Name	Used for
`js/jquery.min.js`	Simpler interaction with HTML elements (DOM)
`js/bootstrap.min.js`	Required by layout (CSS)
`js/BigInt.js`	Local modexp calculation
`js/famodulus.browser.js`	Remote modexp calculation
`js/fd.js`	Main code of famodulus-demo
`js/controller.js`	Mapping of form interaction to `fd.js`

`controller.js` is used to bind callback functions to actions in the form. Besides delegating those actions to `fd.js`, only input validation and setting of default values is performed by this script. `fd.js` contains the main functionality of the demonstrator application, which can roughly be divided in three sections (marked accordingly in the source code):

1. Helper functions for working with form fields (getting or setting values, providing default values, etc.), the generator for generating random modexp parameters and helpers for layouting.
2. A set of functions which are used to compare and display results of calculations. Results, timing information and other parameters are bound to the global `FD` object. Calculations, for outsourced modexps running asynchronously, sets those parameters, which then get evaluated by those functions.

3. The last set of functions is used for calculating local and outsourced modexps. After retrieving and preparing relevant data from the form, they interact with famodulus-client or the local big integer libraries and retrieve the results of calculations.

Benchmarking of performance and comparison between local and outsourced modular exponentiation is an important aspect of the demonstrator application. For this, the same modexps have to be calculated in the browser and remotely using famodulus-client. For local calculations, Leemon's library is used by default, however the Verificatum library can be dynamically loaded to achieve slightly better values for local calculations, see Section A.2.2 of Appendix A for details.

5.5. RESTful API

RESTful interfaces have become increasingly important for APIs and are dominant in web technologies. We do not further detail them here, please refer to Roy Fieldings thesis [32, Chapter 5] for more informations regarding REST architectures.

The API connecting famodulus-client with famodulus-server provides a single method for outsourcing modular exponentiations to the server: transmission of the modexp data using the `HTTP POST` method; in general, the modexp resource described in Section 5.2 is bound to the URI `/api/modexp/`.

5.5.1. Sending a Request

Requests are sent to the server by encoding the modexps as JSON, following the structure given in Figure 5.3.

Figure 5.3.: Structure of modexp API JSON data

```
1      { "brief": boolean,
2        "b": "default base",
3        "e": "default exponent",
4        "m": "default modulus",
5        "modexps": [ modexp1, modexp2, ... ]
6      }
```

`brief` is a boolean option telling the server to either only return the calculated results (true), or to return the results including the original modexp parameters (false). It can be omitted, in which case it defaults to true.

`"b"`, `"e"` and `"m"` are default parameters for modexps; they are applied if any of the enclosed modexps misses one or more parameter. If, for example, all modexps share the same modulus, `"m"` may be given as default and omitted in all enclosed modexps. If none of the enclosed modexps misses a parameter, the default values may be omitted.

`modexps` in turn is an array of JSON objects containing an individual modexp with base, exponent and modulus in the form `{"b":"base", "e":"exponent", "m":"modulus"}`.

Note: all numbers, including the results, are hexadecimal strings.

5.5.2. Receiving a Response

When the server has finished calculating the modexps, the returned result depends on the value of the `"brief"` attribute:

- If `brief` is `false`, it returns a copy of the original request, with the modexp objects having an additional `"r"` attribute per modexp which contains the result.
- Else, if `brief` is `true`, default parameters are removed and the modexp objects only contain the `"r"` attribute.

6. Performance Analysis

After having implemented DEC2 as first outsourcing protocol with blinding, a few tests where made with arbitrary parameters. Immediately, it became clear that the performance obtained with DEC2 was largely inferior to the one experienced while running tests with direct outsourcing. Without knowing the exact cause for this degradation, implementation of further protocols would not have made sense.

This chapter describes the thorough performance analysis that we have conducted after discovering the problem. We start with a description of the testing methodology and continue with an assessment of the data obtained during the tests. Subsequently, we present our results and conclude with an identification of the root cause for the performance degradation.

6.1. Methodology and Test Setup

All tests where conducted on a single machine with a Core i7 CPU (eight cores), running at 1.73 GHz and having 8 GB of RAM. As operating system, Debian GNU/Linux on the current testing branch has been used.

During the tests, the machine was running two famodulus-server instances as well as a famodulus-demo web server. Using the *taskset* utility, each process has been assigned to a different CPU core and adherence to this setting was monitored. All processes which were not required for the tests, monitoring or for the operating system itself have been stopped. Memory consumption during the tests was monitored throughout.

Tests where conducted with an off-the-shelf Firefox 50.1.0, without any specific configuration and with all network communication taking place over the loop-back device. The Firefox process has also been pinned to a separate CPU core.

Measurement of times has been done for most parts directly in the famodulus applications. For the browser side, times where measured by famodulus-demo as part of its functionality. On the server, the logging level for famodulus-server has been set to `FINE`, which enables logging the duration of the modexp calculation (see Section A.2.1 of Appendix A, p. 63 for details on configuration). Additionally, the built-in Firefox developer tools have been used for performance measurements and runtime analysis. All test data obtained has been stored in CSV files, which where then further analyzed using LibreOffice Calc.

Calculations have been run in batches of 10 to 10'000 modexps each. All calculations in the browser, besides those required by famodulus-client, have been performed using the injected Verificatum library (see Section A.2.2, p. 68 for details).

We have noticed that timings differ largely when running single modexp calculations, while the obtained values are more stable, the larger the number of modexps is. This is probably due to factors external to the protocols; we suspect influences from just-in-time compilation in JavaScript and/or events of the operating system to be possible sources for deviation.

To ensure a certain degree of accuracy, all calculations have been performed either five times (comparison of parameter lengths) or three times (remaining tests) using the same parameters. For the analysis, we used the average of the measured times; if the tests included the DEC2 protocol (which makes simultaneous requests to both servers), the maximum of the averaged times measured on both servers has been taken. With these precautions, and comparing to values obtained during hundreds of tests during the development process, we are confident of having minimized possible side effects in the results as good as possible.

6.2. Overall Running Time Performance

To obtain reference values for further comparisons, two initial series of tests with 100 modexps each where conducted using direct outsourcing. We have compared running time, while varying the input parameters as follows:

1. Per modexp random base and exponent of 1024 bits; a single, fixed prime modulus from 1024 to 4096 bit
2. Per modexp random base and exponent of sizes varying between 256 and 4096 bits; a single, fixed prime modulus of 1024 bits

The obtained results are presented in Figures 6.1 and 6.2 respectively. It can be seen immediately, that calculation on the server side is much faster than in the browser. Also, the values obtained match our expectations regarding asymptotic behavior: modulus size has polynomial influence, base and exponent only linear. We will come back on this in Section 6.4 below.

Figure 6.1.: Running time comparison client/server for 100 modexps with varying modulus size

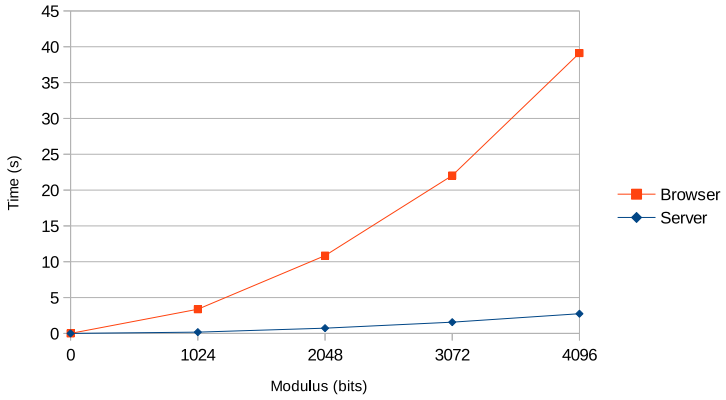
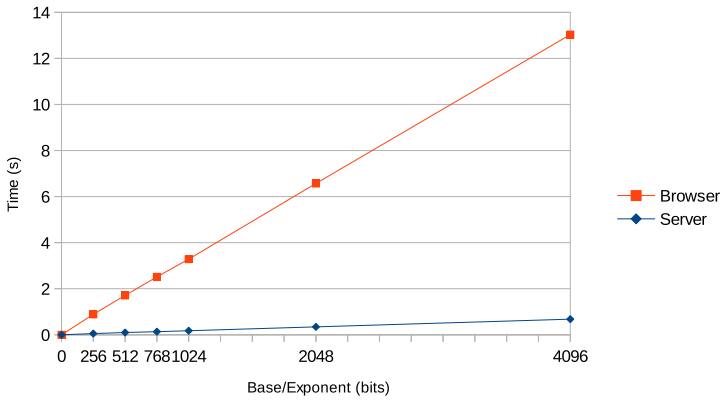


Figure 6.2.: Running time comparison client/server for 100 modexps with varying base/exponent size



6.2.1. Absolute Running Times

With a second series of tests, our observations regarding low DEC2 performance should be confirmed. For this, in-browser calculation time has been compared to times obtained from outsourcing the same calculations using direct outsourcing as well as checked (C) and unchecked (U) DEC2.

Figure 6.3 shows the running times when calculating different batches, ranging from 10 to 10'000 modexps (each with a different, 512 bit random base/exponent and the same 1024 bit prime modulus).

Figure 6.3.: Absolute running time comparison between protocols and locally

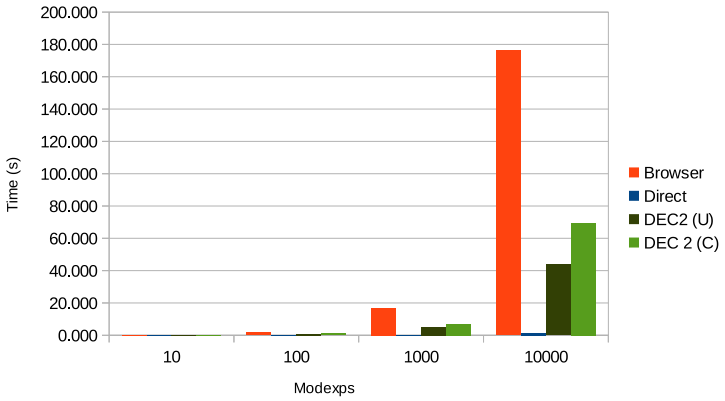
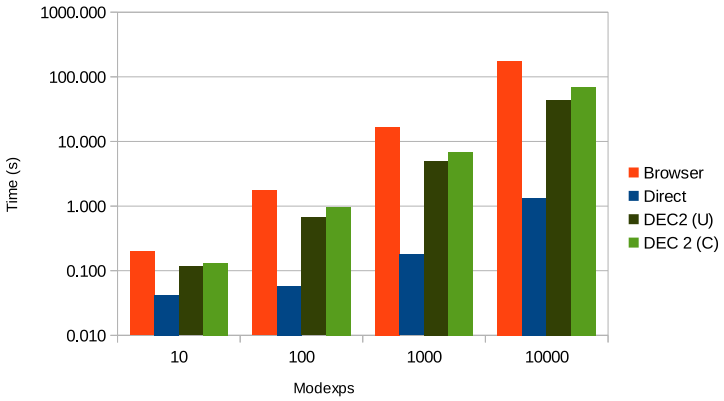


Figure 6.4, depicting the same results on a logarithmic time scale, shows that the relative difference between running times of all protocols is constant and independent of the input size (at least up to 10'000 modexps). This is an important observation which leads us to the assumption, that external factors (e.g. JavaScript memory management) do not have a significant impact on the running time.

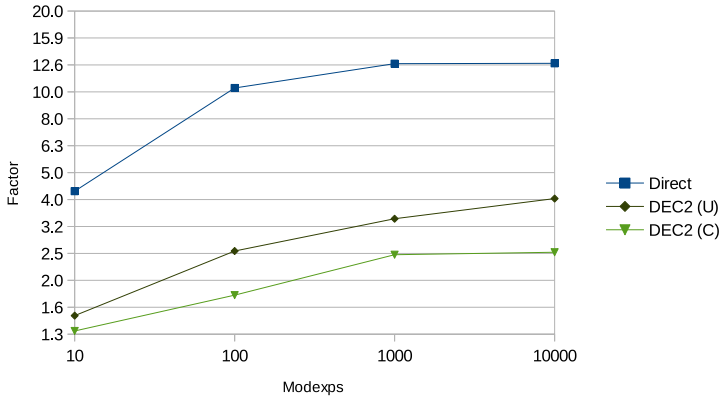
Figure 6.4.: Absolute running time comparison between protocols and locally (log scale)



6.2.2. Relative Performance Gain

Based on the data acquired so far, we were able to establish the relative performance gain of direct outsourcing and outsourcing using DEC2, compared to calculation in the browser. The result, shown in Figure 6.5, represents fractions of the time used per protocol compared to the calculations in the browser.

Figure 6.5.: Runtime factors compared to local calculation (log scale)



We observe, that with a given amount of modexps, the runtime fraction seems to even out at a certain point. We consider this due to client overhead and network latency becoming increasingly less influential on running time, the longer the calculations take; i.e. the more modexps there are to calculate.

For direct outsourcing, a stable value is attained at roughly 12 times less running time compared to local calculation. In contrast, checked DEC2 evens out only at around roughly 2.5 times performance gain, while for the unchecked variant, data of up to 10'000 modexps does not indicate a stable factor with certainty. Looking at the data, we would expect evening out somewhere between 4 and 5, which would also make sense compared to checked DEC2, which requires twice as much modexps to be calculated by the server.

6.3. Investigating DEC2 Performance Degradation

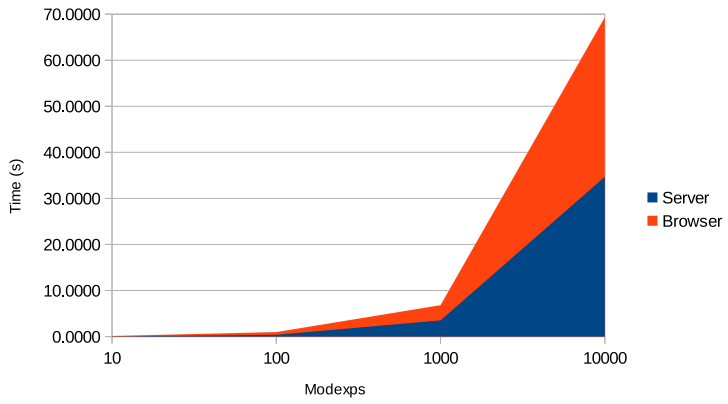
The results in the previous section have substantiated the initially observed degradation in outsourcing performance using DEC2. Looking at Figures 6.1 and 6.2, we can be quite confident that the modexp calculation on the server is efficient enough and should not be the main issue. Of course, as DEC2 is definitely more complex than direct outsourcing, we would expect its performance to be lower than the values obtained by direct outsourcing; however three to five times less seems not plausible when analyzing its algorithmic complexity.

Revisiting the protocol (Protocol 4.1.2, p. 13), we identify two computationally expensive operations as possible causes for the degradation: the modular multiplication, which is in $O(n^2)$ (or even $O(n^{log_2 3})$ or below, depending on implementation and input) and the modular exponentiation, being somewhere between $O(n^2)$ and $O(n^3)$.[1] From this, it is unlikely, that the measured, important difference in running time is due to the the additional multiplication, when weighted against the modexp calculations occurring on the server.

6.3.1. Running Time Distribution

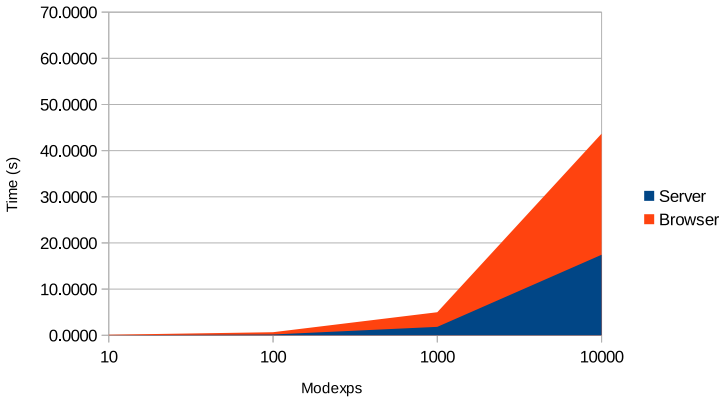
As a first step in investigating the possible causes of performance degradation, the amount of time spent in the browser has been compared to the amount of time spent on the server. Figure 6.6 and even more Figure 6.7 show, that indeed a large part of the running time is spent in the browser. In fact, it is around 50% (checked) and even 60% (unchecked) for 1000 and 10'000 modexps, respectively.

Figure 6.6.: Running time server vs. client, DEC 2 checked



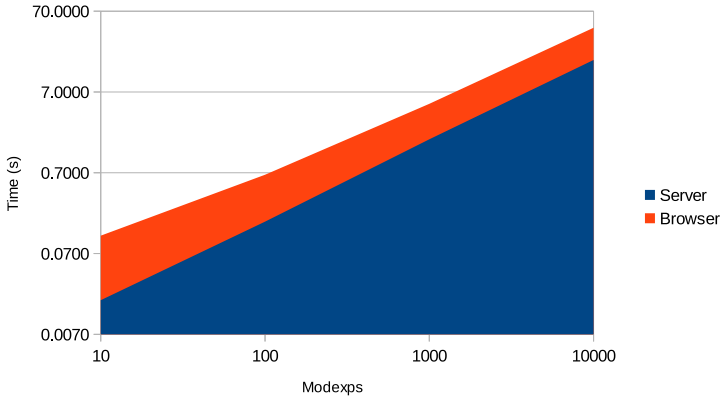
[1] See [37, Appendix B, p. 553] for more details.

Figure 6.7.: Running time server vs. client, DEC2 unchecked



The same comparison on a logarithmic scale is depicted in Figure 6.8 for DEC2 unchecked (checked looks similar and is omitted). It shows us, that the overhead on the client side is growing linearly with the amount of modexps to calculate.

Figure 6.8.: Running time server vs. client, DEC2 unchecked (log scale)



6.3.2. famodulus-client Profiling

Analyzing famodulus-client regarding the performance of the DEC2 protocol was the next logical step, after having ensured that indeed an important amount of time is spent on the client side.

Using the performance analyzer of the Firefox developer tools, different invocations of DEC2 have been profiled. A typical result is given in Figures 6.9 and 6.10, the former being measured before the modexps are sent to the server and the latter after the results have been received.[2] The tables shown represent a function call tree, sorted by the amount of time spent in each function.

It can immediately be seen, that roughly 60% of the time is spent in multiplication and division operations on big integers. While the division can possibly be explained by performed modular arithmetic, the multiplications *before* sending the modexps to the server do not make any sense, considering the operations used in DEC2.

[2]Depicted are the values of a test with 1000 modexps, results with smaller and larger batches are interchangeable and the timings scale accordingly.

Figure 6.9.: Call tree for DEC2 unchecked, 100 modexps, sending

476.91 ms	35.00%	476.91 ms	35.00%	477	multInt_ famodulus.browser.js:1624 localhost:8080
344.93 ms	25.31%	344.93 ms	25.31%	345	divInt_ famodulus.browser.js:1643 localhost:8080
125.98 ms	9.24%	125.98 ms	9.24%	126	str2bigInt famodulus.browser.js:1399 localhost:8080
62.99 ms	4.62%	62.99 ms	4.62%	63	Gecko
60.99 ms	4.48%	60.99 ms	4.48%	61	JIT
44.99 ms	3.30%	44.99 ms	3.30%	45	randomBytes famodulus.browser.js:4208 localhost:8080
39.99 ms	2.93%	39.99 ms	2.93%	40	bigInt2str famodulus.browser.js:1499 localhost:8080
28.99 ms	2.13%	28.99 ms	2.13%	29	random famodulus.browser.js:314 localhost:8080
26.99 ms	1.98%	26.99 ms	1.98%	27	int2bigInt famodulus.browser.js:1386 localhost:8080
25.99 ms	1.91%	25.99 ms	1.91%	26	isZero famodulus.browser.js:1489 localhost:8080
24.00 ms	1.76%	24.00 ms	1.76%	24	bitSize famodulus.browser.js:724 localhost:8080

Figure 6.10.: Call tree for DEC2 unchecked, 100 modexps, receiving

423.90 ms	42.06%	423.90 ms	42.06%	424	multInt_ famodulus.browser.js:1624 localhost:8080
157.96 ms	15.67%	157.96 ms	15.67%	158	divInt_ famodulus.browser.js:1643 localhost:8080
139.97 ms	13.89%	139.97 ms	13.89%	140	str2bigInt famodulus.browser.js:1399 localhost:8080
121.97 ms	12.10%	121.97 ms	12.10%	122	linCombShift_ famodulus.browser.js:1673 localhost:8080
36.99 ms	3.67%	36.99 ms	3.67%	37	Gecko
21.99 ms	2.18%	21.99 ms	2.18%	22	JIT
17.00 ms	1.69%	17.00 ms	1.69%	17	Graphics
14.00 ms	1.39%	14.00 ms	1.39%	14	bigInt2str famodulus.browser.js:1499 localhost:8080
12.00 ms	1.19%	12.00 ms	1.19%	12	addInt_ famodulus.browser.js:1559 localhost:8080
12.00 ms	1.19%	12.00 ms	1.19%	12	isZero famodulus.browser.js:1489 localhost:8080
10.00 ms	0.99%	10.00 ms	0.99%	10	multMod_ famodulus.browser.js:1785 localhost:8080

When unfolding the call tree further (not shown), it becomes clear, that both operations are part of the `str2bigInt` and `bigInt2str` functions of Leemon's BigInt library. The relevant code in `str2bigInt`, consisting of the loop executing the multiplications, is shown in Figure 6.11 (operations taking place in lines 8 and 9). For every character of the input string, in our case the hexadecimal representation of a big integer value with typically 1024 to 3072 bits, an iteration of the loop is performed.

Figure 6.11.: Main loop of function str2bigInt

```
1     for (i=0;i<k;i++) {
2      d=digitsStr.indexOf(s.substring(i,i+1),0);
3      if (base<=36 && d>=36)  //convert lowercase to uppercase if base<=36
4        d-=26;
5      if (d>=base || d<0) {   //stop at first illegal character
6        break;
7      }
8      multInt_(x,base);
9      addInt_(x,d);
10    }
```

6.4. Identified Root Cause and Remediation

It follows from the given observations, that for converting numbers typically used in our application to internal big integer representations, 256 to 768 multiplications have to be made per number. Clearly, this is very inefficient and must be considered as the root cause for the degradation in DEC2 performance observed. Let us briefly verify this claim.

DEC2 requires two invocations of `str2bigInt` for converting exponent and modulus before sending the modexps to the server, as well as another two invocations for converting the results obtained

from the servers.[3] This matches the times measured in profiling, which are nearly identical for both parts.

When further comparing the times consumed in the most relevant functions used by `str2bigInt` and `bigInt2str` to the overall time and the time spent on the server side, we confirm that the conversion of strings from and to big integers is responsible for the large amount of computation performed by the client. In fact, the most relevant functions make up roughly 80% of the running time on the client side, multiplication alone using roughly 40%.[4]

We thus conclude, that implementing (further) outsourcing protocols based on Leemon's library, at least without modified conversion functions, will not lead to efficient results.

A series of modifications to the code made out of own interest by R. Haenni at least indicate, that a more efficient solution for the conversions could be found and will probably substantially reduce the amount of computing time required on the client side. Unfortunately, at the late stage of the project, we were not able to conduct further tests in this direction.

Another option would also be an adaption of famodulus-client to another, underlying JavaScript big integer library, however this choice would have to be carefully validated based on the insights gained working with the current implementation.

In Chapter 7, we will further conclude our work and revisit the results obtained in this chapter.

[3] As the base is not blinded in DEC2, no calculations require the base and thus no conversion to big integers has to be done.

[4] Note that the multiplication of the results as part of DEC2 is not even part of these figures, as the `multMod` function is used for modular multiplication.

7. Conclusion and Results

With this chapter, we reach the end of our long journey through outsourcing modular exponentiations. We will now conclude our work by first taking a short look back, directly followed by a short look into the future. Finally, a highly personal conclusion will end this chapter.

7.1. Fulfillment of Objectives

Let us first revisit the objectives of the project, given in Chapter 3.

The objective of implementing a demonstrator application including client and server has been accomplished from our point of view. It has proven to be functional and comprehensive in countless tests used for the development of the software itself. To simplify benchmarking and collection of performance data, a possible extension could consist of automated test suites for different input parameters. Such functionality is not directly provided by our unit tests and would have been of help while performing the tests used for Chapter 6.

Regarding the RESTful interface between client and server, not a lot has to be said, as we consider it to be complete for the current state of the application. If at a later point, more functionality is added to client or server, an extension of the interface would follow naturally.

Clearly, the third objective could not have been reached, due to the performance degradation issue discussed in great detail in Chapter 6. To compensate for this, data gathered during the tests provides important insights for research, future implementation of outsourcing protocols and for our use cases in general. We will further detail this point in the following section.

We have invested a considerable amount of effort in documentation of the implementation, its installation and usage. With a clear separation between the components and by providing a single git repository for the installation of the whole system, we hope having made the installation as straightforward as possible. We thus consider the fourth objective, mandating simple installation and usage to be fulfilled.

To fulfill the last objective concerning high code quality, persistent source code documentation has been written throughout all components. The correct function of all code has been monitored during development using comprehensive unit tests, based on measured code coverage. For more details, please refer to Chapter 5.

7.2. Future Work

While not providing ultimate answers, we consider our results to be solid groundwork for further research on solving modular exponentiation on computationally limited devices. Analysis of the performance issue and identification of its root cause directly pointed out possible ways for solution, which have already been investigated during the final days of the project. Unfortunately, time ran out and we were not able to present them as part of our results.

As our analysis has shown, working with big integers in JavaScript needs attention. Having chosen the underlying library based on benchmarks of its mathematical functionality, we did not consider performance of other key functions provided, like converting between different number representations. This proved to be an error in our case. Future work must seriously reconsider the internal representation of big integers, either by own implementation or by evaluating the libraries again, based on the new requirements found.

Future work should also investigate other means of improving performance of modular exponentiations. Based on our results, we assume the maximum gain of performance using outsourcing to one or two servers to be in the order of a magnitude.

From our point of view, technology in this area still evolves and we have to take into account, that modern smartphones and tablets still become more and more powerful. Furthermore, the programming environment offered by web technologies continuously evolves. Future research should target multi-threading capabilities in the browser for parallelization and eventually consider features from the Web Crypto API [21], still under development by the W3C.

Another interesting idea for future work consists in porting native code to JavaScript. The GNU multi-precision library (GMP) used in our server code has apparently been trans-compiled to JavaScript [1], due to lack of time we did not further investigate this path, however.

Parallelization could also play an important role on the server side. Batches with hundreds of modexps coming from a single client can easily be calculated by multiple threads, possibly even on multiple servers, when respecting required security properties.

As can be seen from this non-exhaustive enumeration, many opportunities for future projects remain and we are interested to hear of any future work conducted.

7.3. Personal Conclusion

In the introduction, I have described my experiences during this project as a fascinating journey into the unknown, and I will remember it as such. From time to time, however, it was also a rough journey. Battling for hours with renaming of variables in pseudo-code, obtaining stellar performance values which turn to ashes when revisited on the next morning and stumbling over subtle implementation details more often than not were to an equal amount part of this project, as has been the fascination described.

However, things always have to be like this and I am very happy to conclude my studies with this thesis, which to me, independent of its outcome, means a lot.

There are only minor things I would do differently on another occasion, I'll mention the two most important hindsights here. First of all, not having done a preliminary study in this area during the 'Project 2' module of my studies, proved to be a serious handicap for the first half of the project; and second, I'll have to remind my brain to stay more on track, even if research in a certain area opens up a plethora of interesting topics in adjacent disciplines.

So long, and thanks for all the fish.

Erklärung der Diplomandinnen und Diplomanden
Déclaration des diplômant-e-s

Selbständige Arbeit / *Travail autonome*

Ich bestätige mit meiner Unterschrift, dass ich meine vorliegende Bachelor-Thesis selbständig durchgeführt habe. Alle Informationsquellen (Fachliteratur, Besprechungen mit Fachleuten, usw.) und anderen Hilfsmittel, die wesentlich zu meiner Arbeit beigetragen haben, sind in meinem Arbeitsbericht im Anhang vollständig aufgeführt. Sämtliche Inhalte, die nicht von mir stammen, sind mit dem genauen Hinweis auf ihre Quelle gekennzeichnet.

Par ma signature, je confirme avoir effectué ma présente thèse de bachelor de manière autonome. Toutes les sources d'information (littérature spécialisée, discussions avec spécialistes etc.) et autres ressources qui m'ont fortement aidé-e dans mon travail sont intégralement mentionnées dans l'annexe de ma thèse. Tous les contenus non rédigés par mes soins sont dûment référencés avec indication précise de leur provenance.

Name/*Nom*, Vorname/*Prénom* .....

Datum/*Date* .....

Unterschrift/*Signature* .....

Dieses Formular ist dem Bericht zur Bachelor-Thesis beizulegen.
Ce formulaire doit être joint au rapport de la thèse de bachelor.

Glossary

Algorithm In context of this thesis, an algorithm is generally a set of steps to perform to achieve a certain goal. We use algorithms in mathematical sense, providing functions which lead to a certain goal, i.e. an efficient calculation of an exponentiation. Opposed to protocols, algorithms are only run by a single party alone.

API The Application Programming Interface is a set of functions provided by a software component (usually a library), which can be used by a programmer to make use of the functionality provided by the component.

HTTP Hypertext Transfer Protocol, a stateless transmission layer used for transferring data on the web. HTTP supports a set of methods (or 'verbs') for retrieving and uploading data from and to web servers.

JSON JSON, often called JavaScript Object Notation, is a standard format for transmitting data in a human readable form. It supports attribute-value pairs and is becoming increasingly popular, often replacing XML..

Modexp A modular exponentiation $c \equiv b^e \pmod{m}$. A detailed introduction to the topic is given in chapter 4 (p. 9).

POJO A POJO is a 'Plain Old Java Object', an object requiring no special class path and not being bound to any other restriction than the ones imposed by the Java Language Specification.

Primitive (Cryptographic) A cryptographic primitive is a building block which can be used by more advanced systems or protocols to achieve a certain functionality. Functions for encrypting or hashing data for instance are considered as primitives.

Protocol (Cryptographic) A cryptographic protocol can be considered as algorithm running between multiple parties. For instance, outsourcing a modular exponentiation to a server is a protocol with at least two parties, a client wishing to outsource the modexp and a server supporting the client. Protocols may lead to a result on their own or serve as primitive for other uses.

Secure Channel A secure channel is a well known concept in cryptography for transmitting information without possibility of overhearing or tampering. An adversary can not read any data exchanged over the channel and the recipient of the data can be confident that the data received matches exactly the data sent. Many standardized software packages nowadays implement secure channels, for instance TLS.

TLS Transport Layer Security [30] is the specification of one of the most widely used secure channels with many popular implementations.

URI The URI, or less general the URL is a so-called uniform resource identifier. A typical example is a web address like http://www.bfh.ch.

Bibliography

[1] "A port of the GNU Multiple-Precision Library (GMP), a library for arbitrary precision arithmetic, to JavaScript using Emscripten," https://github.com/kripken/gmp.js, accessed: 2017-01-16.

[2] "Apache Maven Project," https://maven.apache.org/, accessed: 2017-01-16.

[3] "Browserify," http://browserify.org/, accessed: 2017-01-16.

[4] "Chrome V8, Google's high performance, open source, JavaScript engine." https://developers.google.com/v8/, accessed: 2017-01-16.

[5] "Cobertura, A code coverage utility for Java." https://cobertura.github.io/cobertura/, accessed: 2017-01-16.

[6] "CommonJS, JavaScript module system," http://www.commonjs.org/, accessed: 2017-01-16.

[7] "ECMAScript 2015 Language Specification," http://www.ecma-international.org/ecma-262/6.0/, accessed: 2017-01-13.

[8] "Faster RSA in Java with GMP, A new java library that wraps libgmp," https://medium.com/square-corner-blog/faster-rsa-in-java-with-gmp-8b13c51c6ec4, accessed: 2017-01-16.

[9] "Istanbul - a JS code coverage tool written in JS," https://www.npmjs.com/package/istanbul, accessed: 2017-01-16.

[10] "Java API for RESTful Services (JAX-RS)," https://jax-rs-spec.java.net/, accessed: 2017-01-15.

[11] "JavaScript Big Integer Library by Leemon Baird," http://leemon.com/crypto/BigInt.js, accessed: 2016-10-01.

[12] "Jersey, RESTful Web Services in Java." https://jersey.java.net/, accessed: 2017-01-15.

[13] "JNA GMP project," https://github.com/square/jna-gmp/, accessed: 2017-01-16.

[14] "JSDoc, Documentation Strings for JavaScript," http://usejsdoc.org/, accessed: 2017-01-16.

[15] "node.js, JavaScript runtime built on the Chrome V8 JavaScript engine," https://nodejs.org/, accessed: 2017-01-16.

[16] "NPM, node package manager," https://www.npmjs.com/, accessed: 2017-01-16.

[17] "Project Grizzly, Java NIO based HTTP server," https://grizzly.java.net/, accessed: 2017-01-16.

[18] "The GNU Multiple Precision Arithmetic Library," https://gmplib.org/, accessed: 2017-01-16.

[19] "The MIT License," https://opensource.org/licenses/MIT, accessed: 2017-01-04.

[20] "Verificatum JavaScript Crypto library (VJSC)," http://www.verificatum.com/html/product_vjsc.html, accessed: 2017-01-16.

[21] "Web Cryptography API, W3C Proposed Recommendation 15 December 2016," https://www.w3.org/TR/WebCryptoAPI/, accessed: 2017-01-16.

[22] "Cross-Origin Resource Sharing,W3C Recommendation," https://www.w3.org/TR/cors/, January 2014, accessed: 2017-01-16.

[23] A.Essex, "Cryptographic End-To-End Verifiability for Real-World Elections," Ph.D. dissertation, University of Waterloo, Canada, 2012.

[24] S. Caarls, *E-Voting Handbook – Key Steps in the Implementation of E-Enabled Elections*, ser. GGIS (2010) 5 fin. E. Council of Europe Publishing, 2010.

[25] B. Cavallo, G. Di Crescenzo, D. Kahrobaei, and V. Shpilrain, *Efficient and Secure Delegation of Group Exponentiation to a Single Server*. Cham: Springer International Publishing, 2015, pp. 156–173. [Online]. Available: http://dx.doi.org/10.1007/978-3-319-24837-0_10

[26] X. Chen, J. Li, J. Ma, Q. Tang, and W. Lou, "New Algorithms for Secure Outsourcing of Modular Exponentiations," *IEEE Transactions on Parallel and Distributed Systems*, vol. 25, no. 9, pp. 2386–2396, Sept 2014.

[27] C. Chevalier, F. Laguillaumie, and D. Vergnaud, *Privately Outsourcing Exponentiation to a Single Server: Cryptanalysis and Optimal Constructions*. Cham: Springer International Publishing, 2016, pp. 261–278. [Online]. Available: http://dx.doi.org/10.1007/978-3-319-45744-4_13

[28] ——, "Privately Outsourcing Exponentiation to a Single Server: Cryptanalysis and Optimal Constructions," Cryptology ePrint Archive, Report 2016/309, 2016, http://eprint.iacr.org/2016/309.

[29] L. N. Childs, *A Concrete Introduction to Higher Algebra*. Springer Science+Business Media LLC, 2009.

[30] T. Dierks and E. Rescorla, "The Transport Layer Security (TLS) Protocol Version 1.2," RFC 5246 (Proposed Standard), Internet Engineering Task Force, Aug. 2008, updated by RFCs 5746, 5878, 6176, 7465, 7507, 7568, 7627, 7685, 7905, 7919. [Online]. Available: http://www.ietf.org/rfc/rfc5246.txt

[31] Eidgenössisches Finanzdepartement, "E-Government-Schwerpunktplan 2017-2019 verabschiedet," https://www.admin.ch/gov/de/start/dokumentation/medienmitteilungen.msg-id-64299.html, 2016, accessed: 2017-01-03.

[32] R. T. Fielding, "Architectural Styles and the Design of Network-based Software Architectures," Ph.D. dissertation, University of California, Irvine, 2000, https://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm.

[33] R. P. Gallant, R. J. Lambert, and S. A. Vanstone, *Faster Point Multiplication on Elliptic Curves with Efficient Endomorphisms*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2001, pp. 190–200. [Online]. Available: http://dx.doi.org/10.1007/3-540-44647-8_11

[34] R. Haenni, R. E. Koenig, and E. Dubuis, "Cast-as-Intended Verification in Electronic Elections Based on Oblivious Transfer," 2016.

[35] S. Hohenberger and A. Lysyanskaya, *How to Securely Outsource Cryptographic Computations*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 264–282. [Online]. Available: http://dx.doi.org/10.1007/978-3-540-30576-7_15

[36] H. L. Jonker, "Security Matters: Privacy in Voting and Fairness in Digital Exchange," Ph.D. dissertation, Eindhoven University of Technology and University of Luxembourg, 2009.

[37] J. Katz and Y. Lindell, *Introduction to Modern Cryptography, Second Edition*, 2nd ed. Chapman & Hall/CRC, 2014.

[38] M. S. Kiraz and O. Uzunkol, "Efficient and verifiable algorithms for secure outsourcing of cryptographic computations," *International Journal of Information Security*, vol. 15, no. 5, pp. 519–537, 2016. [Online]. Available: http://dx.doi.org/10.1007/s10207-015-0308-7

[39] S. Lang, *Undergraduate Algebra*, ser. Undergraduate Texts in Mathematics. Springer New York, 2005.

[40] X.-J. Lin, L. Sun, H. Qu, and X. Zhang, "New Approaches for Secure Outsourcing Algorithm for Modular Exponentiations," Cryptology ePrint Archive, Report 2016/045, 2016, http://eprint.iacr.org/2016/045.

[41] P. Locher and R. Haenni, "Verifiable Internet Elections with Everlasting Privacy and Minimal Trust," 2015.

[42] X. Ma, J. Li, and F. Zhang, "Outsourcing computation of modular exponentiations in cloud computing," *Cluster Computing*, vol. 16, pp. 787–796, 2013.

[43] A. J. Menezes, P. C. V. Oorschot, S. A. Vanstone, and R. L. Rivest, "Handbook of Applied Cryptography," 1997.

[44] P. Q. Nguyen, I. E. Shparlinski, and J. Stern, "Distribution of Modular Sums and the Security of the Server Aided Exponentiation," in *Proceedings of the Workshop on Comp. Number Theory and Crypt.*, 1999, pp. 1–16.

[45] E. G. Straus, "Problems and solutions: Addition chains of vectors," *American Mathematical Monthly*, vol. 71, pp. 806–808, 1964.

[46] Y. Wang, Q. Wu, D. S. Wong, B. Qin, S. S. M. Chow, Z. Liu, and X. Tan, *Securely Outsourcing Exponentiations with Single Untrusted Program for Cloud Storage*. Cham: Springer International Publishing, 2014, pp. 326–343. [Online]. Available: http://dx.doi.org/10.1007/978-3-319-11203-9_19

List of Figures

List of Tables

List of Algorithms and Protocols

APPENDICES

A. Installation and Usage

This chapter consists of two parts: First, we provide installation instructions for the famodulus system described in Chapter 5 (p. 29), and second, we outline how to use it.

For the installation, there are three main options to be considered:

- Using the famodulus bundle
- Installing components individually
- Custom deployments

The preferred way to install famodulus is by using the bundle, which is also the option that we will detail in this chapter. Another option would be to install all three components or only a subset of them individually and possibly on distinct machines. We will outline differences to the installation using the bundle where needed. Finally, the third option consists in custom deployments of the server parts (e.g. to Java EE containers) or in hosting the demonstrator application on a custom web server; we will not further detail these possibilities as they are not considered relevant to the typical usage of our software.

A.1. Installation

The installation of famodulus is done in three steps: installing necessary prerequisites, obtaining and installing the source code and testing the correct operation of the system. We will now describe these steps.

A.1.1. Prerequisites

famodulus has been programmed in Java and JavaScript, thus appropriate development environments for both languages are required for building and installing. Specifically, the tools listed in Table A.1 have to be installed.

Table A.1.: Build environment

Tool	Min. Version	URI
git[1]	-	`https://git-scm.com`
Java Development Kit (JDK)	1.8	`http://openjdk.java.net/projects/jdk8/`
Maven	3.0.5	`https://maven.apache.org`
node.js	6.9.1	`https://nodejs.org`

[1] Not a strict requirement if downloading release archives.

A.1.2. Installation

Obtaining the Source

All famodulus source code is maintained at github.com in different repositories. Each component of the system (client, server and demonstrator application) has its own repository and there is a superior repository for a bundle which includes all components. Table A.2 provides an overview of these repositories.

Table A.2.: famodulus repositories

Name	URI
famodulus bundle	`https://github.com/mainini/famodulus`
famodulus-server	`https://github.com/mainini/famodulus-server`
famodulus-client	`https://github.com/mainini/famodulus-client`
famodulus-demo	`https://github.com/mainini/famodulus-demo`

The simplest way to obtain the source is to clone the bundle repository directly using '`git`'. If this is not possible, archives containing individual releases may be downloaded at the URIs given in Table A.2. *Note that using the bundle with release archives is not possible; when using archives, all components must be downloaded separately and extracted to a common directory.*

Building and Installing

We will now continue the installation using the famodulus bundle. To clone the repository to a local directory called `famodulus`, issue the following command:

```
git clone https://github.com/mainini/famodulus.git
```

To complete the initialization of the repository, submodules have to be initialized and downloaded as well. This is achieved with the following commands:

```
cd famodulus
git submodule init
git submodule update
```

Sources are now set up and the build process can be started. famodulus-demo depends on famodulus-client, famodulus-server has no dependencies; this requires building of the client library before building the demonstrator application. We proceed as follows:

1. In the famodulus-client subdirectory, installing of dependencies and building the library is accomplished with the command '`npm install`'. This step creates the directory `.build`, which contains the library in `js/famodulus.browser.js` as well as the corresponding API documentation.
2. Changing to the famodulus-demo directory, the demonstrator application is compiled using the command '`mvn compile`'.
3. As last step, in the famodulus-server directory, the server is again built using '`mvn compile`'.

After these steps, the build process has been completed and the system can be tested, which will be detailed in the next section.

A.1.3. Testing

Extensive unit tests are provided with the client and the server, these can be used to check the correct operation of the installed components.

For famodulus-client, the unit tests can be run in two possible ways: either directly using node.js or in a browser of choice. To run the tests using node.js, run the command '`npm test`' in the client directory. By issuing the command '`npm run test-browser`', a temporary URI gets displayed, which, when accessed by any browser, runs the same tests again.

To test famodulus-server, unit tests can be run using maven in the server directory: '`mvn test`'

There are no automated tests for the famodulus-demo, refer to Section A.2 below for instructions on how to use it.

Code Coverage

The demonstrator application contains a link to code coverage documentation for the client library, which is not built by default. To build it, the following additional command has to be executed *before* compiling the demonstrator application in step 2 of the building instructions in the previous section: '`npm run coverage`'

Code coverage for the server can be generated as part of the maven site report, using the command '`mvn site`'. Refer to the maven documentation for further details.

Please be aware of the following limitations of the code coverage reports:

1. Not every generation of the library coverage report includes full branch coverage. This is due to certain branches in code having non deterministic reachability because they are dependent on random numbers being generated while running the tests. For all branches of famodulus-client, however, unit tests are provided.

2. The main class of the server application has low coverage because testing it would require running its main method from another class. Such a test has not been written because the code of the main class is trivial and will not fail without detection.

A.2. Usage

After having completed the installation as described in the previous section, a working famodulus system is now present and can be used for testing outsourcing of modular exponentiation and benchmarking performance. In this section, we cover the usage of the famodulus-demo application; using the library in own applications will not be discussed. Further details about using the library can be found in the documentation provided with the repositories and in the generated API documentation, details about the RESTful interface between client and server are given in Section 5.5 of Chapter 5 (p. 29).

A.2.1. Starting the System

If famodulus has been installed according to the instructions in this chapter, two server processes have to be started to use famodulus-demo: First, the web server of famodulus-demo itself, which only serves the website and corresponding static resources, and second, at least one running instance of famodulus-server for outsourcing calculations. Both can be run on the same machine or also be split to multiple separate machines. Starting the servers by using maven is the best option as it will properly adjust the Java class path amongst other things. The servers are started by running the command '`mvn exec:java`'. Exemplary output of starting both servers is given in Figure A.1 (p. 67).

Change Default Ports

If the servers are started without further options, they will listen to the default ports `8080` (famodulus-demo), respectively `8081` (famodulus-server). These can be adjusted using Java system properties, which can be set using environment variables. For instance, to start the demo web server on port `80` instead of `8080`, the following command may be used:

```
MAVEN_OPTS='-Dfamodulus.base=http://localhost:80/' mvn exec:java
```

Change Logging Level

Both servers use the default Java logging capabilities. In order to increase the logging level, the following lines may be appended either to local or system wide Java logging configuration:

```
ch.mainini.famodulus.level = FINE
org.glassfish.grizzly.level = FINE
```

Setting the level to `FINEST` will enable even more verbose logging.

Figure A.1.: Exemplary output of famodulus servers

```
famodulus-server$ mvn exec:java

[INFO] Scanning for projects...
[INFO]
[INFO] ------------------------------------------------------------------------
[INFO] Building famodulus-server 1.0.0
[INFO] ------------------------------------------------------------------------
[INFO]
[INFO] --- exec-maven-plugin:1.5.0:java (default-cli) @ famodulus-server ---
2017-01-13 15:22:40 INFO   ch.mainini.famodulus.server.Server main      Configuring and starting famodulus-server webserver...
2017-01-13 15:22:41 INFO   org.glassfish.grizzly.http.server.NetworkListener start      Started listener bound to [localhost:8081]
2017-01-13 15:22:41 INFO   org.glassfish.grizzly.http.server.HttpServer start      [HttpServer] Started.
2017-01-13 15:22:41 INFO   ch.mainini.famodulus.server.Server main      Webserver running at http://localhost:8081/ !
2017-01-13 15:22:41 INFO   ch.mainini.famodulus.server.Server main      Hit enter to stop it...

famodulus-demo$ mvn exec:java

[INFO] Scanning for projects...
[INFO]
[INFO] ------------------------------------------------------------------------
[INFO] Building famodulus-demo 1.0.0
[INFO] ------------------------------------------------------------------------
[INFO]
[INFO] --- exec-maven-plugin:1.5.0:java (default-cli) @ famodulus-demo ---
2017-01-13 15:33:32 INFO   ch.mainini.famodulus.demo.Server main      Configuring and starting famodulus-demo webserver...
2017-01-13 15:33:32 INFO   org.glassfish.grizzly.http.server.NetworkListener start      Started listener bound to [localhost:8080]
2017-01-13 15:33:32 INFO   org.glassfish.grizzly.http.server.HttpServer start      [HttpServer] Started.
2017-01-13 15:33:32 INFO   ch.mainini.famodulus.demo.Server main      Webserver running at http://localhost:8080/ !
2017-01-13 15:33:32 INFO   ch.mainini.famodulus.demo.Server main      Hit enter to stop it...
```

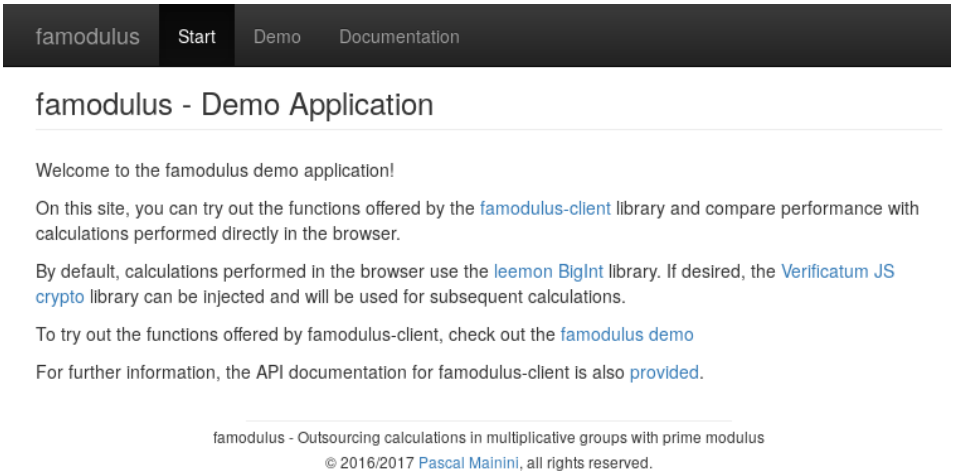
A.2.2. Working with famodulus-demo

With the servers installed and running, the famodulus-demo home page may now be accessed by pointing a web browser at the following URI:[1]

```
http://localhost:8080/
```

The homepage is depicted in Figure A.2.

Figure A.2.: famodulus-demo start page



The main navigation at the top of the page provides access to the following functions of the application:

- The home page
- The demo page, allowing to try out and benchmark famodulus-client
- Access to documentation

In the following section, we describe using the demo page for testing famodulus-client. The documentation page simply provides references to the API documentation as well as to the coverage report.

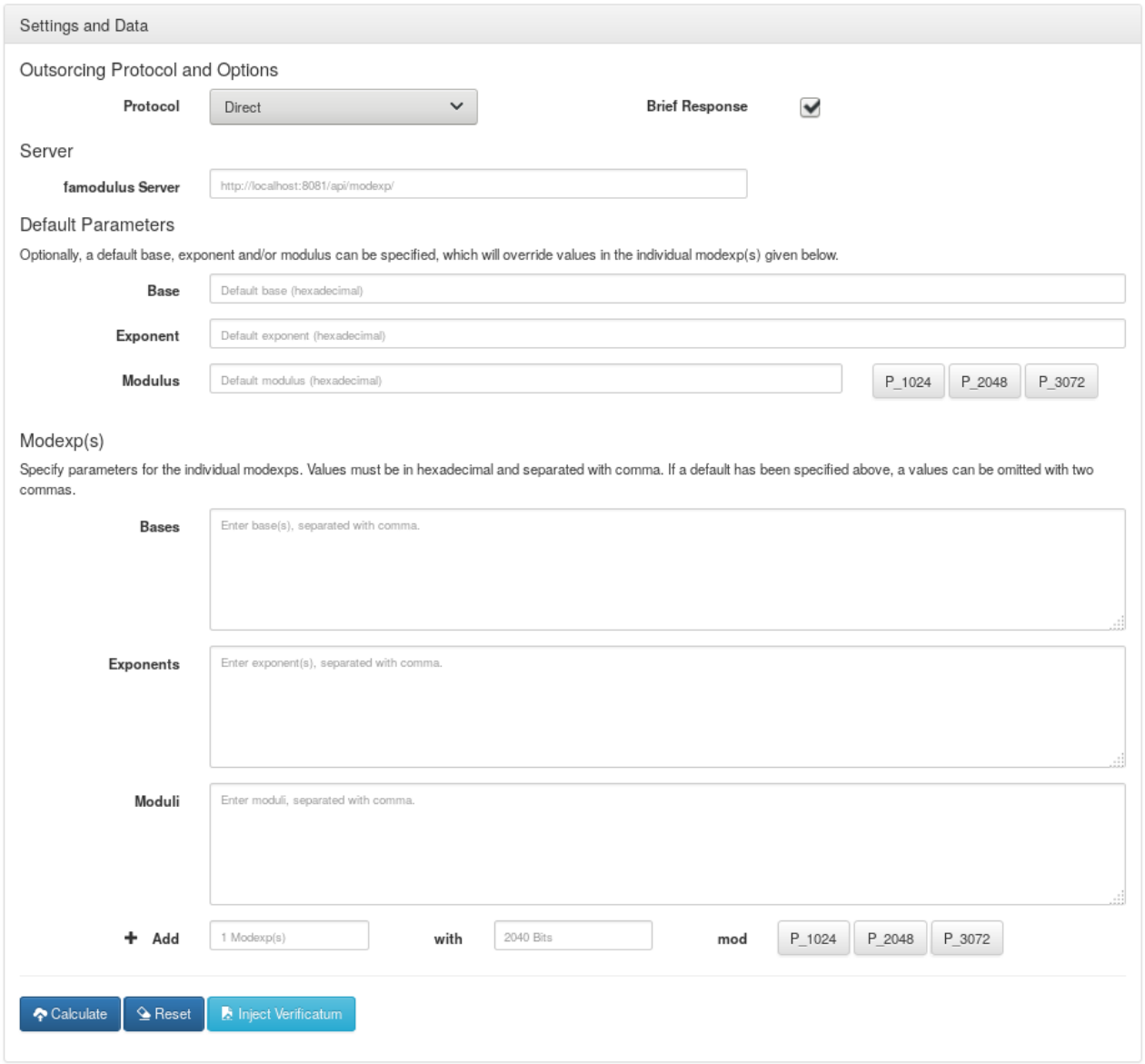
Entering and Calculating Modexps

After opening the demo page (`http://localhost:8080/demo.html`), the form depicted in Figure A.3 is displayed. This form offers access to all API functionality provided by famodulus-client and consists of four sections which we will now describe.

In the first section, the protocol used for outsourcing the modexp(s) can be chosen. Except for the protocol *Direct*, the protocols are described in Chapter 4 (p. 9). The Direct protocol is not a protocol per se; when used for outsourcing, all modexp(s) are simply sent directly to the server without any prior blinding or added checkability. It has mainly been added for testing the client library and

[1] famodulus is based on modern JavaScript (ECMAScript 6, [7]), thus we recommend using a decent browser version; our tests where conducted using Firefox 50.0.

Figure A.3.: famodulus-client demo form



the server as well as for performance measurement without any precalculations occurring on the client.

The *brief* option which can be toggled in the first section as well is part of the RESTful interface of the server. If disabled, the server will return long responses containing the full, original modexp data it has received. By default, it is enabled and the server will only return the results of the calculation, requiring the client to keep track of the order of the modexps sent to the server.

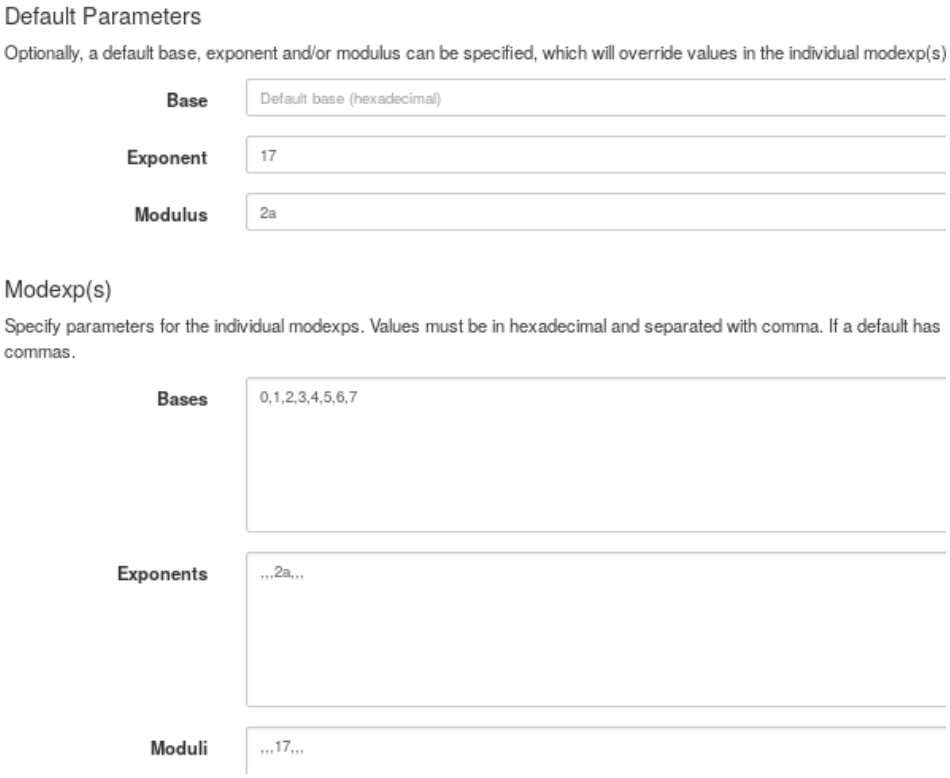
In the next section, URI(s) of the API endpoints of famodulus servers may be specified. Depending on the outsourcing protocol, one or two URI(s) are required.[2] The demo does not require the servers to be distinct, calculations may as well also be performed only on a single server (which would obviously not make sense in a practical scenario).

The last two sections provide fields to enter the parameters of the modexp(s) to calculate. The first three fields allow to specify defaults for base, exponent and modulus, which are applied in the case

[2] As default, `http://localhost:8081/api/modexp` is used.

that any of them is missing in the modexp(s) specified below. The following three large text areas take the parameters of one or more individual modexp(s) to calculate. All numbers have to be in hexadecimal format (also for the defaults) and multiple modexps must be separated using commas and optionally newlines. Figure A.4 gives an example for calculating eight modexps with exponent 0x17 (23 decimal) and modulus 0x2a (42 decimal), except for base 0x3, for which exponent and modulus are swapped.

Figure A.4.: Calculating 8 modexps with default parameters



There are six buttons which help setting up test parameters in the form. In the defaults section, the buttons *P_1024*, *P_2048* and *P_3072* will set the default modulus to a predefined prime number with 1024, 2048 or 3072 bits of size respectively. The same buttons in the modexps section below will add one or more modexp with the same prime exponents and with a random base and exponent each. The amount of modexps added as well as the size(s) in bits of base(s) and exponent(s) may be specified using the two fields in front of the buttons. By default, a single modexp with base and exponent of size 2040 bits is added.

When the definition of all parameters is complete, the calculation may be started using the *Calculate* button. The button *Reset* bring the form back to its initial state.

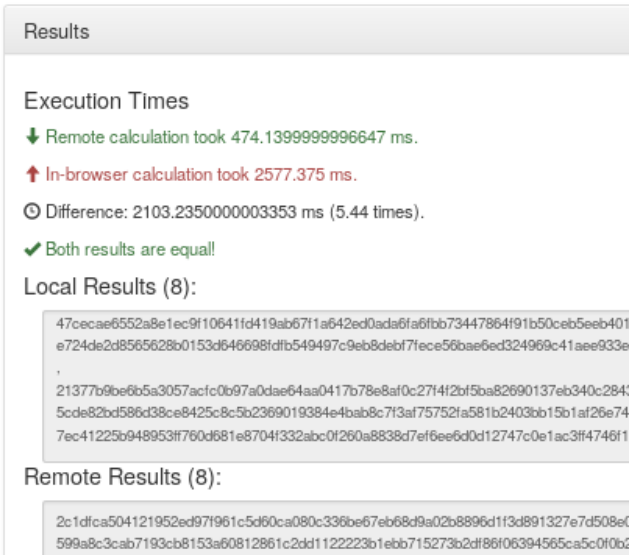
The last button, *Inject Verificatum*, dynamically injects the Verificatum JavaScript crypto library; the rationale for this is given in Section 5.3.2 of Chapter 5 (p. 29).[3] To revert calculations to the default implementation, the page has to be reloaded.

[3]The library is directly loaded from `http://www.verificatum.com/files/vjsc-1.1.0.js`.

Evaluating Results

After the calculations have been started by clicking on the *Calculate* button, two rounds of calculation take place. First, each modexp is calculated locally in the browser and the results are displayed in the results section of the demo. After that, the same calculation takes place on the server(s) by applying the selected outsourcing protocol. For both rounds, times are measured and a comparison is made when the results from the server are returned. An example of such a result is given in Figure A.5, see Chapter 6 (p. 39) for some actual benchmarks and a discussion.

Figure A.5.: Example of calculation results



B. Meetings and Decisions

This appendix gives a short summary of all meetings held during the project. It focuses mainly on decisions taken which were relevant for further work or the final result.

B.1. Meetings with Advisor

2016-09-19 Initial project kick-off.

Decisions:
- Create internal project repository
- Software has to be published using an open-source license

2016-10-03 Review of project plan.

Decisions:
- Organize meeting with expert for semester week 8/9
- No decisions on protocols yet, more work needed
- Simultaneous multi-exponentiation protocols are optional

2016-10-10 Devised initial version of comparison matrix.

Decisions:
- Exclusion of publications M4-S2, S4, M3 from scope
- S4 may be used for calculating inverses
- M3 is only listed as future option
- Recheck meeting with Expert

2016-10-27 Review of project plan and response from Ms. Chevalier, discussion of protocols, the RESTful interface, and programming languages.

Decisions: None

2016-11-07 Decisions:
- Start with default modPow()
- Implement DEC first
- Then GMP and eventually precomputation/Montgomery
- Then RANDEXP, M2, INV, RANDINV, S1

```
2016-11-24    Discussion Chapter 4, first implementation REST API and JS lib.

              Decisions:
              - Further development agile
              - Finish DEC for next meeting

2016-12-12    Demonstration of library implementation, review of protocols and
              report.

              Decisions:
              - We support only prime moduli ("mod p")
              - Priorities:
                1. DEC
                2. Other protocols according use cases
                3. Other server functions
                4. GMP
              - No further meeting currently

2017-01-12    Originally planned final meeting before presentation.
              Discussion of current state of work.

              Decisions:
              - Final priorities:
                1. short look at performance / GMP integration
                2. Additional algorithms (Hohenberger, Chevalier)
              - Report for advisor will be delivered as PDF
              - Report for expert also, I'll ask if paper version is desired

2017-01-17    Additional meeting for discussing the results of the performance
              analysis. Further tests conducted with a fix by R. Haenni.

              Decisions: None
```

B.2. Meetings with Expert

A single meeting with the expert took place on 2016-11-07. During this meeting, the idea and the goals of the project were presented. Also, the current state of work, especially regarding the research phase, has been given.

The expert did not request any further meetings until the defense of the thesis.