Bern University
of Applied Sciences

Improving the Performance of Cryptographic Voting Protocols

Rolf Haenni, Philipp Locher, Nicolas Gailly

Voting'19 @ FC'19, St. Kitts, February 22, 2019

Outline

- Introduction
- Modular Exponentiation
- Product Exponentiation Algorithms
- Fixed-Based Exponentiation Algorithms
- Application to Shuffle Proof
- Conclusion

Motivation 1: CHVote Vote Casting

Receiver		Sender
knows $\mathbf{s} = (s_1, \ldots, s_k)$		knows $\mathbf{m} = (M_1, \ldots, M_n)$
for $j = 1, \ldots, k$ – pick random $r_j \in_R \mathbb{Z}_q$ – compute $a_{j,1} = \Gamma(s_j) \cdot g_1^{r_j}$ – compute $a_{j,2} = g_2^{r_j}$ – let $a_j = (a_{j,1}, a_{j,2})$	$\mathbf{a} = (a_1, \ldots, a_k)$ $\longrightarrow$	
		compute $\mathbf{b}$, $\mathbf{C}$, d
	$\mathbf{b} = (b_1, \ldots, b_k)$, $\mathbf{C} = (C_{ij})_{n \times k}, d$ $\longleftarrow$	
for $j = 1, \ldots, k$ – compute $k_j = b_j \cdot d^{-r_j}$ – compute $M_{s_j} = C_{s_j,j} \oplus h(k_j)$		

Motivation 1: CHVote Vote Casting

Receiver		Sender
knows $\mathbf{s} = (s_1, \ldots, s_k)$		knows $\mathbf{m} = (M_1, \ldots, M_n)$
for $j = 1, \ldots, k$ – pick random $r_j \in_R \mathbb{Z}_q$ – compute $a_{j,1} = \Gamma(s_j) \cdot g_1^{r_j}$ – compute $a_{j,2} = g_2^{r_j}$ – let $a_j = (a_{j,1}, a_{j,2})$	$\mathbf{a} = (a_1, \ldots, a_k)$ $\longrightarrow$	
		compute $\mathbf{b}$, $\mathbf{C}$, d
	$\mathbf{b} = (b_1, \ldots, b_k)$, $\mathbf{C} = (C_{ij})_{n \times k}, d$ $\longleftarrow$	
for $j = 1, \ldots, k$ – compute $k_j = b_j \cdot d^{-r_j}$ – compute $M_{s_j} = C_{s_j,j} \oplus h(k_j)$		

Motivation 1: CHVote Vote Casting

Receiver		Sender
knows $\mathbf{s} = (s_1, \ldots, s_k)$		knows $\mathbf{m} = (M_1, \ldots, M_n)$
for $j = 1, \ldots, k$ – pick random $r_j \in_R \mathbb{Z}_q$ – compute $a_{j,1} = \Gamma(s_j) \cdot g_1^{r_j}$ – compute $a_{j,2} = g_2^{r_j}$ – let $a_j = (a_{j,1}, a_{j,2})$	$\mathbf{a} = (a_1, \ldots, a_k)$ $\longrightarrow$	
		compute $\mathbf{b}$, $\mathbf{C}$, d
	$\mathbf{b} = (b_1, \ldots, b_k)$, $\mathbf{C} = (C_{ij})_{n \times k}, d$ $\longleftarrow$	
for $j = 1, \ldots, k$ – compute $k_j = b_j \cdot d^{-r_j}$ – compute $M_{s_j} = C_{s_j,j} \oplus h(k_j)$		
$s = 4$		

Motivation 1: CHVote Vote Casting

Receiver		Sender
knows $\mathbf{s} = (s_1, \ldots, s_k)$		knows $\mathbf{m} = (M_1, \ldots, M_n)$
for $j = 1, \ldots, k$ – pick random $r_j \in_R \mathbb{Z}_q$ – compute $a_{j,1} = \Gamma(s_j) \cdot g_1^{r_j}$ – compute $a_{j,2} = g_2^{r_j}$ – let $a_j = (a_{j,1}, a_{j,2})$	$\mathbf{a} = (a_1, \ldots, a_k)$ $\longrightarrow$	
		compute $\mathbf{b}$, $\mathbf{C}$, d
	$\mathbf{b} = (b_1, \ldots, b_k)$, $\mathbf{C} = (C_{ij})_{n \times k}, d$ $\longleftarrow$	
for $j = 1, \ldots, k$ – compute $k_j = b_j \cdot d^{-r_j}$ – compute $M_{s_j} = C_{s_j,j} \oplus h(k_j)$	$s = 4$	

For $k \leq 150$ and $s = 4$, we get up to $k \cdot (s + 2) = 900$ modexps

Motivation 2: Verify Shuffle Proof

Algorithm: CheckShuffleProof$(\tilde{\pi}, \mathbf{e}, \tilde{\mathbf{e}}, pk)$
$\mathbf{h} \leftarrow$ GetGenerators(N)
$\mathbf{u} \leftarrow$ GetNIZKPChallenges$(N, (\mathbf{e}, \tilde{\mathbf{e}}, \mathbf{c}), \tau)$
$y \leftarrow (\mathbf{e}, \tilde{\mathbf{e}}, \mathbf{c}, \hat{\mathbf{c}}, pk)$
$c \leftarrow$ GetNIZKPChallenge(y, t, τ)
$\bar{c} \leftarrow \prod_{i=1}^{N} c_i / \prod_{i=1}^{N} h_i \bmod p$
$u \leftarrow \prod_{i=1}^{N} u_i \bmod q$
$\hat{c} \leftarrow \hat{c}_N / h^u \bmod p$
$\tilde{c} \leftarrow \prod_{i=1}^{N} c_i^{u_i} \bmod p$
$t'_1 \leftarrow \bar{c}^{\,c} \cdot g^{s_1} \bmod p$
$t'_2 \leftarrow \hat{c}^{\,c} \cdot g^{s_2} \bmod p$
$t'_3 \leftarrow \tilde{c}^{\,c} \cdot g^{s_3} \prod_{i=1}^{N} h_i^{\tilde{s}_i} \bmod p$
$(a, b) \leftarrow (\prod_{i=1}^{N} a_i^{u_i} \bmod p, \prod_{i=1}^{N} b_i^{u_i} \bmod p)$
$(t'_{4,1}, t'_{4,2}) \leftarrow (a^c \cdot pk^{-s_4} \prod_{i=1}^{N} \tilde{a}_i^{\tilde{s}_i} \bmod p, b^c \cdot g^{-s_4} \prod_{i=1}^{N} \tilde{b}_i^{\tilde{s}_i} \bmod p)$
$\hat{c}_0 \leftarrow h$
for $i = 1, \ldots, N$ **do**
  $\hat{t}'_i \leftarrow \hat{c}_i^{\,c} \cdot g^{\hat{s}_i} \cdot \hat{c}_{i-1}^{\tilde{s}_i} \bmod p$
$t' \leftarrow (t'_1, t'_2, t'_3, (t'_{4,1}, t'_{4,2}), (\hat{t}'_1, \ldots, \hat{t}'_N))$
return $(t = t')$

Motivation 2: Verify Shuffle Proof

Algorithm: CheckShuffleProof$(\tilde{\pi}, \mathbf{e}, \tilde{\mathbf{e}}, pk)$
$\mathbf{h} \leftarrow$ GetGenerators(N)
$\mathbf{u} \leftarrow$ GetNIZKPChallenges$(N, (\mathbf{e}, \tilde{\mathbf{e}}, \mathbf{c}), \tau)$
$y \leftarrow (\mathbf{e}, \tilde{\mathbf{e}}, \mathbf{c}, \hat{\mathbf{c}}, pk)$
$c \leftarrow$ GetNIZKPChallenge(y, t, τ)
$\bar{c} \leftarrow \prod_{i=1}^{N} c_i / \prod_{i=1}^{N} h_i \bmod p$
$u \leftarrow \prod_{i=1}^{N} u_i \bmod q$
$\hat{c} \leftarrow \hat{c}_N / h^u \bmod p$
$\tilde{c} \leftarrow \prod_{i=1}^{N} c_i^{u_i} \bmod p$
$t'_1 \leftarrow \bar{c}^{\,c} \cdot g^{s_1} \bmod p$
$t'_2 \leftarrow \hat{c}^{\,c} \cdot g^{s_2} \bmod p$
$t'_3 \leftarrow \tilde{c}^{\,c} \cdot g^{s_3} \prod_{i=1}^{N} h_i^{\tilde{s}_i} \bmod p$
$(a, b) \leftarrow (\prod_{i=1}^{N} a_i^{u_i} \bmod p, \prod_{i=1}^{N} b_i^{u_i} \bmod p)$
$(t'_{4,1}, t'_{4,2}) \leftarrow (a^c \cdot pk^{-s_4} \prod_{i=1}^{N} \tilde{a}_i^{\tilde{s}_i} \bmod p, b^c \cdot g^{-s_4} \prod_{i=1}^{N} \tilde{b}_i^{\tilde{s}_i} \bmod p)$
$\hat{c}_0 \leftarrow h$
for $i = 1, \ldots, N$ **do**
  $\hat{t}'_i \leftarrow \hat{c}_i^{\,c} \cdot g^{\hat{s}_i} \cdot \hat{c}_{i-1}^{\tilde{s}_i} \bmod p$
$t' \leftarrow (t'_1, t'_2, t'_3, (t'_{4,1}, t'_{4,2}), (\hat{t}'_1, \ldots, \hat{t}'_N))$
return $(t = t')$

Motivation 2: Verify Shuffle Proof

Algorithm: CheckShuffleProof$(\tilde{\pi}, \mathbf{e}, \tilde{\mathbf{e}}, pk)$
$\mathbf{h} \leftarrow$ GetGenerators(N)
$\mathbf{u} \leftarrow$ GetNIZKPChallenges$(N, (\mathbf{e}, \tilde{\mathbf{e}}, \mathbf{c}), \tau)$
$y \leftarrow (\mathbf{e}, \tilde{\mathbf{e}}, \mathbf{c}, \hat{\mathbf{c}}, pk)$
$c \leftarrow$ GetNIZKPChallenge(y, t, τ)
$\bar{c} \leftarrow \prod_{i=1}^{N} c_i / \prod_{i=1}^{N} h_i \bmod p$
$u \leftarrow \prod_{i=1}^{N} u_i \bmod q$
$\hat{c} \leftarrow \hat{c}_N / h^u \bmod p$
$\tilde{c} \leftarrow \prod_{i=1}^{N} c_i^{u_i} \bmod p$
$t'_1 \leftarrow \bar{c}^{\,c} \cdot g^{s_1} \bmod p$
$t'_2 \leftarrow \hat{c}^{\,c} \cdot g^{s_2} \bmod p$
$t'_3 \leftarrow \tilde{c}^{\,c} \cdot g^{s_3} \prod_{i=1}^{N} h_i^{\tilde{s}_i} \bmod p$
$(a, b) \leftarrow (\prod_{i=1}^{N} a_i^{u_i} \bmod p, \prod_{i=1}^{N} b_i^{u_i} \bmod p)$
$(t'_{4,1}, t'_{4,2}) \leftarrow (a^c \cdot pk^{-s_4} \prod_{i=1}^{N} \tilde{a}_i^{\tilde{s}_i} \bmod p, b^c \cdot g^{-s_4} \prod_{i=1}^{N} \tilde{b}_i^{\tilde{s}_i} \bmod p)$
$\hat{c}_0 \leftarrow h$
for $i = 1, \ldots, N$ **do**
  $\hat{t}'_i \leftarrow \hat{c}_i^{\,c} \cdot g^{\hat{s}_i} \cdot \hat{c}_{i-1}^{\tilde{s}_i} \bmod p$
$t' \leftarrow (t'_1, t'_2, t'_3, (t'_{4,1}, t'_{4,2}), (\hat{t}'_1, \ldots, \hat{t}'_N))$
return $(t = t')$

For $N \approx 10^6$ and $s = 4$, we get up to $9Ns = 36 \cdot 10^6$ modexps

Research Questions

- Let $|p| = 2048$ or higher
- Can 900 modexps be computed in the web browser?
  - In less than 60 seconds
  - On all platforms (including mobile phones)
  - Reasonably up-to-date web browser

Research Questions

- Let $|p| = 2048$ or higher
- Can 900 modexps be computed in the web browser?
  - In less than 60 seconds
  - On all platforms (including mobile phones)
  - Reasonably up-to-date web browser
- Can $36 \cdot 10^6$ modexps be computed on a notebook computer?
  - In less than 2 hours
  - On a high-end notebook
  - Using fastest native code libraries

Outline

- Introduction
- Modular Exponentiation
- Product Exponentiation Algorithms
- Fixed-Based Exponentiation Algorithms
- Application to Shuffle Proof
- Conclusion

Computing Modular Exponentiations

- Prime-order subgroup $\mathbb{G}_q \subset \mathbb{Z}_p^*$ of integers modulo $p = kq + 1$
  - $|p| \leq 2048$
  - $224 \leq |q| < 2048$

Computing Modular Exponentiations

- Prime-order subgroup $\mathbb{G}_q \subset \mathbb{Z}_p^*$ of integers modulo $p = kq + 1$
  - $|p| \leq 2048$
  - $224 \leq |q| < 2048$

- Single modular exponentiation (modexp):

$$z = \mathrm{Exp}(b, e, p) = b^e \bmod p$$

  for base $b \in \mathbb{G}_q$ and exponent $e \in \mathbb{Z}_q$

Computing Modular Exponentiations

- Prime-order subgroup $\mathbb{G}_q \subset \mathbb{Z}_p^*$ of integers modulo $p = kq + 1$
  - $|p| \leq 2048$
  - $224 \leq |q| < 2048$

- Single modular exponentiation (modexp):

$$z = \mathrm{Exp}(b, e, p) = b^e \bmod p$$

  for base $b \in \mathbb{G}_q$ and exponent $e \in \mathbb{Z}_q$

- Multiple modular exponentiations:

$$\mathbf{z} = \mathrm{MultExp}(\mathbf{b}, \mathbf{e}, p) = (\mathrm{Exp}(b_1, e_1, p), \ldots, \mathrm{Exp}(b_N, e_N, p))$$

  for $\mathbf{b} = (b_1, \ldots, b_N)$, $\mathbf{e} = (e_1, \ldots, e_N)$, $\mathbf{z} = (z_1, \ldots, z_N)$

Possible Speedups

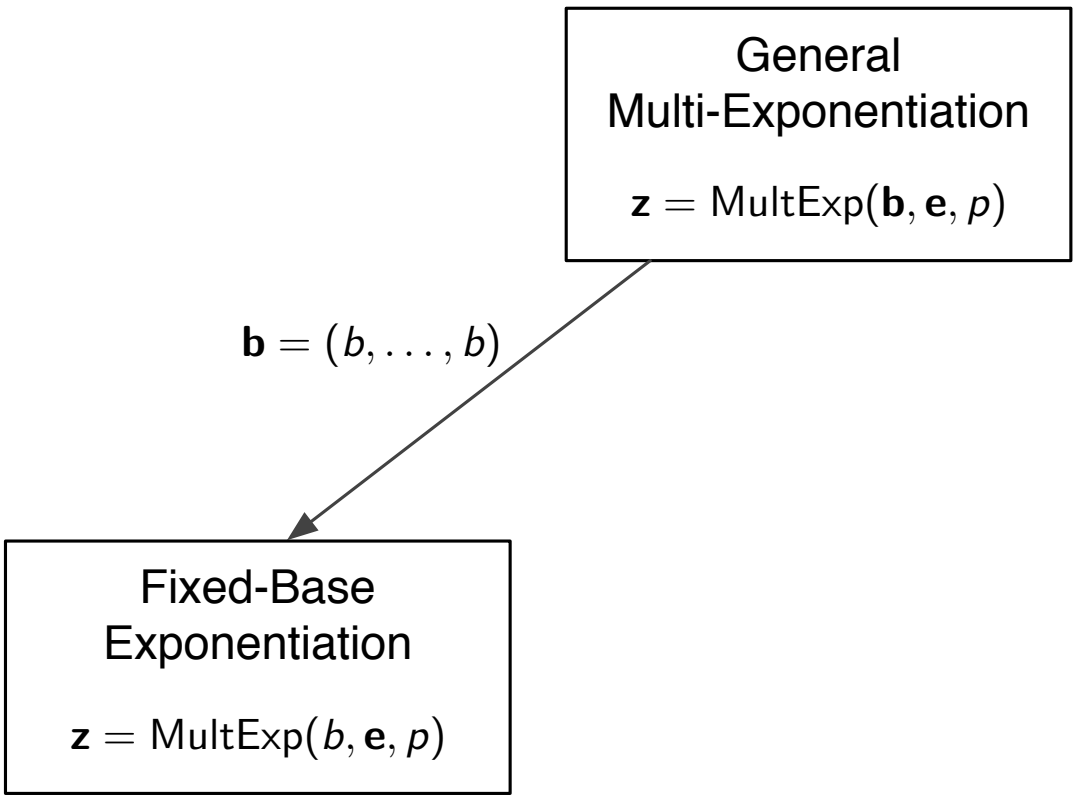
- Secure outsourcing protocols (see paper at Voting'18)
- Precomputations (e.g. upper part of OT protocol)
- Parallelism
  - Client: multi-core CPU (JS web worker)
  - Server: multi-core CPU, multiple CPUs
- Compute $(r, g^r) \in_R (\mathbb{Z}_q \times \mathbb{G}_q)$ instead of $r \in_R \mathbb{Z}_q$ and $z = g^r$
- Short exponents
  - Small subgroup (e.g. $|q| = 224$ for $|p| = 2048$)
  - Replace DL by DLSE (e.g. $|e| = 224$ for $|q| = 2047$)
- Elliptic curves
- Special-purpose algorithms: fixed-base, fixed-exponent, product exponentiation

Special Cases

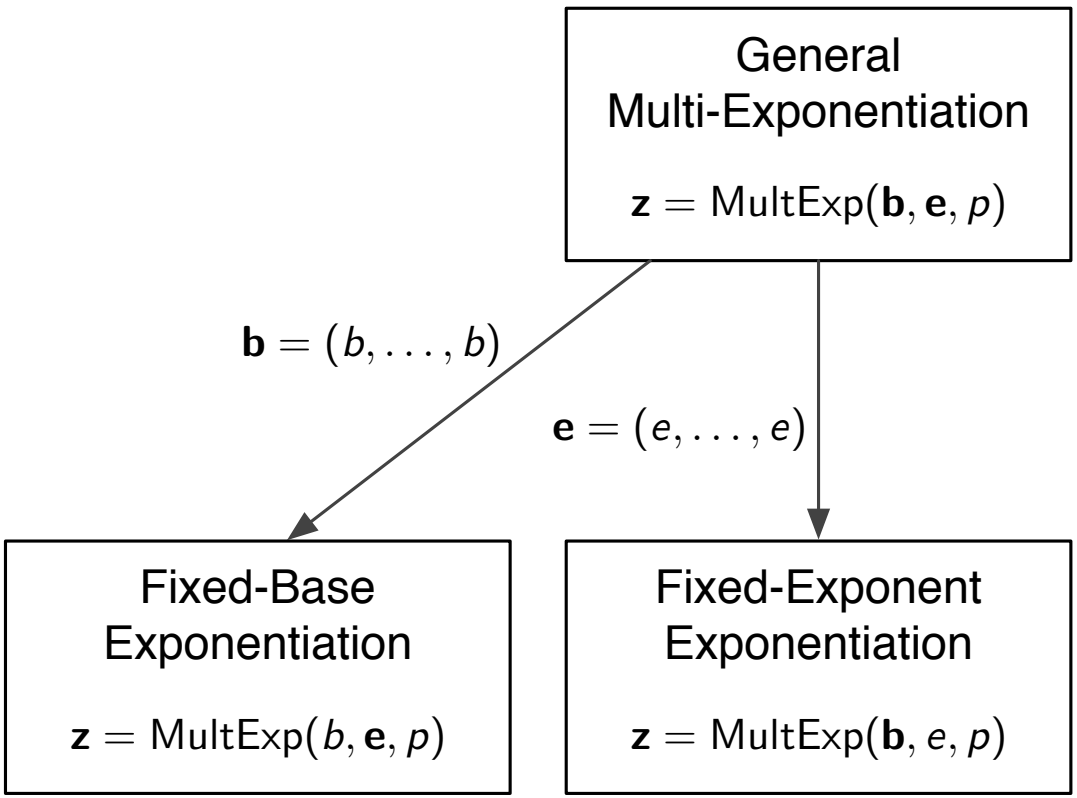
General
Multi-Exponentiation

$\mathbf{z} = \mathsf{MultExp}(\mathbf{b}, \mathbf{e}, p)$

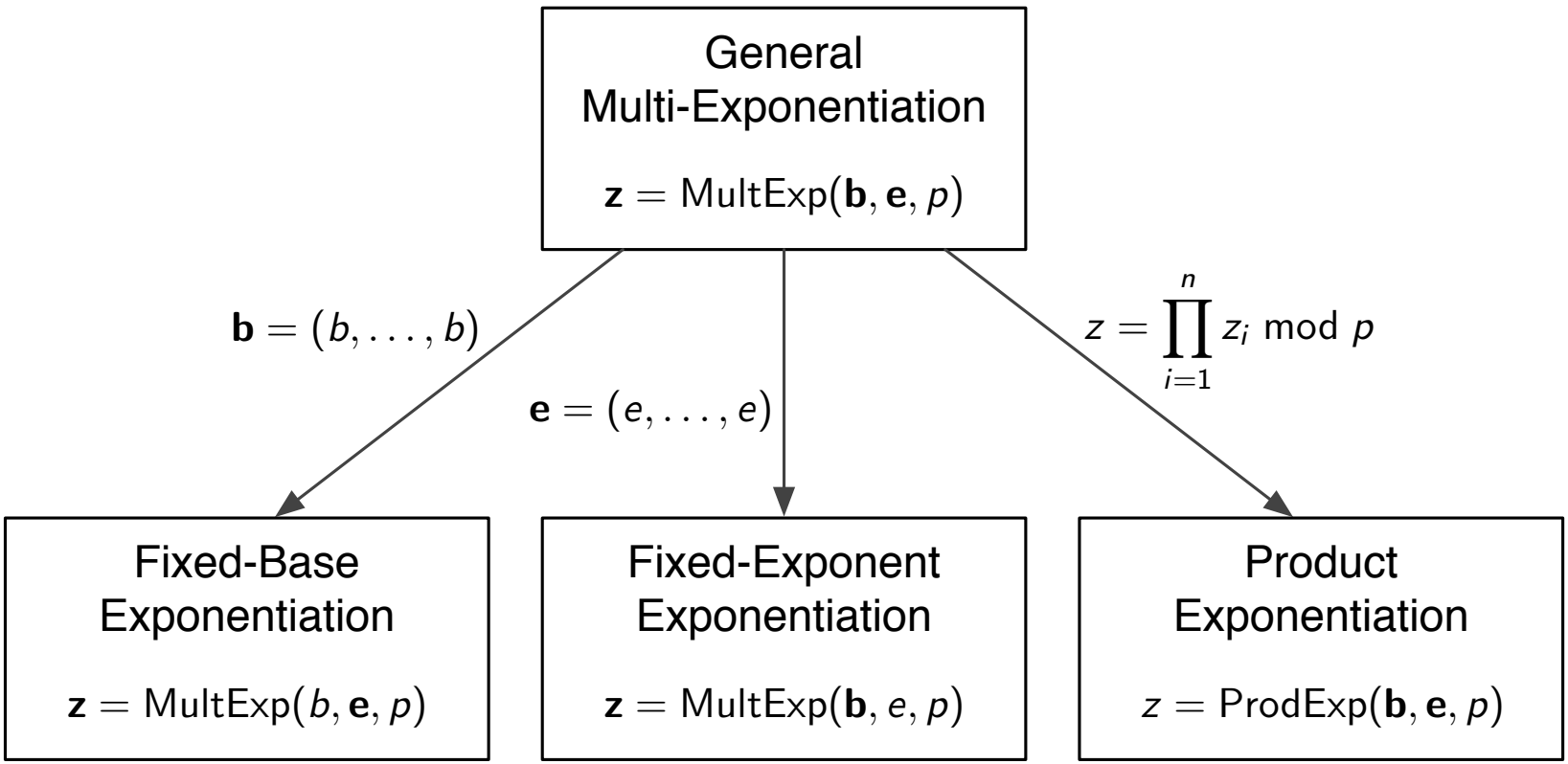
Special Cases



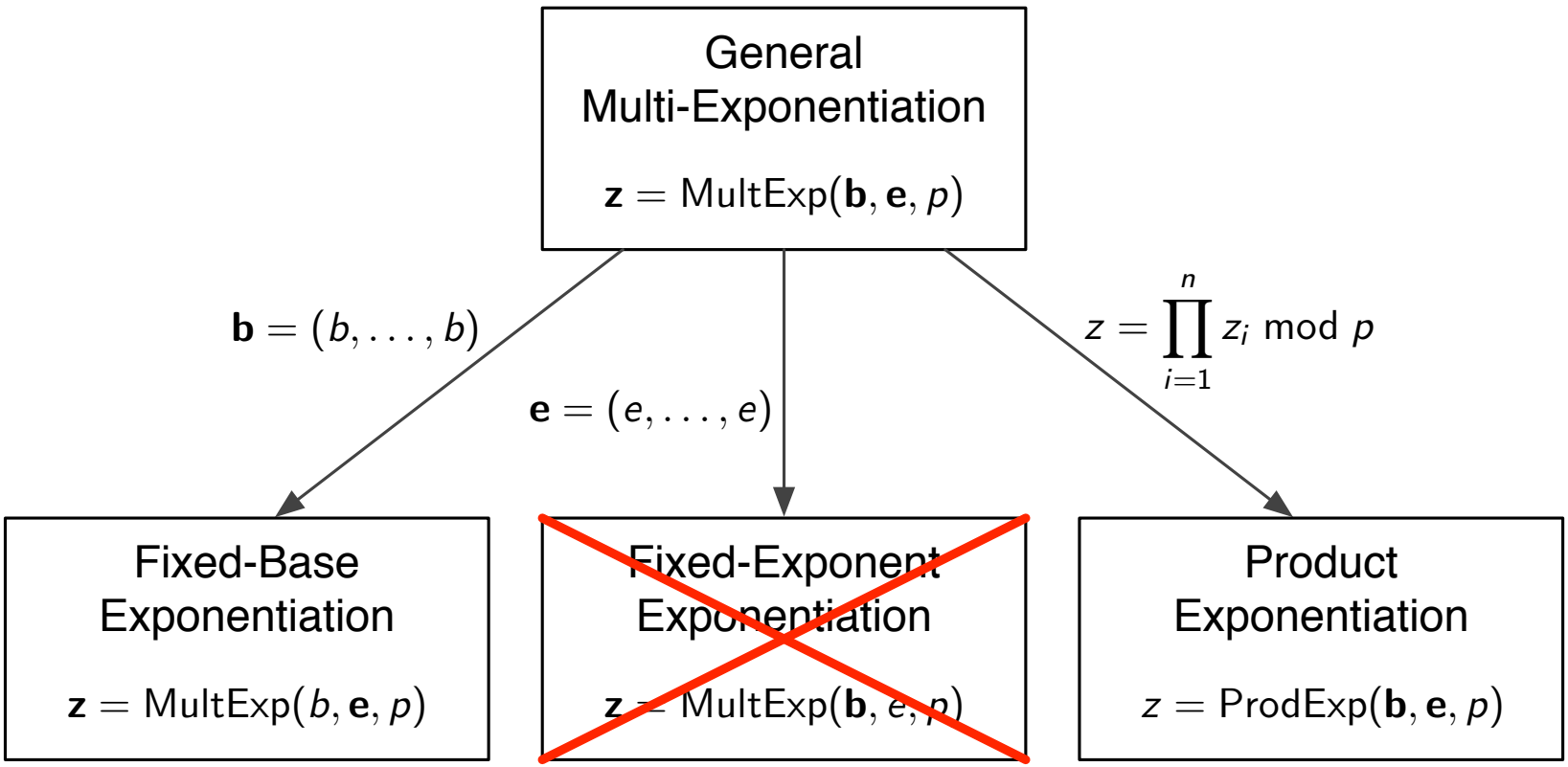
Special Cases



Special Cases



Special Cases



Assumptions and Notation

- Assumptions
  - Modular squaring and multiplications are equally efficient
  - Cost of modular multiplications is independent of $|b|$
  - Exponents are all of the same size $\ell = |e_i| \leq |q|$

Assumptions and Notation

- Assumptions
  - Modular squaring and multiplications are equally efficient
  - Cost of modular multiplications is independent of $|b|$
  - Exponents are all of the same size $\ell = |e_i| \leq |q|$
- Running time of multi-exponentiation algorithm Alg

$$M_{\mathsf{Alg}}(\ell, N) = \ \#\text{modular multiplications}$$

Assumptions and Notation

- Assumptions
  - Modular squaring and multiplications are equally efficient
  - Cost of modular multiplications is independent of $|b|$
  - Exponents are all of the same size $\ell = |e_i| \leq |q|$
- Running time of multi-exponentiation algorithm Alg

$$M_{\mathsf{Alg}}(\ell, N) = \ \#\text{modular multiplications}$$

- Average running time of algorithm Alg

$$\widetilde{M}_{\mathsf{Alg}}(\ell, N) = \frac{M_{\mathsf{Alg}}(\ell, N)}{N}$$

Assumptions and Notation

- Assumptions
  - Modular squaring and multiplications are equally efficient
  - Cost of modular multiplications is independent of $|b|$
  - Exponents are all of the same size $\ell = |e_i| \leq |q|$
- Running time of multi-exponentiation algorithm Alg

$$M_{\mathsf{Alg}}(\ell, N) = \ \#\text{modular multiplications}$$

- Average running time of algorithm Alg

$$\widetilde{M}_{\mathsf{Alg}}(\ell, N) = \frac{M_{\mathsf{Alg}}(\ell, N)}{N}$$

- Impact factor of Alg over general-purpose algorithm Alg^*

$$\mu_{\mathsf{Alg}}(\ell, N) = \frac{\widetilde{M}_{\mathsf{Alg}^*}(\ell, N)}{\widetilde{M}_{\mathsf{Alg}}(\ell, N)} = \frac{M_{\mathsf{Alg}^*}(\ell)}{\widetilde{M}_{\mathsf{Alg}}(\ell, N)}$$

General-Purpose Algorithms

- Reference algorithms in Handbook of Applied Cryptography
  - HAC 14.79: Square-and-multiply
  - HAC 14.82: Basic windowing method
  - HAC 14.83: Improved windowing method
  - HAC 14.85: Sliding-window method
- In HAC 14.82, 14.83, and 14.85, the exponent is decomposed into blocks of size k bits

$$M^k_{\text{HAC14.83}}(\ell) \;<\; 2^{k-1} + \ell \cdot \frac{k+1}{k}$$

- Optimal window size k^{opt} depends on ℓ

ℓ	82–184	217–545	566–1434	1465–3759	3802–9368	>9425
k^{opt}	4	5	6	7	8	9

- Example: $M^7_{\text{HAC14.83}}(2047) = 2401$

Outline

- Introduction
- Modular Exponentiation
- Product Exponentiation Algorithms
- Fixed-Based Exponentiation Algorithms
- Application to Shuffle Proof
- Conclusion

Product Exponentiation

- HAC 14.88 solves ProductExp$(\mathbf{b}, \mathbf{e}, p) = \prod_{i=1}^{N} b_i^{e_i}$ in time exponential in N

$$\widetilde{M}_{\text{HAC14.88}}(\ell, N) \ < \ \frac{2^N + 2\ell}{N}$$

- Best performance $\widetilde{M}_{\text{HAC14.88}}(2047, 9) = 510$ for $N = 9$

- For large N, much better performance results from splitting $\prod_{i=1}^{N} b_i^{e_i}$ into sub-products of size m and solve each sub-product using HAC 14.88

Algorithm: $\mathsf{ProductExp}_m(\boldsymbol{b}, \boldsymbol{e}, p)$

Input: Bases $\boldsymbol{b} = \boldsymbol{b}_0 || \cdots || \boldsymbol{b}_s$
Exponents $\boldsymbol{e} = \boldsymbol{e}_0 || \cdots || \boldsymbol{e}_s$
Modulus p
Sub-task size $1 \leq m \leq N$

$z \leftarrow 1$
for $j = 0, \ldots, s$ **do**
  $z_j \leftarrow \mathsf{HAC\ 14.88}(\boldsymbol{b}_j, \boldsymbol{e}_j, p)$
  $z \leftarrow z \cdot z_j \bmod p$
return z

Algorithm 1: Simple product exponentiation algorithm based on HAC 14.88.

Algorithm: $\mathsf{ProductExp}_m(\boldsymbol{b}, \boldsymbol{e}, p)$

Input: Bases $\boldsymbol{b} = \boldsymbol{b}_0 || \cdots || \boldsymbol{b}_{s-1}$
Exponents $\boldsymbol{e} = \boldsymbol{e}_0 || \cdots || \boldsymbol{e}_{s-1}$
Modulus p
Sub-task size $1 \leq m \leq N$

for $i = 0, \ldots, 2^m - 1$ **do**
  $(i_{m-1}, \ldots, i_0)_2 \leftarrow i$
  for $j = 0, \ldots, s-1$ **do**
    $(b_0, \ldots, b_{m-1}) \leftarrow \boldsymbol{b}_j$
    $B_{ij} \leftarrow \prod_{l=0}^{m-1} b_l^{i_l} \bmod p$
$z \leftarrow 1$
for $l = 0, \ldots, \ell - 1$ **do**
  $z \leftarrow z^2 \bmod p$
  for $j = 0, \ldots, s-1$ **do**
    $i \leftarrow \boldsymbol{E}_j[l]$
    $z \leftarrow z \cdot B_{ij} \bmod p$
return z

Algorithm 2: Improved product exponentiation algorithm based on HAC 14.88 and Alg.1.

Algorithm: ProductExp$_m(\boldsymbol{b}, \boldsymbol{e}, p)$

Input: Bases $\boldsymbol{b} = \boldsymbol{b}_0 || \cdots || \boldsymbol{b}_s$
Exponents $\boldsymbol{e} = \boldsymbol{e}_0 || \cdots || \boldsymbol{e}_s$
Modulus p
Sub-task size $1 \leq m \leq N$

$z \leftarrow 1$
for $j = 0, \ldots, s$ **do**
  $z_j \leftarrow$ HAC 14.88$(\boldsymbol{b}_j, \boldsymbol{e}_j, p)$
  $z \leftarrow z \cdot z_j \bmod p$
return z

Algorithm 1: Simple product exponentiation algorithm based on HAC 14.88.

Algorithm 2 is approximately 45% faster than Algorithm 1

Algorithm: ProductExp$_m(\boldsymbol{b}, \boldsymbol{e}, p)$

Input: Bases $\boldsymbol{b} = \boldsymbol{b}_0 || \cdots || \boldsymbol{b}_{s-1}$
Exponents $\boldsymbol{e} = \boldsymbol{e}_0 || \cdots || \boldsymbol{e}_{s-1}$
Modulus p
Sub-task size $1 \leq m \leq N$

for $i = 0, \ldots, 2^m - 1$ **do**
  $(i_{m-1}, \ldots, i_0)_2 \leftarrow i$
  for $j = 0, \ldots, s-1$ **do**
    $(b_0, \ldots, b_{m-1}) \leftarrow \boldsymbol{b}_j$
    $B_{ij} \leftarrow \prod_{l=0}^{m-1} b_l^{i_l} \bmod p$
$z \leftarrow 1$
for $l = 0, \ldots, \ell - 1$ **do**
  $z \leftarrow z^2 \bmod p$
  for $j = 0, \ldots, s-1$ **do**
    $i \leftarrow \boldsymbol{E}_j[l]$
    $z \leftarrow z \cdot B_{ij} \bmod p$
return z

Algorithm 2: Improved product exponentiation algorithm based on HAC 14.88 and Alg.1.

Discussion

- The average running time of Algorithm 2 is

$$\widetilde{M}^{m}_{\text{Alg.2}}(\ell, N) < \frac{2^m + \ell}{m} + \frac{\ell}{N}$$

- For large N, the optimal block size m^{opt} depends only on ℓ

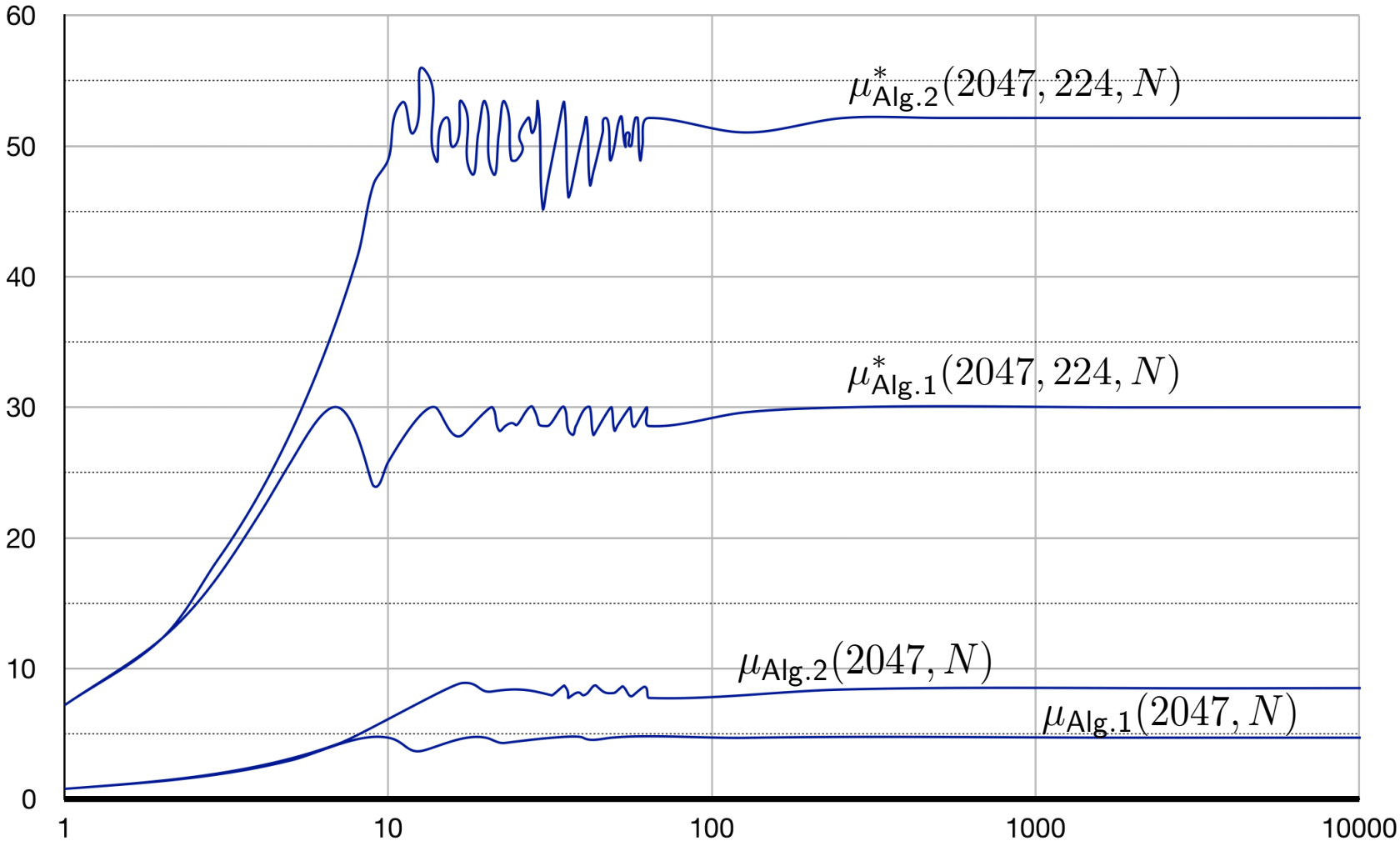
ℓ	80–147	174–349	380–802	845–1839	1896–4148	4231–9284	>9285
m^{opt}	5	6	7	8	9	10	11

- For $\ell = 2047$ and large N, we get

$$\widetilde{M}^{9}_{\text{Alg.2}}(2047, N) = 282 \quad \text{and} \quad \mu_{\text{Alg.2}}(2047, N) = 8.51$$

- We are not aware of any public description of Algorithm 2, but it is implemented in Wikström's GNP-MEE library

Performance Comparison



Outline

- Introduction
- Modular Exponentiation
- Product Exponentiation Algorithms
- Fixed-Based Exponentiation Algorithms
- Application to Shuffle Proof
- Conclusion

Fixed-Based Exponentiation

- For $e = (E_{t-1} \cdots E_1 E_0)_B$ written in base $B = 2^k$, we get

$$\begin{aligned}\mathrm{Exp}(b, e, p) = b^{\sum_{i=0}^{t-1} E_i B^i} &= \prod_{i=0}^{t-1} b^{E_i B^i} = \prod_{i=0}^{t-1} (b^{B^i})^{E_i} \\ &= \mathrm{ProductExp}((b_0, \ldots, b_{t-1}), (E_0, \ldots, E_{t-1})), p),\end{aligned}$$

  for $b_i = b^{B^i}$

- If b is a fixed base in $\mathrm{MultExp}(b, \mathbf{e}, p)$, then $(b_0, \ldots, b_{t-1})$ remains the same for all N modexps

- Algorithm sketch:
  - For $0 \leq i \leq t-1$, pre-compute $b_i = b^{B^i}$
  - Save $(b_0, \ldots, b_{t-1})$ for later use
  - For $1 \leq i \leq N$, compute $z_i = b^{e_i}$ using HAC 14.88 or Alg. 2

Algorithm: $\mathsf{FixedBaseExp}_{k,m}(b, \boldsymbol{e}, p)$

Input: Base b
Exponents $\boldsymbol{e} = (e_1, \ldots, e_N)$, $e_i = (e_{i,t-1} \cdots, e_{i,1} e_{i,0})_B$
Modulus p
Block size $1 \le k \le \ell$, $B = 2^k$
Sub-task size $1 \le m \le t$

for $i = 0, \ldots, t-1$ **do**
  $b_i \leftarrow b$
  if $i < t-1$ **then**
    for $j = 1, \ldots, k$ **do**
      $b \leftarrow b^2 \bmod p$

$\boldsymbol{b} = (b_0, \ldots, b_{t-1})$
for $i = 1, \ldots, N$ **do**
  $z_i \leftarrow \mathsf{ProductExp}_m(\boldsymbol{b}, (e_{i,0}, \ldots, e_{i,t-1}), p)$ // using HAC 14.88, Alg.1, or Alg.2
$\boldsymbol{z} \leftarrow (z_1, \ldots, z_N)$
return $\boldsymbol{z}$

Algorithm 3: Fixed-base exponentiation algorithm based on HAC 14.88, Alg.1, or Alg.2. In case of HAC 14.88, the parameter m is irrelevant.

Algorithm: $\mathsf{FixedBaseExp}_{k,m}(b, \boldsymbol{e}, p)$

Input: Base b
Exponents $\boldsymbol{e} = (e_1, \ldots, e_N)$, $e_i = (e_{i,t-1} \cdots, e_{i,1} e_{i,0})_B$
Modulus p
Block size $1 \le k \le \ell$, $B = 2^k$
Sub-task size $1 \le m \le t$

for $i = 0, \ldots, t-1$ **do**
  $b_i \leftarrow b$
  if $i < t-1$ **then**
    for $j = 1, \ldots, k$ **do**
      $b \leftarrow b^2 \bmod p$

PRECOMPUTATION

$\boldsymbol{b} = (b_0, \ldots, b_{t-1})$
for $i = 1, \ldots, N$ **do**
  $z_i \leftarrow \mathsf{ProductExp}_m(\boldsymbol{b}, (e_{i,0}, \ldots, e_{i,t-1}), p)$ // using HAC 14.88, Alg.1, or Alg.2
$\boldsymbol{z} \leftarrow (z_1, \ldots, z_N)$
return $\boldsymbol{z}$

Algorithm 3: Fixed-base exponentiation algorithm based on HAC 14.88, Alg.1, or Alg.2. In case of HAC 14.88, the parameter m is irrelevant.

Discussion

- The average running time of Algorithm 3 in combination with Algorithm 2 is

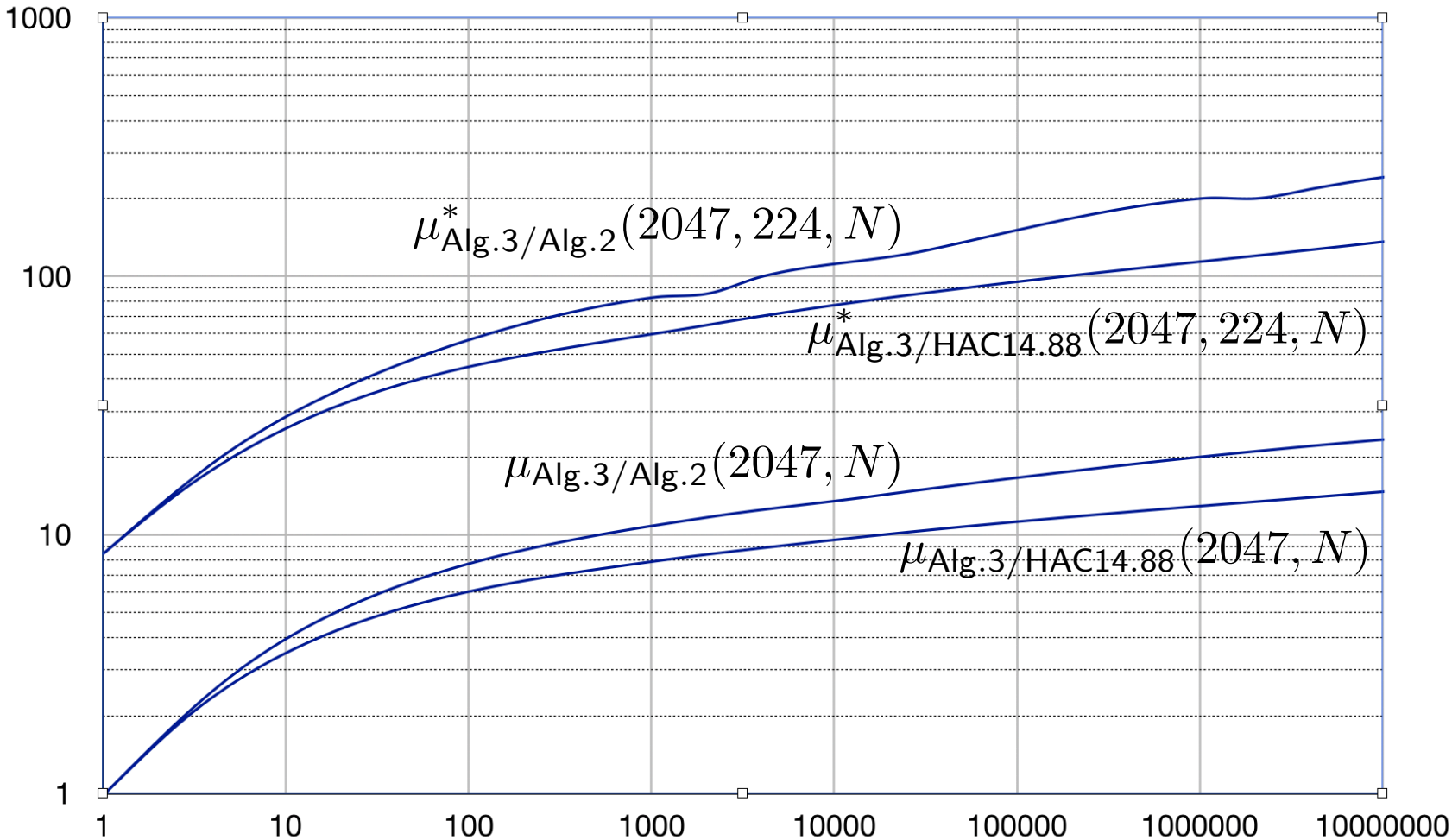
$$\widetilde{M}^{k,m}_{\text{Alg.3/Alg.2}}(\ell, N) < \frac{s \cdot 2^m + \ell}{N} + \frac{\ell}{m} + k$$

- Optimal values k^{opt} and m^{opt} depends on both ℓ and N, e.g.:

$$\widetilde{M}^{19,12}_{\text{Alg.3/Alg.2}}(2047, 1000) = 225$$
$$\widetilde{M}^{6,17}_{\text{Alg.3/Alg.2}}(2047, 1000000) = 120$$

- Remark: These running times correspond exactly to the comb algorithm by Lim & Lee (HAC 14.117)

Performance Comparison



Outline

- Introduction
- Modular Exponentiation
- Product Exponentiation Algorithms
- Fixed-Based Exponentiation Algorithms
- Application to Shuffle Proof
- Conclusion

Overview of Modular Exponentiations

		Shuffle	**Generate Proof**	**Verify Proof**	
		ℓ_{long}	ℓ_{long}	ℓ_{long}	ℓ_{short}
Exp		–	–	$n+7$	n
ProductExp		–	$3n$	$3n$	$3n$
FixedBaseExp	g	n	$4n+4$	$n+4$	–
	h	–	n	–	–
	pk	n	1	–	–
Total		$2n$	$8n+5$	$5n+11$	$4n$

Table 3: Number of exponentiations for shuffling n votes in a verifiable mix-net.

Overview of Modular Exponentiations

		Shuffle	**Generate Proof**	**Verify Proof**	
		ℓ_{long}	ℓ_{long}	ℓ_{long}	ℓ_{short}
Exp		–	–	$n+7$	n
ProductExp		–	$3n$	$3n$	$3n$
FixedBaseExp	g	n	$4n+4$	$n+4$	–
	h	–	n	–	–
	pk	n	1	–	–
Total		$2n$	$8n+5$	$5n+11$	$4n$

Table 3: Number of exponentiations for shuffling n votes in a verifiable mix-net.

Performance

	$n = 10$		$n = 100$		$n = 1000$		$n = 10^4$		$n = 10^5$		$n = 10^6$	
Shuffle	1216	3.95	626	7.66	450	10.66	352	13.63	286	16.78	240	19.99
Generate proof	2563	7.08	1884	9.06	1623	10.44	1463	11.58	1359	12.46	1283	13.20
Verify proof	5132	2.96	3486	3.68	3306	3.81	3276	3.84	3265	3.85	3262	3.85

Table 4: Relative running times (1st column) and impact factors (2nd column) of different shuffle algorithms in a setting with $\ell_{\mathsf{long}} = 2047$ and $\ell_{\mathsf{short}} = 112$.

Performance

	$n = 10$		$n = 100$		$n = 1000$		$n = 10^4$		$n = 10^5$		$n = 10^6$	
Shuffle	1216	3.95	626	7.66	450	10.66	352	13.63	286	16.78	240	19.99
Generate proof	2563	7.08	1884	9.06	1623	10.44	1463	11.58	1359	12.46	1283	13.20
Verify proof	5132	2.96	3486	3.68	3306	3.81	3276	3.84	3265	3.85	3262	3.85

Table 4: Relative running times (1st column) and impact factors (2nd column) of different shuffle algorithms in a setting with $\ell_{\mathsf{long}} = 2047$ and $\ell_{\mathsf{short}} = 112$.

Outline

- Introduction
- Modular Exponentiation
- Product Exponentiation Algorithms
- Fixed-Based Exponentiation Algorithms
- Application to Shuffle Proof
- Conclusion

Existing Libraries

Library	**JSBN**	**Leemon**	**VJSC**	**MiniGMP**	**GMP/MEE**
Language	JavaScript	JavaScript	JavaScript	C	C
Author(s)	T. Wu	L. C. Baird	D. Wikström	N. Möller	T. Granlund D. Wikström
Exp	HAC 14.85*	HAC 14.79*	HAC 14.83	HAC 14.79	HAC 14.85*
ProductExp	*unsupported*	*unsupported*	HAC 14.88	*unsupported*	Alg.2
FixedBaseExp	*unsupported*	*unsupported*	Alg.3 / Alg.1	*unsupported*	Alg.3 / Alg.1

Table 5: JavaScript and C libraries for large integer arithmetic. Algorithm marked with a star (*) use Montgomery reduction.

Existing Libraries

Library	**JSBN**	**Leemon**	**VJSC**	**MiniGMP**	**GMP/MEE**
Language	JavaScript	JavaScript	JavaScript	C	C
Author(s)	T. Wu	L. C. Baird	D. Wikström	N. Möller	T. Granlund D. Wikström
Exp	HAC 14.85*	HAC 14.79*	HAC 14.83	HAC 14.79	HAC 14.85*
ProductExp	*unsupported*	*unsupported*	HAC 14.88	*unsupported*	Alg.2
FixedBaseExp	*unsupported*	*unsupported*	Alg.3 / Alg.1	*unsupported*	Alg.3 / Alg.1

Table 5: JavaScript and C libraries for large integer arithmetic. Algorithm marked with a star (*) use Montgomery reduction.

Performance of Existing Libraries

	Server		**Client**			
ℓ	GMP	MiniGMP	VJSC	JSBN	Leeman	MiniGMP/WASM
2048	3.05ms	19.23ms	81.55ms	105.68ms	181.89ms	133.59ms
3072	8.97ms	63.14ms	248.69ms	332.81ms	589.74ms	447.27ms

Table 6: Average running times for modular exponentiations in different libraries.

Performance of Existing Libraries

	Server		Client			
ℓ	GMP	MiniGMP	VJSC	JSBN	Leeman	MiniGMP/WASM
2048	3.05ms	19.23ms	81.55ms	105.68ms	181.89ms	133.59ms
3072	8.97ms	63.14ms	248.69ms	332.81ms	589.74ms	447.27ms

Table 6: Average running times for modular exponentiations in different libraries.

Conclusion

- Product and fixed-base exponentiation algorithms improve the performance by one order of magnitude
  - Algorithm 2 for ProductExp($\mathbf{b}, \mathbf{e}, p$)
  - Algorithm 3 (with Algorithm 2) for MultExp($b, \mathbf{e}, p$)

Conclusion

- Product and fixed-base exponentiation algorithms improve the performance by one order of magnitude
  - Algorithm 2 for ProductExp($\mathbf{b}, \mathbf{e}, p$)
  - Algorithm 3 (with Algorithm 2) for MultExp($b, \mathbf{e}, p$)

- Short-exponents improve the performance by an additional order of magnitude

Conclusion

- Product and fixed-base exponentiation algorithms improve the performance by one order of magnitude
  - Algorithm 2 for ProductExp($\mathbf{b}, \mathbf{e}, p$)
  - Algorithm 3 (with Algorithm 2) for MultExp($b, \mathbf{e}, p$)
- Short-exponents improve the performance by an additional order of magnitude
- Parallelization using multi-core CPUs adds another order of magnitude

Conclusion

- Product and fixed-base exponentiation algorithms improve the performance by one order of magnitude
  - Algorithm 2 for ProductExp($\mathbf{b}, \mathbf{e}, p$)
  - Algorithm 3 (with Algorithm 2) for MultExp($b, \mathbf{e}, p$)
- Short-exponents improve the performance by an additional order of magnitude
- Parallelization using multi-core CPUs adds another order of magnitude
- Best libraries available:
  - Server: GMP with GMP-MEE
  - Client: VJSC

Research Questions

- Can 900 modexps be computed in the web browser?
  - In less than 60 seconds
  - On all platforms (including mobile phones)
  - Reasonably up-to-date web browser

Research Questions

- Can 900 modexps be computed in the web browser?
  - In less than 60 seconds
  - On all platforms (including mobile phones)
  - Reasonably up-to-date web browser

YES

Research Questions

- Can 900 modexps be computed in the web browser?
  - In less than 60 seconds
  - On all platforms (including mobile phones)
  - Reasonably up-to-date web browser

YES

- Can $36 \cdot 10^6$ modexps be computed on a notebook computer?
  - In less than 2 hours
  - On a high-end notebook
  - Using fastest native code libraries

Research Questions

- Can 900 modexps be computed in the web browser?
  - In less than 60 seconds
  - On all platforms (including mobile phones)
  - Reasonably up-to-date web browser

YES

- Can $36 \cdot 10^6$ modexps be computed on a notebook computer?
  - In less than 2 hours
  - On a high-end notebook
  - Using fastest native code libraries

YES