Elliptic Curve Cryptography in JavaScript

Haustenne Laurie - De Neyer Quentin - Pereira Olivier
Université catholique de Louvain

ECRYPT Workshop on Lightweight Cryptography
November 28-29, 2011, Louvain-la-Neuve, Belgium

Context

Motivation

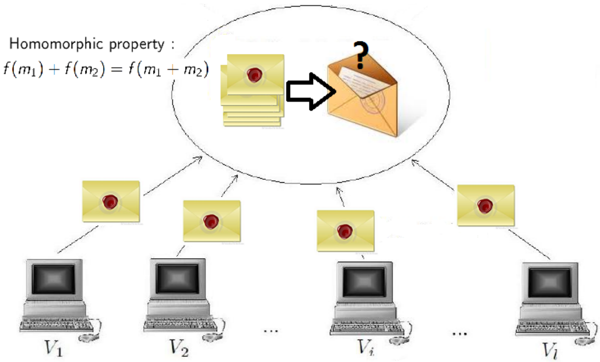
- Motivation: privacy preserving operations in browser
  - Ex: vote, password management, multiparty computation ...
- Browser status:
  - SSL/TLS offer secure channels, but nothing more
  - cryptographic libraries not available to web applications
- One single possibility for implementing cryptographic applications in any browser: JavaScript

JavaScript

- JavaScript engine provided in all major browsers
- Increasingly used in other contexts
  - PDF
  - OpenOffice
  - Node.js
- Problem : despite recent improvements of browsers, JS remains very slow

Application

Motivating application : Helios votins system [CGS97]



$$(\alpha P, \alpha Q + Q, wP, wQ, (r_2 + \alpha d_2)P, (r_2 + (\alpha + 1)d_2)Q, d_2, c - d_2, d_1, w - \alpha d_1)$$

Application constraints

- Adapt CGS on elliptic curves
  $\longrightarrow$ allows working with smaller field elements
- Only two base points
  $\longrightarrow$ suggests the use of precomputation
- Large bandwidth available in web applications
  $\longrightarrow$ allows precomputation on the server side

Strategy

Adopted strategy

Starting point: jsbn.js (prime fields library by Tom Wu)

Experimenting at two levels:

- finite fields arithmetic
  - improve arithmetic on prime fields (jsbn)
  - test binary fields and OEF's
  - test different field multiplication methods (Karatsuba, accumulation)
- EC arithmetic
  - design efficient EC point multiplication

Karatsuba

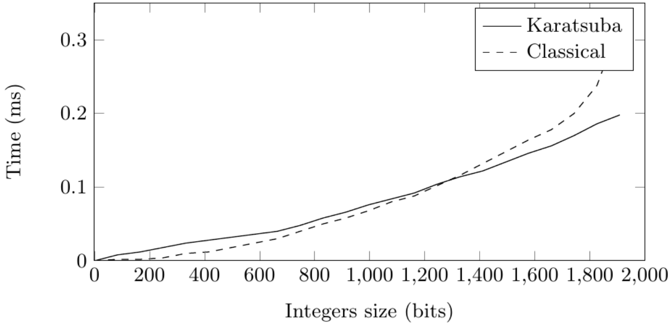
$$ab = (a_1 2^k + a_0)(b_1 2^k + b_0)$$

Classic : $ab = a_1 b_1 2^{2k} + (a_1 b_0 + a_0 b_1)2^k + a_0 b_0 \rightarrow \mathbf{4}$

Karatsuba : $ab = \underbrace{a_1 b_1}_{1} 2^{2k} + (\underbrace{(a_1 + a_0)(b_1 + b_0)}_{2} - a_1 b_1 - \underbrace{a_0 b_0}_{3})2^k + a_0 b_0 \rightarrow \mathbf{3}$

Multiplication methods

- Divide and conquer method : Karatsuba



- Accumulation strategy: efficient for OEF's but not for primes

Prime fields

Example of field element

$a = 26662129772183233595804137767533742264566028071077889952350268396785$

- Multiplication
  - **classic** with wordsize = 28 bits
  - with accumulation
  - Karatsuba : efficient for very large numbers
- Reduction
  - NIST primes designed for 32-bit architecture
  - idea : work with primes for optimal reduction on 28 bits
    $\rightarrow p = 2^{224} + 2^{140} + 2^{56} + 1$: **very efficient**

Binary fields

Example of field element

$a = \mathbf{1}z^{224} + \mathbf{0}z^{223} + \mathbf{1}z^{222} + \ \dots + \mathbf{0}z^{2} + \mathbf{1}z^{1} + \mathbf{1}$

- Squaring : linear complexity
- Multiplication implies many bit shifts : not efficient in software $\rightarrow$ poor performance

OEF's

Example of field element

$a = 16776211z^9 + 15356032z^8 + 13984561z^7 + \ldots + 11579833z^2 + 4567390z + 14375908$

- Choice of the parameters
- Multiplication
  - classic
  - **with accumulation**

Choice of coordinates

- Precomputation is made on the server side
  $\longrightarrow$ choose coordinate system to optimize on line computation
  $\longrightarrow$ avoid on-line inversions that are too expensive

Doubling		General addition		Mixed coordinates	
$2A \rightarrow A$	$1I, 2M, 2S$	$A + A \rightarrow A$	$1I, 2M, 1S$	$J + A \rightarrow J$	$8M, 3S$
$2P \rightarrow P$	$7M, 3S$	$P + P \rightarrow P$	$12M, 2S$	$J + C \rightarrow J$	$11M, 3S$
$2J \rightarrow J$	$4M, 4S$	$J + J \rightarrow J$	$12M, 4S$	$C + A \rightarrow C$	$8M, 3S$
$2C \rightarrow C$	$5M, 4S$	$C + C \rightarrow C$	$11M, 3S$		

- Optimum is reached when precomputation is stored in affine
  $\longrightarrow$ mixed Jacobian-Affine addition
  $\longrightarrow$ Jacobian doubling

- If precomputation was made on the client side, different choices should be made

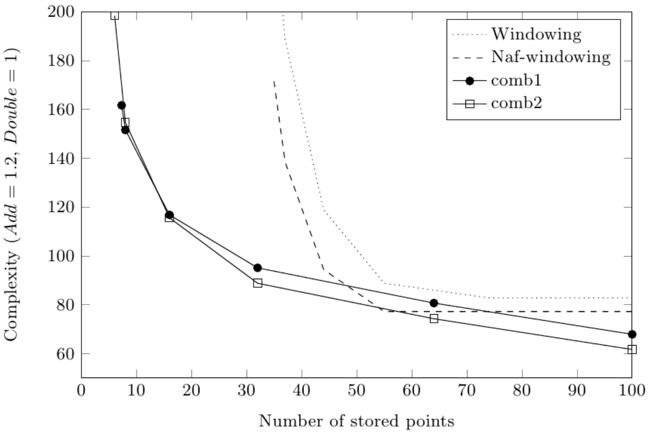
Point multiplication

- Multiplication
  - $kP = \sum_{i=1}^{n} k_i 2^i P$

    $\rightarrow$ Double-and-add
- Multiplication with precomputation
  - naive : precompute $2^i P$ for $i = 1, 2, 3, ..., n$
  - but clever methods exist...

Point multiplication methods

- Complexity study with precomputation



Results

Tests modalities

- Computer : Intel Core 2 Solo processor SU3500 (1.4 GHz, 800 MHz FSB)
- OS : Windows Vista
- Browsers
  - FFX : Mozilla FireFox 4.0.1
  - IEX : Internet Explorer 9.0.1
  - CHR : Google Chrome 11.0.696.71
  - SAF : Safari 5.0.5

Results

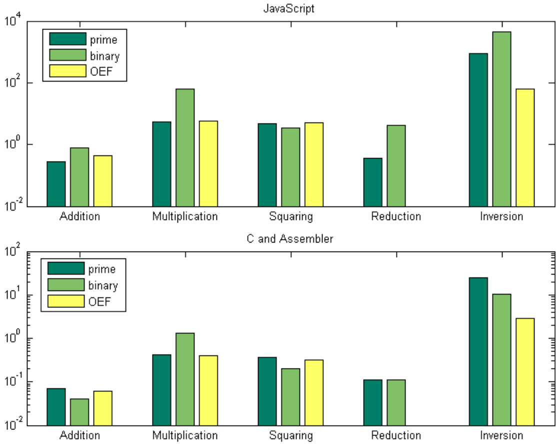
- Satisfactory timings per candidate for EC CGS (*ms*)

		Prime Fields				OEF's			
	M	FFX	IEX	CHR	SAF	FFX	IEX	CHR	SAF
Ballot Construction	2	8.8	7	5.3	9.7	7	11.7	9.9	10.3
Validity Proof	6	19.8	22	15	29	21.5	41.3	28.7	31.2
Total per candidate	8	**34.1**	**29.4**	**20.4**	**39**	**28.4**	**57.7**	**39.2**	**41.4**

- Voting time is linear in n (# candidates)

Comparison I

- Comparison JavaScript(FFX) - other implementation (μs)
  Similar trends



Comparison II

- Most recent comparison with jsbn of EC point multiplication on prime fields with Chrome 14 on Intel Core $i7-640M$ Processsor at 2.8GHz

	UCL	jsbn
EC mult (μs)	550	30000

- Acceleration factor of 50 due to:
  - dedicated modulus
  - precomputation
  - code improvement

Future works

Future Works

- Enlarging possibilities
  - Mixnet solution
  - Point multiplication without precomputation
  - Different security levels
- Speeding up
  - Code improvement
  - Testing other curves
- Ensuring security
  - Randomness source


MERCI!
THANK YOU!
FRAPAR.