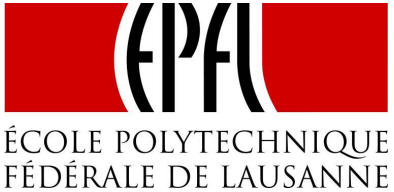


Modular Exponentiation

In the browser !?

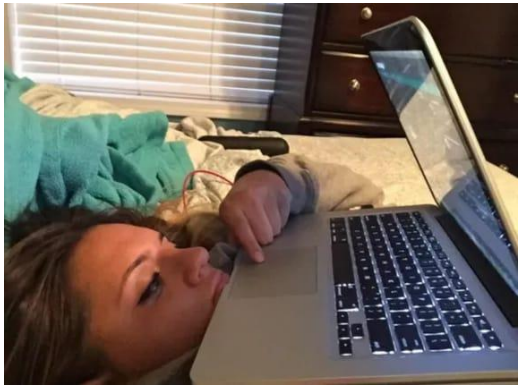
P.h.D. semester project, 2017.
Supervised by Bryan Ford (DEDIS) and Thomas Hofer (DGSI).

Background

The digital world takes an overwhelming part in our daily life.

Voting is still paper-based and requires physical presence...

Can we make people vote from their bed in a secure way ??



CHVote

Geneva is developing a next-gen voting solution for its canton: CHVote.

Lot of people living abroad are expected to use the solution.

Full formal specifications written by people from the e-Voting group, RISIS, in BFH.

Implementation in progress by the DGSI (“Direction générale des systèmes d'information”)

CHVote

Berner Fachhochschule
Haute école spécialisée bernoise
Bern University of Applied Sciences

Encrypted vote in the browser...

Secure voting requires encryption of the vote at the client's side

- Up to hundreds of votes to encrypt for one client

RSA encryption uses modular exponentiation with 1024,2048 or 4096 bit keys.

Modular exponentiation is a **slow** operation.

Modular exp. in Javascript ?

Javascript is an interpreted language and runs in the browser

- It it *not* fast
- Garbage collected
- Not to mention all the security issues...

Nevertheless, a better choice than sending

a vote in the clear!

What can we do ?

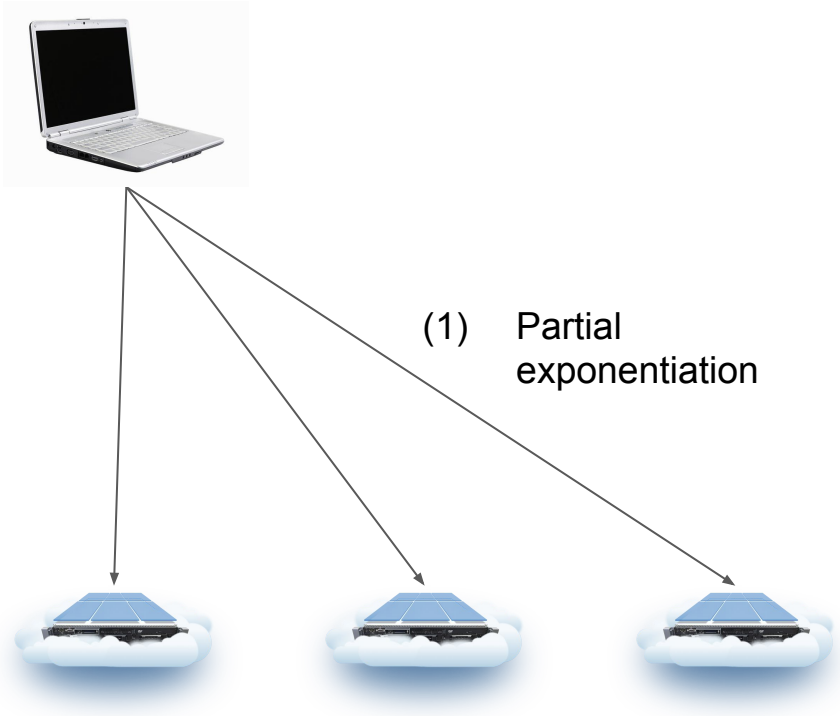
Outsource the heavy computation to remote servers (honest-but-curious).

In this context:

- Base is the public key so it is public
- Exponent is private (encoded vote)
- Modulo is public (security parameter)

Partial exponentiation request &

Fast reconstruction locally with *multiplication*



What can we do ?

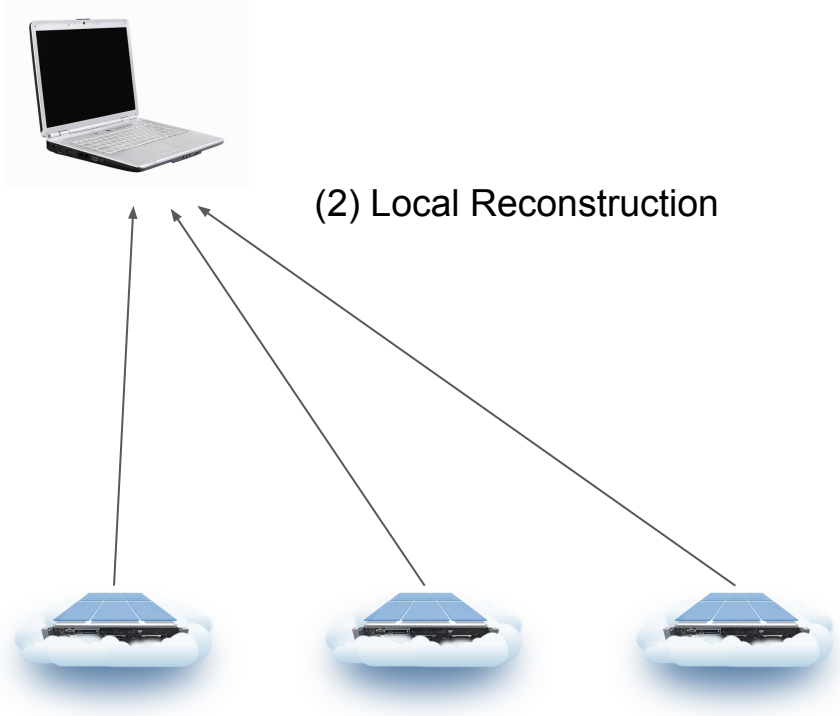
Offload the heavy computation to remote servers (honest-but-curious) !

In this context:

- Base is the public key so it is public
- Exponent is private (encoded vote)
- Modulo is public (security parameter)

Partial exponentiation request &

Fast reconstruction locally with *multiplication*



What can we do ?

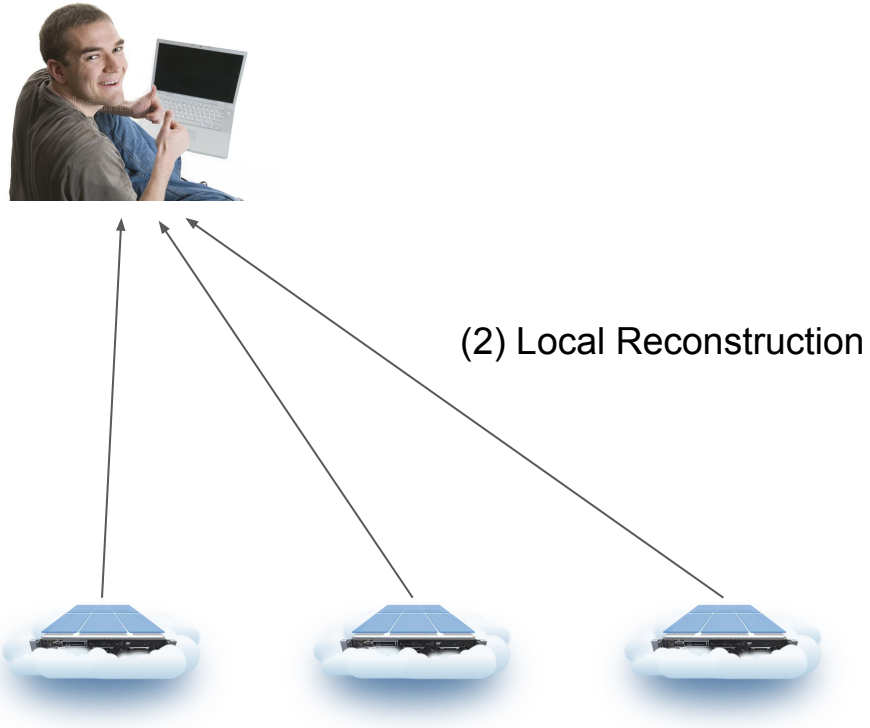
Offload the heavy computation to remote servers (honest-but-curious)!

In this context:

- Base is the public key so it is public
- Exponent is private (encoded vote)
- Modulo is public (security parameter)

Partial exponentiation request &

Fast reconstruction locally



Splitting the computation

Partial exponentiation:

v = <vote>
a = <public key>
q = <modulo>

s_i = <random> (i: 0...n-1)
s_n = v - SUM(s_i) (i: 0 … n-1)

Each server i computes:

r_i = a^(s_i) mod q

Splitting the computation

Partial exponentiation:

v = <vote>
a = <public key>
q = <modulo>

s_i = <random> (i: 0...n-1)
s_n = v - SUM(s_i) (i: 0 … n-1)

Each server i computes:

r_i = a^(s_i) mod q

Local Reconstruction:

e = <encrypted vote>

e = MUL(r_i) (i: 0 … n)

= a ^ (SUM(s_i)) mod q

= a ^ [SUM(s_i) + v - SUM(s_i)]

= a ^ v mod q

Evaluation:

Comparison between:

- **Pure Javascript**
- Split method
- WebAssembly

- Using JSBN library from Tom Wu at Stanford (fastest library ?)
- Simple one line of code...

```
run(deferred) {
    var res = null;
    for(var count = 0; count < this.copies; count++) {
        res = this.base.modPow(this.exp,this.mod);
    }
    deferred.resolve();
}
```

Evaluation:

Comparison between:

- Pure Javascript
- **Split method**
- WebAssembly

- Front end in JS (share splitting + JSON encoding)
  - ~50 lines
- Backend in Go using binding to GMP
  - Less than 100 lines
- Optimized to send the minimum amount of data

```
var response = &Response{Results: make([]string, len(request.Modexps))}
modulo := atoi(request.Modulo)
for i, req := range request.Modexps {
    base := atoi(req.Base)
    exp := atoi(req.Exp)
    result := computeExponentiation(base, exp, modulo)
    response.Results[i] = itoa(result)
}

if err := json.NewEncoder(w).Encode(response); err != nil {
    http.Error(w, "failed", http.StatusInternalServerError)
    return
}
```

Evaluation:

Comparison between:

- Pure Javascript
- Split method
- **WebAssembly**

- Compiled GMP to Wasm in 32 bit
  - Using LLVM 32 bit
  - Without assembly code :(
- Small wrapper in C for mod. Exp.
- Copy data to Wasm heap from JS
  - All in one call

```
run(deferred) {
    var nbData = this.copies * 2 + 1;
    var dataPtr = allocate(4*nbData,'i32',ALLOC_NORMAL);

    var allPtrs = new Uint32Array(this.copies * 2);
    Module.setValue(dataPtr + 4 * 1,this.i2wasm(this.mod.toString(16)),"i32");
    for(var i =0; i < this.copies; i++) {
        // first the base
        var baseIdx = i*2;
        allPtrs[baseIdx] = this.i2wasm(this.base.toString(16));
        Module.setValue(dataPtr + 4 * baseIdx,allPtrs[baseIdx],"i32");
        // then the exp
        var expIdx = (i*2) + 1;
        allPtrs[expIdx] = this.i2wasm(this.exp.toString(16));
        Module.setValue(dataPtr + 4 * expIdx,allPtrs[expIdx],"i32");
    }

    // copy modulo as last element
    var modPtr = this.i2wasm(this.mod.toString(16));
    Module.setValue(dataPtr + 4 * (nbData-1),modPtr,"i32");

    wasmModExpMatrix(dataPtr,nbData)

    for(var i = 0; i < this.copies; i++) {
        Module._free(allPtrs[i*2]);
        Module._free(allPtrs[(i*2)+1]);
    }
    Module._free(dataPtr.byteOffset);
    deferred.resolve()
}
```

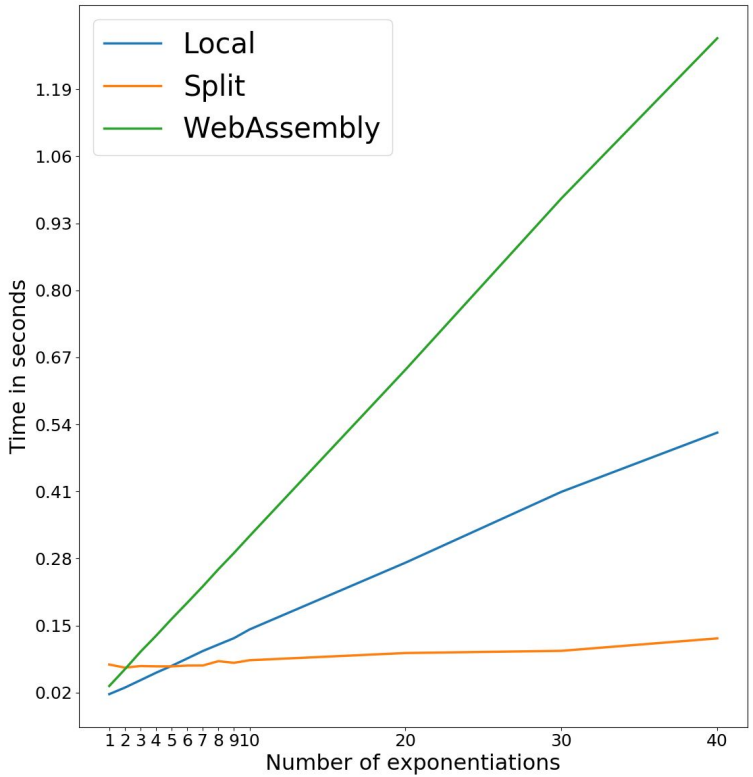
Results:

Comparison between:

- Pure Javascript
- Split method
- WebAssembly

For different key sizes:

- **1024 bits**
- 2048 bits
- 4096 bits



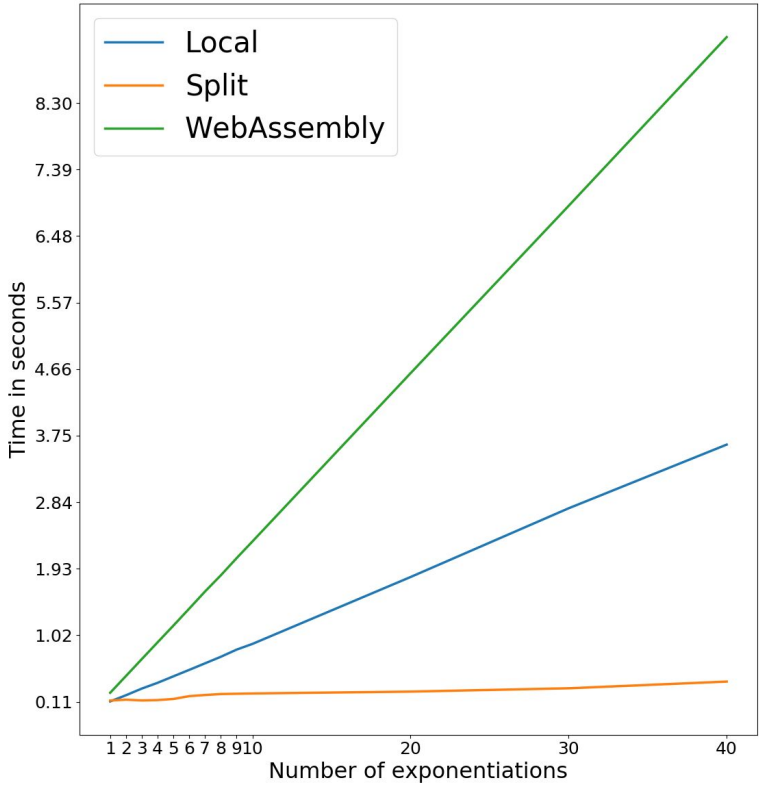
Results:

Comparison between:

- Pure Javascript
- Split method
- WebAssembly

For different key sizes:

- 1024 bits
- **2048 bits**
- 4096 bits



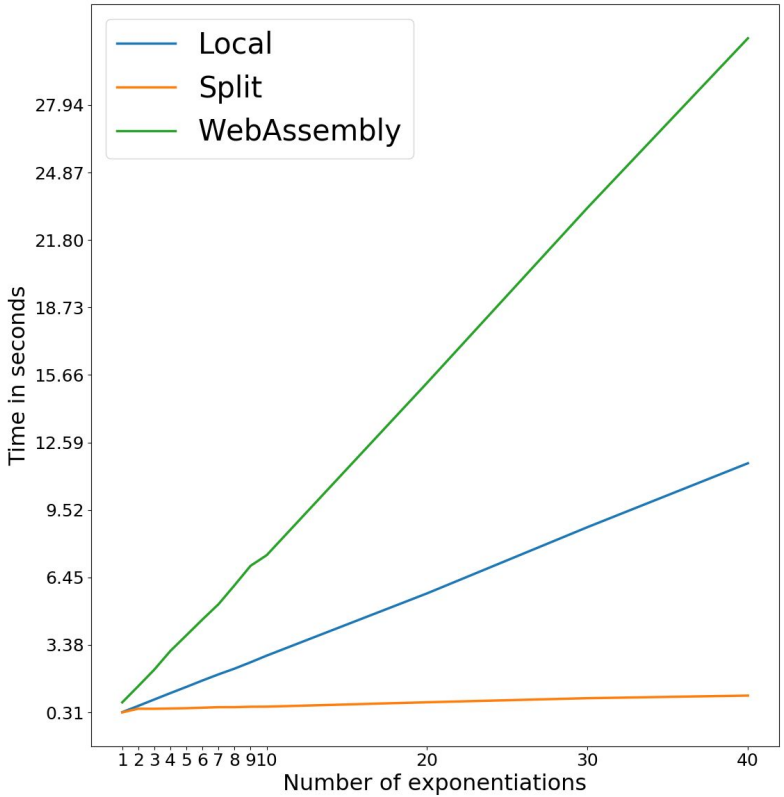
Results:

Comparison between:

- Pure Javascript
- Split method
- WebAssembly

For different key sizes:

- 1024 bits
- 2048 bits
- **4096 bits**



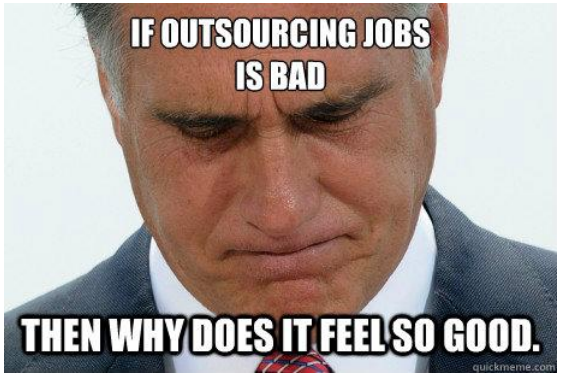
Future work

- Look at verifiable computation (NIZK)
  - Is it possible ?
  - Is it expensive ?
  - Look at recent progress such as “**CExp: secure and verifiable outsourcing of composite modular exponentiation with single untrusted server “** (Shuai Li)
- Code optimized hand-written WebAssembly code for modular exponentiation
- Experience with a varying number of servers (3 so far)

Conclusions

Outsourcing the heavy computation is good **in this context**

- Performs an order of magnitude better than other solutions
- No need for verification of correct output

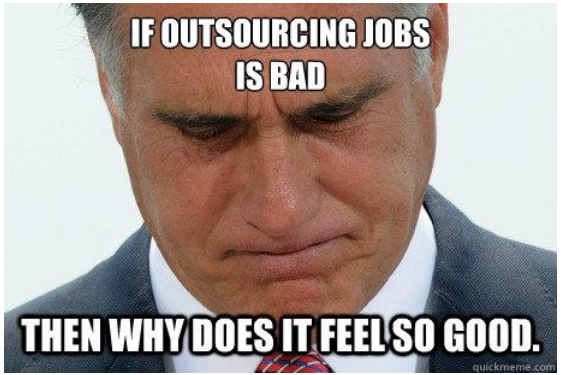


Conclusions

https://github.com/dedis/students_17_geneva

Outsourcing the heavy computation **IS** good in this context

- Performs an order of magnitude better than other solutions
- No need for verification of correct output



WebAssembly is not ready for prime time **yet**.

- Performs much better in an infinite loop (graphics)
- Compiles only in 32 bit
- Can’t compile hand-written assembly