Fixed Argument Pairings

Craig Costello

craig.costello@qut.edu.au
Queensland University of Technology

LatinCrypt 2010
Puebla, Mexico

Joint work with Douglas Stebila

Pairings

A mapping $e : \mathbb{G}_1 \times \mathbb{G}_2 \to \mathbb{G}_T$:

- $P \in \mathbb{G}_1$, $Q \in \mathbb{G}_2$ and $e(P, Q) \in \mathbb{G}_T$: groups are all of prime order r (usually)
- **Bilinear** : $\mathbf{e(aP, bQ) = e(P, Q)^{ab} = e(bP, aQ)}$
- Now used all over the place: all types of encryption, all types of signatures, all types of key-agreement schemes, all types of proof systems, etc ...

Motivation: speed of pairing computation

- The efficient implementation of pairings has become quite a broad field of research in its own right
- Remarkable progress in the field: computing $e(P, Q)$
  - 1993: a few minutes
  - Today: less than a millisecond (next talk)
- Somewhat strangely, they still have this “slow” stigma attached to them
- Until this myth is dispelled (and probably long after), we will continue to look for optimizations wherever we can find them...

This work...

Fixed Argument Pairings

Computing $e(P, Q)$ where one of the arguments, either P or Q, is fixed

- It could be that $P = P_{\texttt{priv}}$ is a long-term secret key that is used to decrypt many messages (paired with many different Q_i)
- It could be that $Q = Q_{\texttt{pub}}$ is a public parameter that is used in every encryption (paired with many P_i's)
- It could be that $Q = Q_{\texttt{ID}}$ is an identity-based parameter belonging to an identity with whom communication is regular
- It could be a whole range of things...

Fixed arguments in pairing-based cryptosystems

	# pairings	fixed arguments	# pairings	fixed arguments
Public key encryption	**Encryption**		**Decryption**	
Boyen-Mei-Waters	0		1	2^{nd}
ID-based encryption	**Encryption**		**Decryption**	
Boneh-Franklin	1	2^{nd}	1	1^{st}
Boneh-Boyen	0		1	2^{nd}
Waters	0		2	both in 2^{nd}
Attribute-based encr.	**Encryption**		**Decryption**	
GPSW	0		$\leq$ #attr.	all in 1^{st}
LOSTW	0		$\leq 2 \cdot$ #attr.	all in 2^{nd}
ID-based signatures	**Signing**		**Verification**	
Waters	0		2	1 in 2^{nd}
ID-based key exchange	**Initiator**		**Responder**	
Smart-1	2	1 in 1^{st}, 1 in 2^{nd}	2	1 in 1^{st} , 1 in 2^{nd}
Chen-Kudla	1	1^{st}	1	2^{nd}
McCullagh-Barreto	1	2^{nd}	1	2^{nd}

Boneh Franklin IBE

1. **Setup:** Public parameters are the pairing groups, some hash functions and $\langle P, P_0 \rangle$, where $P = sP_0$ and s is TA's master secret
2. **Extract:** Given an identity $\mathtt{ID} \in \{0,1\}^*$, set $d_{\mathtt{ID}} = sH_1(\mathtt{ID})$ as the private key of the identity.
3. **Encrypt:** Inputs are the message M and a target identity ID.
    1. Choose random $t \in \mathbb{Z}_r$
    2. Compute the ciphertext

$$C = \langle tP, M \oplus H_2(e(H_1(\mathtt{ID}), P_0)^t) \rangle$$

.

4. **Decrypt:** Given a ciphertext $\langle U, V \rangle$ and a private key $d_{\mathtt{ID}}$, compute:

$$M = V \oplus H_2(e(d_{\mathtt{ID}}, U))$$

.

“Our” work: other options for the title

- The possibility of exploiting fixed argument pairings was first discussed by Scott and again by Scott *et al.* in 2006
- They suggested natural pre-computations in the fixed argument, before the second argument exists or is known
- We simply build on their ideas and propose more pre-computations
- Other possible (more honest) titles didn’t sound as good:
  - ~~“Doing a few more pre-computations when one of the arguments is fixed in a pairing-based protocol”~~
  - ~~“Another other look at fixed argument pairings”~~
  - ~~“Fixed argument pairings revisited again”~~

"Our" work cont.: other options for the author list

- We are also indebted to the anonymous reviewer of a (similar) prior paper who suggested:

  "The authors might also like to look at the (surprisingly common) scenario, where in the pairing e(P,Q), P is a fixed system parameter, or a fixed secret key. In this case precomputation of the multiples of P greatly speeds the ate pairing in particular".

- Other possible author lists also didn't sound as impressive:
  - ~~"Craig Costello, Douglas Stebila, and Anonymous Previous Reviewer"~~
  - ~~"Craig Costello, Douglas Stebila, and Author Unknown"~~

Here comes the math...

... hang tight...

The elements of $\mathbb{G}_2$ are much bigger than the elements of $\mathbb{G}_1$ (e.g. $k = 12$)

$$\mathbb{F}_{q^{12}} = \mathbb{F}_{q^4}(\alpha) = \mathbb{F}_{q^2}(\gamma) = \mathbb{F}_q(\beta)$$

$P \in \mathbb{G}_1$: [341746248540, 710032105147]

$Q \in \mathbb{G}_2$:

$[((502478767360 \cdot \beta + 1034075074191) \cdot \gamma + 342970860051 \cdot \beta + 225764301423) \cdot \alpha^2 + ((205398279920 \cdot \beta + 182600014119) \cdot \gamma + 860891557473 \cdot \beta + 435210764901) \cdot \alpha + (1043922075477 \cdot \beta + 566889113793) \cdot \gamma + 150949917087 \cdot \beta + 21392569319,$

$((654337640030 \cdot \beta + 744622505639) \cdot \gamma + 1092264803801 \cdot \beta + 895826335783) \cdot \alpha^2 + ((529466169391 \cdot \beta + 550511036767) \cdot \gamma + 985244799144 \cdot \beta + 554170865706) \cdot \alpha + (194564971321 \cdot \beta + 969736450831) \cdot \gamma + (579122687888 \cdot \beta + 581111086076)]$

The twisted curve

- Original curve is $E(\mathbb{F}_q) : y^2 = x^3 + ax + b$
- Twisted curve is $E'(\mathbb{F}_{q^{k/d}})$: $y^2 = x^3 + a\omega^4 x + b\omega^6$, $\omega \in \mathbb{F}_{q^k}$
- Possible degrees of twists are $d \in \{2, 3, 4, 6\}$: the bigger the better!
- Twist $\Psi : E' \to E : (x', y') \to (x'/\omega^2, y'/\omega^3)$ induces $\mathbb{G}_2' = E'(\mathbb{F}_{q^{k/d}})[r]$ so that $\Psi : \mathbb{G}_2' \to \mathbb{G}_2$
- Instead of working with $Q \in \mathbb{G}_2$, a lot of work can be done with $Q' \in \mathbb{G}_2'$ defined over subfield $\mathbb{F}_{q^e} = \mathbb{F}_{q^{k/d}}$

$P \in \mathbb{G}_1$: $(341746248540, 710032105147)$
$Q' \in \mathbb{G}_2' = \Psi^{-1}(\mathbb{G}_2)$:

$((917087150949\beta + 25693192139) \cdot \omega^2, (878885791226\beta + 860765811110) \cdot \omega^3)$

Tate vs. ate pairings

Tate-like pairings

$$e_r : \mathbb{G}_1 \times \mathbb{G}_2 \to \mu_r, \ (P, Q) \mapsto f_{r,P}(Q)^{\frac{q^k-1}{r}}.$$

Ate-like pairing

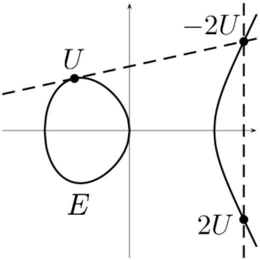
$$a_T : \mathbb{G}_2 \times \mathbb{G}_1 \to \mu_r, \ (Q, P) \mapsto f_{T,Q}(P)^{\frac{q^k-1}{r}}.$$

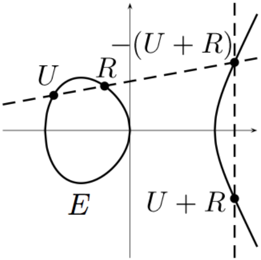
- Pairings $e(R, S)$ require the computation of Miller functions $f_{m,R}(S)$
- Function $f_{m,R}$ is of degree m
- Constructions require $\lfloor \log_2 m \rfloor$ iterations of Miller's algorithm
- Most of the work is done in the first argument
- Tate needs $\lfloor \log_2 r \rfloor$ iters, ate needs $\lfloor \log_2 T \rfloor$ iters, $T \ll r$
- Trade-off is that more work in ate is done in larger field $(\mathbb{G}_2')$

Miller's algorithm to compute $e(R, S) = f_{m,R}(S)^{(q^k-1)/r}$

$m = (m_{l-1}, \ldots, m_1, m_0)_2$ initialize: $U = R$, $f = 1$

1. for $i = l - 2$ to 0 do
   a. i. Compute $f_{\mathrm{DBL}(U)}$ in the doubling of U
      ii. $U \leftarrow [2]U$ //(DBL)
      iii. $f \leftarrow f^2 \cdot f_{\mathrm{DBL}(U)}(S)$
   b. if $m_i = 1$ then
      i. Compute $f_{\mathrm{ADD}(U,R)}$ in the addition of $U + R$
      ii. $U \leftarrow U + R$ //(ADD)
      iii. $f \leftarrow f \cdot f_{\mathrm{ADD}(U,R)}(S)$
2. $f \leftarrow f^{(q^k-1)/r}$.





An iteration of Miller's algorithm

a. i. Compute $f_{\mathrm{DBL}(U)}$ in the doubling of U
   ii. $U \leftarrow [2]U$ //(DBL)
   iii. $f \leftarrow f^2 \cdot f_{\mathrm{DBL}(U)}(S)$

b. if $m_i = 1$ then
   i. Compute $f_{\mathrm{ADD}(U,R)}$ in the addition of $U + R$
   ii. $U \leftarrow U + R$ //(ADD)
   iii. $f \leftarrow f \cdot f_{\mathrm{ADD}(U,R)}(S)$

(DBL) $[2](x_1, y_1) = (x_3, y_3)$

$$x_3 = \lambda^2 - 2x_1$$
$$y_3 = \lambda(x_1 - x_3) - y_1$$
$$\lambda = (3x_1^2 + a)/(2y_1)$$

(ADD) $(x_1, y_1) + (x_2, y_2) = (x_3, y_3)$

$$x_3 = \lambda^2 - x_1 - x_2$$
$$y_3 = \lambda(x_1 - x_3) - y_1$$
$$\lambda = (y_2 - y_1)/(x_2 - x_1)$$

$$f_{\mathrm{DBL}(U)/\mathrm{ADD}(U+R)} = y - \lambda \cdot x - (y_1 - \lambda \cdot x_1)$$
$$f_{\mathrm{DBL}(U)/\mathrm{ADD}(U+R)}(S) = y_S - \lambda \cdot x_S - (y_1 - \lambda \cdot x_1)$$

We only need to touch $S = (x_S, y_S)$ when we evaluate the line functions

$e(R, S)$: R-dependent vs. S-dependent computations

a. i. Compute $f_{\mathrm{DBL}(U)}$ in the doubling of U

ii. $U \leftarrow [2]U$ //(DBL)

iii. $f \leftarrow f^2 \cdot f_{\mathrm{DBL}(U)}(S)$

b. if $m_i = 1$ then

i. Compute $f_{\mathrm{ADD}(U,R)}$ in the addition of $U + R$

ii. $U \leftarrow U + R$ //(ADD)

iii. $f \leftarrow f \cdot f_{\mathrm{ADD}(U,R)}(S)$

$$f_{\mathrm{DBL}(U)/\mathrm{ADD}(U+R)}(S) = y_S - \lambda \cdot x_S - (y_1 - \lambda \cdot x_1)$$

- All the point operations and line coefficient computations are completely R-dependent ($U = vR$ throughout)
- If R is a fixed argument, we can pre-compute all of this before we input (or know) S
- Pre-compute and store all the $(\lambda, x_{U_i}, y_{U_i})$ tuples (Scott 2006)

The beauty of fixed arguments

An assumption: pre-computation time

We assume that we have ample time to do these pre-computations (at least a few seconds).

Another assumption: storage limit

We also assume that we have access to a significant amount of storage space to store these precomputations.

- Essentially we do whatever it takes to reduce the pairing computation time at runtime (once S is input)
- This allows us to work in (generally more expensive) affine coordinates:
  - Projective Miller lines: $F_{\mathrm{DBL}(U)} = g_x \cdot x + g_y \cdot y + g_0$
  - Affine Miller lines: $f_{\mathrm{DBL}(U)} = x + \lambda \cdot y + c$

Splitting the algorithm

- **Starting observation**: store (λ_i, c_i) tuples instead of (x_i, y_i, λ_i) tuples $\rightarrow$ only storing line functions: natural split

R-dependent pre-comps	S-dependent dynamic comps
Input: R	**Input:** `Vec`, S
for $i = l-2$ **to** 0	**for** $i = l-2$ **to** 0
Compute $f_{\mathrm{DBL}(U)} = (\lambda_i, c_i)$	
$U \leftarrow [2]U$	$f \leftarrow f^2 \cdot (y_S + \lambda_i \cdot x_S + c_i)$
Store `Vec` $\leftarrow (\lambda_i, c_i)$	
if $m_i = 1$ **then**	**if** $m_i = 1$ **then**
Compute $f_{\mathrm{ADD}(U,R)} = (\tilde{\lambda}_i, \tilde{c}_i)$	
$U \leftarrow U + R$	$f \leftarrow f \cdot (y_S + \tilde{\lambda}_i \cdot x_S + \tilde{c}_i)$
Store `Vec` $\leftarrow (\tilde{\lambda}_i, \tilde{c}_i)$	
end if	**end if**
end for	**end for**
Output: `Vec`	**Output:** $f_{m,R}S \leftarrow f$

- No major improvements, but helps to conceptualize what's to come...

Doing more pre-computations

- **Question**: can we possibly push any more of the S-dependent computations across to the R-dependent side?

R-dependent pre-comps	S-dependent dynamic comps
for $i = l-2$ **to** 0	**for** $i = l-2$ **to** 0
Compute $f_{\mathrm{DBL}(U)} = (\lambda_i, c_i)$	
$U \leftarrow [2]U$	$f \leftarrow f^2 \cdot (y_S + \lambda_i \cdot x_S + c_i)$
Store `Vec` $\leftarrow (\lambda_i, c_i)$	
if $m_i = 1$ **then**	**if** $m_i = 1$ **then**
Compute $f_{\mathrm{ADD}(U,R)} = (\tilde{\lambda}_i, \tilde{c}_i)$	
$U \leftarrow U + R$	$f \leftarrow f \cdot (y_S + \tilde{\lambda}_i \cdot x_S + \tilde{c}_i)$
Store `Vec` $\leftarrow (\tilde{\lambda}_i, \tilde{c}_i)$	
end if	**end if**
end for	**end for**

- **Answer**: perhaps we can perform operations on the line functions, before they're evaluate at S
- Once the line function is evaluated at S, it's going to be squared, so why not square the indeterminate function before evaluating it at S?

Doing more pre-computations cont...

- Analogous to prior work (CBGW at WAIFI'10) which was done for general pairings (both arguments input at the same time)
- In the case of fixed arguments, the technique is much more powerful... **any operations we do on the indeterminate functions can be done in advance**
- Loop unrolling for general pairings was only much faster in Tate-like pairings where the line function coefficients were in the ground field $\mathbb{F}_p$
- The ate pairing benefits just as much (if not more) from loop unrolling in the fixed argument scenario, as the extra operations spent in $\mathbb{F}_{p^e}$ are pre-computations anyway

The recipe

- Get the usual R-dependent output

$$\texttt{Vec} = [(\lambda_1, c_1), (\lambda_2, c_2), ..., (\lambda_L, c_L)]$$

which corresponds to L indeterminate line functions of the form $y + \lambda_i x + c_i$

- Combine n of them at a time (keeping in mind that each line function would have been squared) to form new indeterminate functions

$$\prod_{i=1}^{n} (y + \lambda_i x + c_i)^{2^{(i-1)}} = f(x) + g(x) \cdot y$$

$$= \prod_{j=0}^{T_1} z_j \cdot x^j + \prod_{j=0}^{T_2} \hat{z}_j \cdot x^j \cdot y,$$

where the z_j's and $\hat{z}_j$'s are functions of the (λ_i, c_i) tuples

- What's the best n value?
- Store these bigger functions until S exists or is input
- More pre-computational work, more storage requirements...
- **BUT less function evaluations and less Miller updates!**

The old vs. the new

S-dependent comps (OLD)	S-dependent comps (NEW)
Input: `Vec`, S	**Input:** `VecNew`, S
L **iterations**	$\lceil L/n \rceil$ **iterations**
$f \leftarrow f^2 \cdot (y_S + \lambda_i \cdot x_S + c_i)$	$f \leftarrow f^{2^n} \cdot (\prod z_j \cdot x_S^j + \prod \hat{z}_j \cdot x_S^j \cdot y_S)$
if $m_i = 1$ **then**	**if any of the old** m_i **were** 1 **then**
$f \leftarrow f \cdot (y_S + \tilde{\lambda}_i \cdot x_S + \tilde{c}_i)$	$f \leftarrow f \cdot (\prod z_j \cdot x_S^j + \prod \hat{z}_j \cdot x_S^j \cdot y_S)$
end if	**end if**
end for	**end for**
Output: $f_{m,R}S \leftarrow f$	**Output:** $f_{m,R}S \leftarrow f$

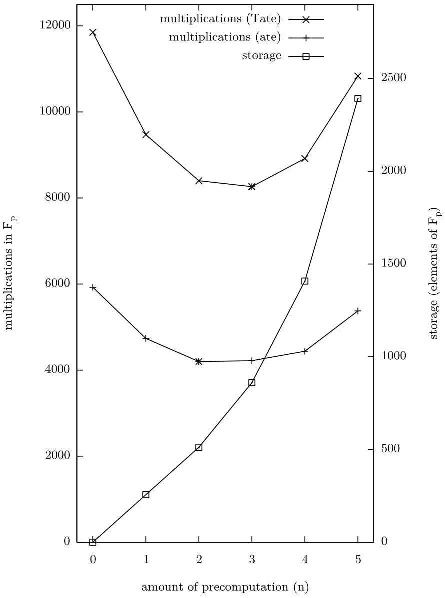
- The old way n function updates every n iterations, where as the new way has 1 function update in the equivalent of every n iterations
- It doesn't look like much, but the savings can be quite substantial...

Results

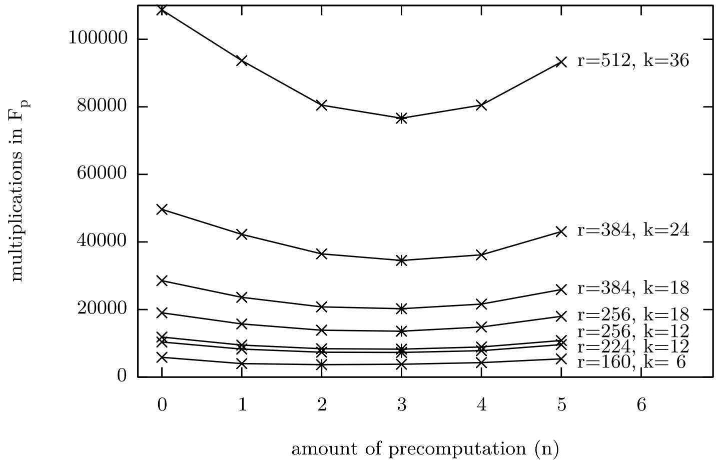
Security & r (bits)	k	Best ρ	$\mathbb{F}_p$ (bits)	$\mathbb{F}_{p^{k/d}}$ (bits)	$\mathbb{F}_{p^k}$ (bits)	Pairing	m	n	$\#\mathbf{m}_1$	% Speedup pre.	% Speedup no pre.
80	6	2.000	320	320	1920	Tate	80	2	1843	7.8	37.1
						ate	80	2	1846	7.7	37.0
$r = 160$	8	1.500	240	480	1920	Tate	120	2	5069	11.2	30.8
						ate	120	2	5058	11.4	30.9
112	12	1.000	224	448	2688	Tate	112	3	7308	11.8	29.5
						ate	56	3	3646	12.0	29.7
$r = 224$	16	1.250	280	1120	4480	Tate	112	2	13460	14.6	25.9
						ate	28	2	3346	15.1	26.3
128	12	1.000	256	512	3072	Tate	128	3	8263	12.7	30.3
						ate	64	2	4198	11.3	29.2
	16	1.250	320	1280	4096	Tate	128	2	15368	14.7	26.0
$r = 256$						ate	32	2	3823	15.1	26.3
	18	1.333	342	1026	4608	Tate	128	3	13590	13.6	28.5
						ate	43	3	4697	11.1	26.5
192	18	1.333	512	1536	6912	Tate	192	3	20173	14.2	29.3
						ate	64	3	6881	12.5	27.6
$r = 384$	24	1.250	478	1912	9216	Tate	192	3	34540	18.2	30.4
						ate	48	3	8577	18.7	30.9
256	32	1.125	576	4608	16384	Tate	256	3	87876	17.9	25.7
						ate	32	3	10777	19.5	27.1
$r = 512$	36	1.167	598	3588	18432	Tate	264	3	102960	18.2	29.5
						ate	43	3	13202	16.1	27.7

n column: represents the optimal number of iterations to merge...
i.e. the optimal number of (λ_i, c_i) "line functions" to combine

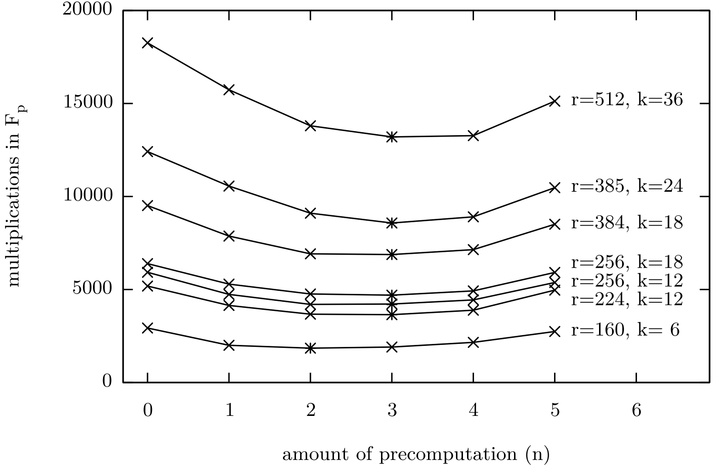
Tate vs. ate $\mathbb{F}_p$-muls vs. storage cost ($k = 12$, $r = 256$)



Tate $\mathbb{F}_p$-muls for different k, n



Ate $\mathbb{F}_p$-muls for different k, n



In case you missed any of that...

- Pairings $e(R,S)$ are just functions of the four coordinates $e(R,S) = f(x_R, y_R, x_S, y_S)$
- We just tweaked the pairing computation algorithm to do a bit more with x_R and y_R, in order to reduce the workload when the $S = (x_S, y_S)'s$ come

Summary

The lesson learned...

- **IF** you're wanting to implement one of the many exciting pairing-based protocols...
- **AND** there is a long-term fixed argument that could be exploited in that protocol...
- **AND** you're still not happy with the efficiency of pairings...
- **AND** you have a little more storage space...

- **THEN** employ some conceptually simple pre-computation and enjoy the (up to 37%) speedups $\ddot\smile$