CHVote: Efficient modular exponentiations

Nicolas GAILLY

January 19, 2018

1 Introduction

The canton of Geneva is working on a new digital votation system called CHVote [6] whose main goal is to increase the electorate confidence by providing a secure, transparent and usable online voting system. The final product must enable a Swiss citizen to vote on any election mandated by the Geneva canton using his laptop or mobile phone in a browser environment.

In this work, we focus on the vote casting part of the system from the client side. In order to cast a vote, the client has to perform an k-out-of-n oblivious transfer protocol [1] with the votation servers. In this protocol, the client must perform between a few and a hundred of modular exponentiation computation, depending on the number of votes the client has to perform for a particular votation. In the context of CHVote, these modular exponentiation computations take place in a multiplicative group whose order is a large prime q. It is typically expected for security reason in this scenario that the number of bits needed to represent q can lie between 1024 bits and up to 8192 bits.

The potentially large number of computation on the client side is problematic in this context. Indeed, modular exponentiation with a large modulo is a computationally expensive operation. Typical optimizations in this space consists of hand-written assembly code such as the GMP library [2] which is composed of a mix of C and assembly. Typically, any developer that wishes to perform modulor exponentiation has to write the code in native Javascript or use a Javascript library. Javascript being an interpreted language running in the browser context performs quite poorly in comparison with native C/asm code [7].

In this work, we present an efficient way of computing these modular exponentations relying on multiple third party servers and evaluate this solution against the state-of-the-art methods available in the browser. This work presents our solution in section 2, our evaluation in section 3, and finally some open questions to be addressed in section 4.

2 Design

The goal of the project is to find an efficient scheme for the client to perform modular exponentiations. In the context of CHVote, the client only must know about the exponent; the base and the modulo need not to be private, simplifying significantly the design of the solution.

The core idea is to decentralize the expensive computation to third party servers while keeping the exponent secret by using a trivial non-threshold secret sharing scheme. The client computes the final result from the individual computations by using inexpensive modular *multiplication* operations.These servers are modeled as honest-but-curious adversaries and should ideally be independent and highly available.

We assume we are working in a multiplicative group G of order prime q. We denote the base $b \in \mathbb{Z}_q^*$ and the exponent $a \in \mathbb{Z}_q^*$. We want to find an efficient way to compute:

$$b^a \mod q$$

We call $S = \{s_1, \ldots, s_n\}$ the set of n servers selected to help the client with his computation. The protocol works as follow:

1. The client creates the n shares:

$$r_i = \textit{ random } \in \mathbb{Z}_q^* \text{ for } i = 0...n-2$$
$$r_{n-1} = a - \sum_{i=0}^{n-2} r_i \mod q$$

2. The client sends a computation request to each server s_i over a secure channel. The request sent to server i contains the share r_i, the base b and the modulo q.

3. Upon reception of a request, the server simply computes $v_i = b_i^r \mod q$ and sends the response v_i to the client.

4. The client waits to receive n responses and then computes the final result v as:

$$\begin{aligned} v &= \prod_{i=0}^{n-1} v_i \mod q \\ &= b^{\sum_{i=0}^{n-1} r_i} \mod q \\ &= b^{\sum_{i=0}^{n-2} r_i} * b^{r_{n-1}} \mod q \\ &= b^a \mod q \end{aligned}$$

This scheme allows an efficient outsourcing of the computation by allowing the client to use only inexpensive operations.

3 Evaluation

In order to be relevant, this proposed approach must be able to outperform the performance of the computation in a browser. In order to evaluate the system, we have designed a Javascript benchmark framework enabling different strategies to be evaluated. The code is available on Github [3].

The evaluation consists of comparing the time it takes to compute N modular exponentiations within the browser using Javascript (the "local" method), using WebAssembly [5], a recent effort to be able to run applications at a near native speed in the browser, and using the proposed system, the "split" method.

The Javascript code has been implemented using the JSBN library [8] which is the fastest in the mathematical open source libraries in Javascript available. Only three lines of code are needed to perform the computation using JSBN.

The WebAssembly code is ported from a C code base using the GMP library. The GMP library and the final C code have been compiled using the Emscripten framework [9]. Javascript Many optimizations have been used such making only one call from Javascript to WebAssembly for all computations and with optimizations flags turned on during the compilation.

The server code has been implemented as a web server using the Golang programming language [4]. The web server listens for incoming HTTP POST requests from the client containing the base, exponent and modulo. The web server then performs the computation using a binding to the GMP library [2] for good performance and sends back the results. The servers run on the same machine locally without any induced latency. It is probably safe to add an average 50ms of latency but this work does not do it since it does not influence the results.The format of the communication is done through JSON with hexadecimal encoding of the parameters. For this experiment, only three web servers have been deployed on the same machine.

The results can be found in the Figure **??** The proposed system clearly outperforms Javascript by an order of magnitude for any number of requested modular exponentations and for any modulus size. The only exception is with 1024 bits with a handful of modular exponentiation where the costs of going through the network outbalances the Javascript computation. However, it is highly discouraged to use 1024 bit keys since recent factorization techniques have been shown to crack such keys. Surprisingly, the WebAssembly's performance is worse than both other techniques. The compiled WebAssembly is only computing modexp and have been optimized with -O3 and the Javascript code copies all data to the WebAssembly stack before calling it. Even then, the costs of repeated calls is too high but is not surprising. Another potential reason might be that for compiling GMP to WebAssembly it is required to disable assembly coded parts, which provide the full power of GMP. Resorting to regular C algorithm significantly lowers the advantage of using GMP. Moreover, WebAssembly is still a recent technology that may not be supported on many majors outdated browsers.

We also note that there are fluctuations of the "split" method with a 1024

bit key size. We explain those fluctuations due to the fact that the experiment is ran over a single laptop while other processes are also active and it is very hard to do micro-benchmarking accurately and reliably in Javascript due to the interpreted nature of Javascript. The relatively short time of computation makes it difficult to get reliable measurements.

4 Discussions

We discuss in this section some open questions and some hints for future work.

1. **Threat model:** One assumptions the system makes is that the server are honest-but-curious. If we want to relieve that assumption, the system needs a *recovery* mechanism for the client to still be able to vote. Some solutions may include the following:

   - **Computing natively:** One solution can simply be to perform the computation using the native Javascript engine as a fallback mechanism. The client must be aware that the computation went wrong so he has an explicit reason to wait longer for the computation to finish.
   - **Invidividual verifiability:** Another solution can be to require each servers to provide a valid NIZK proof of exponentiation. In this case, this proof can be instantiated as a Schnorr signature using r_i as the private key and b^{r_i} as the public key.

5 Conclusion

We presented a system decentralizing an expensive computation to lightweight servers using the GMP computational library. The system shows a significant gain in performance compared to a native Javascript implementation or a WebAssembly compiled version and is therefor, a good target for the CHVote system.

References

[1] Cheng-Kang Chu and Wen-Guey Tzeng. *Efficient k-Out-of-n Oblivious Transfer Schemes with Adaptive and Non-adaptive Queries*, pages 172–183. Springer Berlin Heidelberg, Berlin, Heidelberg, 2005.

[2] Free Software Foundation. The gnu multiple precision arithmetic library. `https://gmplib.org/`, 1993-2016.

[3] Nicolas Gailly. Outsourcing modular exponentiation. `https://github.com/dedis/paper_17_geneva`, 2017.

[4] Google and open source contributors. Golang. https://golang.org/, 2007.

[5] Andreas Haas, Andreas Rossberg, Derek L Schuff, Ben L Titzer, Michael Holman, Dan Gohman, Luke Wagner, Alon Zakai, and JF Bastien. Bringing the web up to speed with webassembly. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 185–200. ACM, 2017.

[6] Rolf Haenni, Reto E. Koenig, Philipp Locher, and Eric Dubuis. Chvote system specification. Cryptology ePrint Archive, Report 2017/325, 2017. http://eprint.iacr.org/2017/325.

[7] StackOverflow. why-does-this-v8-javascript-code-perform-so-badly. https://stackoverflow.com/questions/7025286/why-does-this-v8-javascript-code-perform-so-badly, 2016.

[8] Tom Wu. jsbn: a fast, portable implementation of large-number math in pure javascript. https://github.com/andyperlitch/jsbn, 2013.

[9] Alon Zakai. Emscripten: an llvm-to-javascript compiler. In *Proceedings of the ACM international conference companion on Object oriented programming systems languages and applications companion*, pages 301–312. ACM, 2011.

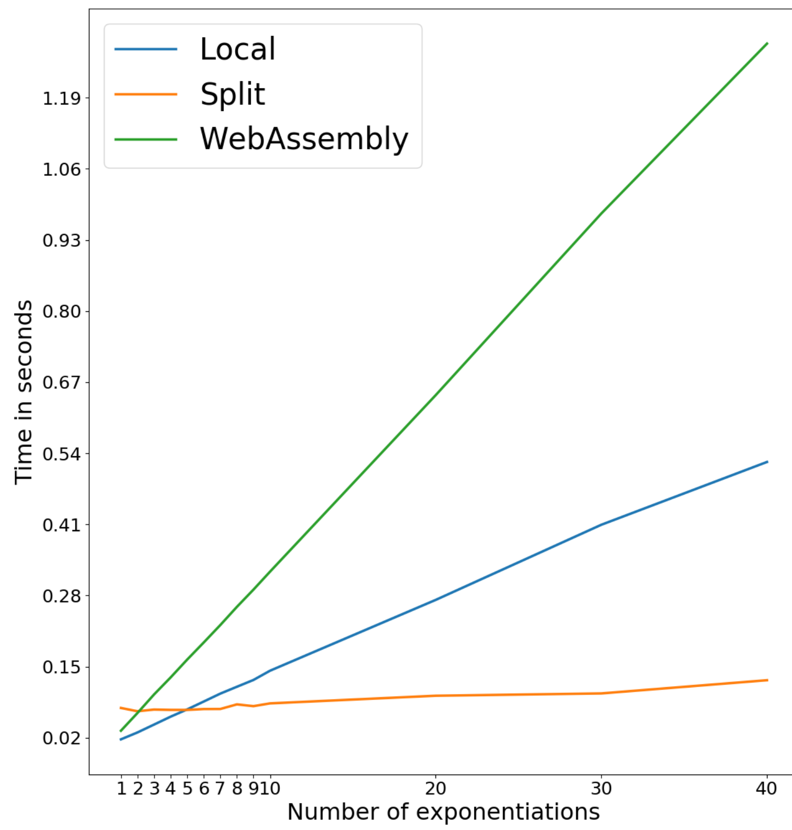

Figure 1: 1024 bits key size

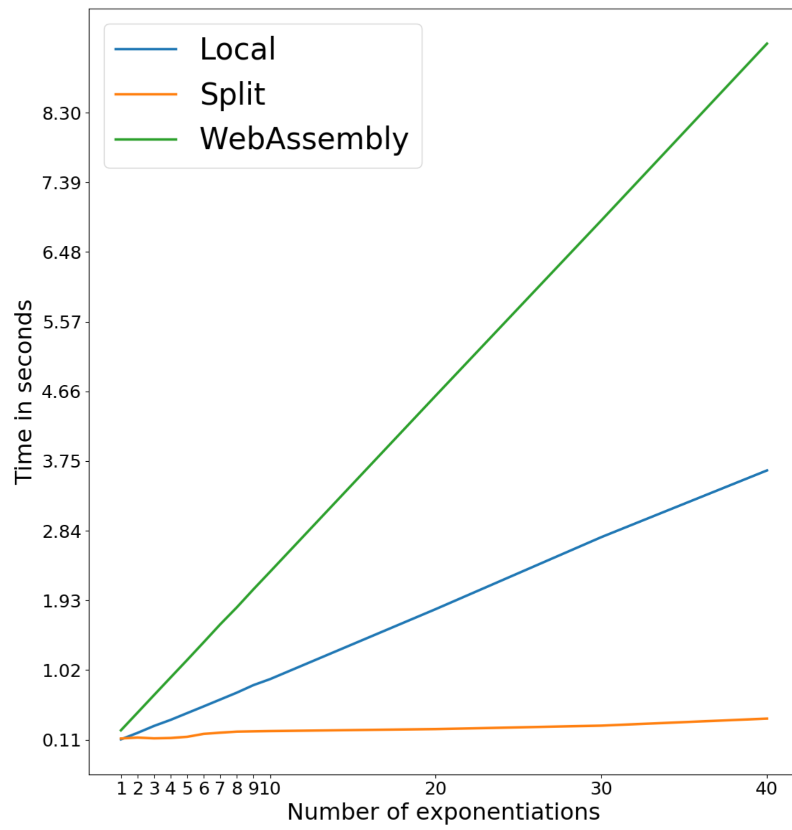

Figure 2: 2048 bits key size

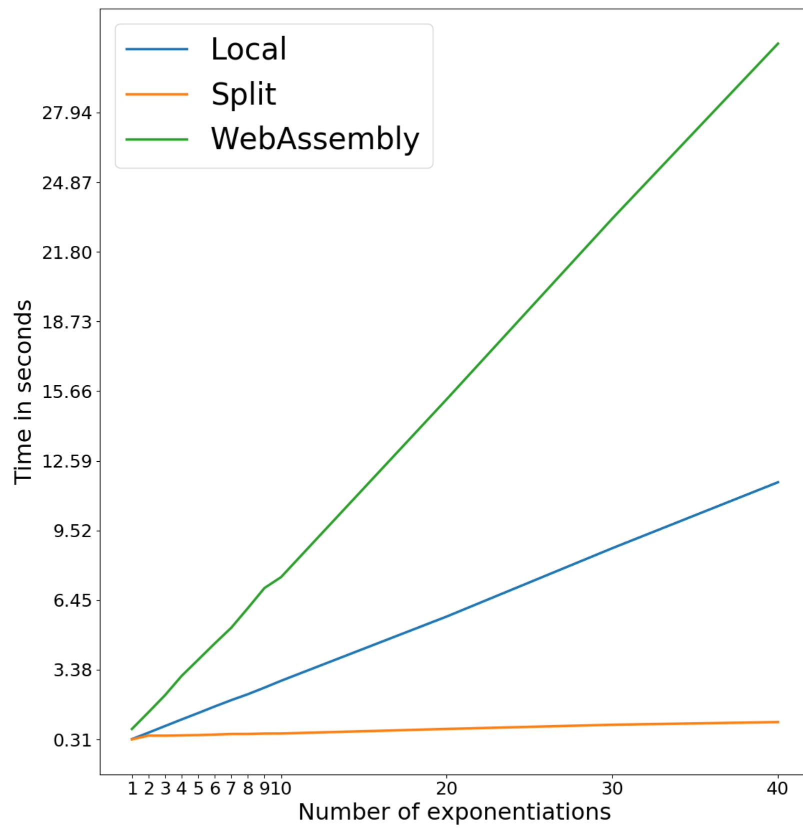

Figure 3: 4096 bits key size