Efficiently computing a pairing: Tricks old and new.

Michael Scott

tii.ae

July 2023

Implementing Pairing-Based Cryptography

- We denote a pairing as $g = e(P, Q)$, where P is in one elliptic curve group, Q is in another elliptic curve group, and g is a group over a finite extension field.

Implementing Pairing-Based Cryptography

- We denote a pairing as $g = e(P, Q)$, where P is in one elliptic curve group, Q is in another elliptic curve group, and g is a group over a finite extension field.
- All these groups are of the same order r. The field extension is denoted k, a.k.a. the embedding degree.

Implementing Pairing-Based Cryptography

- We denote a pairing as $g = e(P, Q)$, where P is in one elliptic curve group, Q is in another elliptic curve group, and g is a group over a finite extension field.
- All these groups are of the same order r. The field extension is denoted k, a.k.a. the embedding degree.
- There are different types of pairing (Tate, Ate, Weil...). Useful pairings are based on either supersingular curves (type 1, limited choice of k), or special pairing-friendly curves (type 3, unlimited k).

Implementing Pairing-Based Cryptography

- We denote a pairing as $g = e(P, Q)$, where P is in one elliptic curve group, Q is in another elliptic curve group, and g is a group over a finite extension field.
- All these groups are of the same order r. The field extension is denoted k, a.k.a. the embedding degree.
- There are different types of pairing (Tate, Ate, Weil...). Useful pairings are based on either supersingular curves (type 1, limited choice of k), or special pairing-friendly curves (type 3, unlimited k).
- The typical structure of a pairing implementation is a Miller loop, followed by a final exponentiation. These can each in turn be subdivided into smaller steps.

Implementing Pairing-Based Cryptography

- We denote a pairing as $g = e(P, Q)$, where P is in one elliptic curve group, Q is in another elliptic curve group, and g is a group over a finite extension field.
- All these groups are of the same order r. The field extension is denoted k, a.k.a. the embedding degree.
- There are different types of pairing (Tate, Ate, Weil...). Useful pairings are based on either supersingular curves (type 1, limited choice of k), or special pairing-friendly curves (type 3, unlimited k).
- The typical structure of a pairing implementation is a Miller loop, followed by a final exponentiation. These can each in turn be subdivided into smaller steps.
- For example the final exponentiation can be divided into an "easy" part and a "hard" part.

Implementing Pairing-Based Cryptography

- We denote a pairing as $g = e(P, Q)$, where P is in one elliptic curve group, Q is in another elliptic curve group, and g is a group over a finite extension field.
- All these groups are of the same order r. The field extension is denoted k, a.k.a. the embedding degree.
- There are different types of pairing (Tate, Ate, Weil...). Useful pairings are based on either supersingular curves (type 1, limited choice of k), or special pairing-friendly curves (type 3, unlimited k).
- The typical structure of a pairing implementation is a Miller loop, followed by a final exponentiation. These can each in turn be subdivided into smaller steps.
- For example the final exponentiation can be divided into an "easy" part and a "hard" part.
- In this talk we will focus attention on the Miller loop, and assume either the Tate or Ate pairing.

The Miller Loop

Algorithm 1: Miller loop

Input: $Q \in \mathbb{G}_2$, $P \in \mathbb{G}_1$, curve parameter u

Output: $f \in \mathbb{F}_{p^k}$

$f \leftarrow 1$

$T \leftarrow Q$

for $i \leftarrow \lfloor \log_2(u) \rfloor - 1$ **to** 0 **do**

  $f \leftarrow f^2.l_{T,T}(P),\ T \leftarrow 2T$

  if $u_i = 1$ **then**

    $f \leftarrow f.l_{T,Q}(P),\ T \leftarrow T + Q$

return f

What is going on?

- The point Q is undergoing a classic left-to-right double-and-add point multiplication by u. The point P is fixed.

What is going on?

- The point Q is undergoing a classic left-to-right double-and-add point multiplication by u. The point P is fixed.
- The line function is undergoing something rather like a left-to-right square-and-multiply algorithm. But not quite!

What is going on?

- The point Q is undergoing a classic left-to-right double-and-add point multiplication by u. The point P is fixed.
- The line function is undergoing something rather like a left-to-right square-and-multiply algorithm. But not quite!
- Every time around the loop f is being squared and multiplied by a line function, once, maybe twice.

What is going on?

- The point Q is undergoing a classic left-to-right double-and-add point multiplication by u. The point P is fixed.
- The line function is undergoing something rather like a left-to-right square-and-multiply algorithm. But not quite!
- Every time around the loop f is being squared and multiplied by a line function, once, maybe twice.
- The line functions are often sparse elements $\in \mathbb{F}_{p^k}$

What is going on?

- The point Q is undergoing a classic left-to-right double-and-add point multiplication by u. The point P is fixed.
- The line function is undergoing something rather like a left-to-right square-and-multiply algorithm. But not quite!
- Every time around the loop f is being squared and multiplied by a line function, once, maybe twice.
- The line functions are often sparse elements $\in \mathbb{F}_{p^k}$
- Important to observe that u is a fixed system parameter, and not a variable.

What is going on?

- The point Q is undergoing a classic left-to-right double-and-add point multiplication by u. The point P is fixed.
- The line function is undergoing something rather like a left-to-right square-and-multiply algorithm. But not quite!
- Every time around the loop f is being squared and multiplied by a line function, once, maybe twice.
- The line functions are often sparse elements $\in \mathbb{F}_{p^k}$
- Important to observe that u is a fixed system parameter, and not a variable.
- As described we are assuming denominator elimination (DE) applies, Barreto et al. [2002],

How many bits in u?

- If the pairing in question is the Tate pairing, then the curve parameter u is simply the group order.

How many bits in u?

- If the pairing in question is the Tate pairing, then the curve parameter u is simply the group order.
- Clearly this gets bigger as the security level of the pairing increases

How many bits in u?

- If the pairing in question is the Tate pairing, then the curve parameter u is simply the group order.
- Clearly this gets bigger as the security level of the pairing increases
- However for the Ate pairing, rather counter-intuitively, the parameter u actually decreases with increased security.

How many bits in u?

- If the pairing in question is the Tate pairing, then the curve parameter u is simply the group order.
- Clearly this gets bigger as the security level of the pairing increases
- However for the Ate pairing, rather counter-intuitively, the parameter u actually decreases with increased security.
- For example for the BLS12-381 $u = d201000000010000$, for the BLS48-581 curve $u = 140000381$.

How many bits in u?

- If the pairing in question is the Tate pairing, then the curve parameter u is simply the group order.
- Clearly this gets bigger as the security level of the pairing increases
- However for the Ate pairing, rather counter-intuitively, the parameter u actually decreases with increased security.
- For example for the BLS12-381 $u = d201000000010000$, for the BLS48-581 curve $u = 140000381$.
- So the Miller loop gets shorter, and in most cases of interest loops less than about 64 times.

What does u look like?

- It turns out that since u is a system parameter it can often be chosen to be extremely sparse.

What does u look like?

- It turns out that since u is a system parameter it can often be chosen to be extremely sparse.
- Which brings obvious advantages, as the "if" clause in the Miller algorithm will then rarely be executed.

What does u look like?

- It turns out that since u is a system parameter it can often be chosen to be extremely sparse.
- Which brings obvious advantages, as the "if" clause in the Miller algorithm will then rarely be executed.
- But this is not always the case. For example so-called MNT curves arise from rare solutions to a Pell equation, in which case we have little control over u.

What does u look like?

- It turns out that since u is a system parameter it can often be chosen to be extremely sparse.
- Which brings obvious advantages, as the “if” clause in the Miller algorithm will then rarely be executed.
- But this is not always the case. For example so-called MNT curves arise from rare solutions to a Pell equation, in which case we have little control over u.
- It also arises when the group order for the Tate pairing is required to be a composite.

What does u look like?

- It turns out that since u is a system parameter it can often be chosen to be extremely sparse.
- Which brings obvious advantages, as the "if" clause in the Miller algorithm will then rarely be executed.
- But this is not always the case. For example so-called MNT curves arise from rare solutions to a Pell equation, in which case we have little control over u.
- It also arises when the group order for the Tate pairing is required to be a composite.
- (I had rather hoped that David Freeman had saved us from that. Then along came the isogenists...)

What happens to Q?

- The point Q is in effect multiplied by u, at the end of the loop $T = uQ$.

What happens to Q?

- The point Q is in effect multiplied by u, at the end of the loop $T = uQ$.
- In the case of the Tate pairing, u is the group order, so the final value of T will be the point at infinity.

What happens to Q?

- The point Q is in effect multiplied by u, at the end of the loop $T = uQ$.
- In the case of the Tate pairing, u is the group order, so the final value of T will be the point at infinity.
- So we get a "free" check that Q is of the correct order!

What happens to Q?

- The point Q is in effect multiplied by u, at the end of the loop $T = uQ$.
- In the case of the Tate pairing, u is the group order, so the final value of T will be the point at infinity.
- So we get a "free" check that Q is of the correct order!
- Less obviously the free group order check on Q also applies to the Ate pairing. See S. "A note on group membership tests for $\mathbb{G}_1$, $\mathbb{G}_2$ and $\mathbb{G}_T$ on BLS pairing-friendly curves".

Let's split the Miller loop in two

Algorithm 2: Calculate and store line functions

```
Input: Q ∈ 𝔾₂, P ∈ 𝔾₁, curve parameter u
Output: An array g of ⌊log₂(u)⌋ line functions ∈ 𝔽_{p^k}
T ← Q
for i ← ⌊log₂(u)⌋ − 1 to 0 do
    g[i] ← l_{T,T}(P), T ← 2T
    if u_i = 1 then
        g[i] ← g[i].l_{T,Q}(P), T ← T + Q
return g
```

Algorithm 3: Intrinsic Miller loop

```
Input: An array g of ⌊log₂(u)⌋ line functions ∈ 𝔽_{p^k}
Output: f ∈ 𝔽_{p^k}
f ← 1
for i ← ⌊log₂(u)⌋ − 1 to 0 do
    f ← f².g[i]
return f
```

What's new?

- In algorithm 2 the line functions are precalculated and stored. The amount of storage required is modest.

What's new?

- In algorithm 2 the line functions are precalculated and stored. The amount of storage required is modest.
- Note that for a single pairing the computation required is identical to that required by the original Miller loop.

What's new?

- In algorithm 2 the line functions are precalculated and stored. The amount of storage required is modest.
- Note that for a single pairing the computation required is identical to that required by the original Miller loop.
- In a multi-pairing context all of the line functions for each of the pairings can be accumulated into a single g array.

What's new?

- In algorithm 2 the line functions are precalculated and stored. The amount of storage required is modest.
- Note that for a single pairing the computation required is identical to that required by the original Miller loop.
- In a multi-pairing context all of the line functions for each of the pairings can be accumulated into a single g array.
- So algorithm 2 will be executed for each of the pairings in a multi-pairing. Since they all share the same u these executions all take place in "lock-step".

What's new?

- In algorithm 2 the line functions are precalculated and stored. The amount of storage required is modest.
- Note that for a single pairing the computation required is identical to that required by the original Miller loop.
- In a multi-pairing context all of the line functions for each of the pairings can be accumulated into a single g array.
- So algorithm 2 will be executed for each of the pairings in a multi-pairing. Since they all share the same u these executions all take place in "lock-step".
- Algorithm 3 is only run once, independent of the number of pairings. Which also applies to the final exponentiation.

Optimizations?

- Clearly not much can be done for algorithm 3.

Optimizations?

- Clearly not much can be done for algorithm 3.
- For a single pairing, the sparsity of g elements can be exploited in algorithm 2.

Optimizations?

- Clearly not much can be done for algorithm 3.
- For a single pairing, the sparsity of g elements can be exploited in algorithm 2.
- However in a multi-pairing context such sparsity is quickly wiped out as contributions from algorithm 2 are accumulated in g.

Optimizations?

- Clearly not much can be done for algorithm 3.
- For a single pairing, the sparsity of g elements can be exploited in algorithm 2.
- However in a multi-pairing context such sparsity is quickly wiped out as contributions from algorithm 2 are accumulated in g.
- Looked at in this way, it can be seen that the cost of the Miller loop cannot be reduced below the requirement of algorithm 3.

Optimizations?

- Clearly not much can be done for algorithm 3.
- For a single pairing, the sparsity of g elements can be exploited in algorithm 2.
- However in a multi-pairing context such sparsity is quickly wiped out as contributions from algorithm 2 are accumulated in g.
- Looked at in this way, it can be seen that the cost of the Miller loop cannot be reduced below the requirement of algorithm 3.
- Algorithm 2 on the other hand is rich in optimization possibilities....

Optimizing Algorithm 2

- In a multi-pairing much depends on the provenance of Q.

Optimizing Algorithm 2

- In a multi-pairing much depends on the provenance of Q.
- For example if it were a constant, its multiples can be precomputed and stored in affine coordinates

Optimizing Algorithm 2

- In a multi-pairing much depends on the provenance of Q.
- For example if it were a constant, its multiples can be precomputed and stored in affine coordinates
- And using affine coordinates results in increased sparsity of the line functions.

Optimizing Algorithm 2

- In a multi-pairing much depends on the provenance of Q.
- For example if it were a constant, its multiples can be precomputed and stored in affine coordinates
- And using affine coordinates results in increased sparsity of the line functions.
- So algorithm 2 can be carefully tuned to the particular context of each individual pairing in a multi-pairing.

What about windowing...

- In the context where u is not sparse, it would seem obvious to deploy a windowing algorithm, as commonly used in a double-and-add context.

What about windowing...

- In the context where u is not sparse, it would seem obvious to deploy a windowing algorithm, as commonly used in a double-and-add context.
- So why not apply windowing to algorithm 1? This has an interesting history...

What about windowing...

- In the context where u is not sparse, it would seem obvious to deploy a windowing algorithm, as commonly used in a double-and-add context.
- So why not apply windowing to algorithm 1? This has an interesting history...
- In a very early paper on pairings by Galbraith et al [2002] it was stated in the context of windowing Miller's algorithm that "The methods are completely standard... and it is not neessary to repeat them here".

What about windowing...

- In the context where u is not sparse, it would seem obvious to deploy a windowing algorithm, as commonly used in a double-and-add context.
- So why not apply windowing to algorithm 1? This has an interesting history...
- In a very early paper on pairings by Galbraith et al [2002] it was stated in the context of windowing Miller's algorithm that "The methods are completely standard... and it is not neessary to repeat them here".
- But whereas the application to the multiplication of Q by u is standard, the impact on the line functions is not entirely obvious.

What about windowing...

- In the context where u is not sparse, it would seem obvious to deploy a windowing algorithm, as commonly used in a double-and-add context.
- So why not apply windowing to algorithm 1? This has an interesting history...
- In a very early paper on pairings by Galbraith et al [2002] it was stated in the context of windowing Miller's algorithm that "The methods are completely standard... and it is not neessary to repeat them here".
- But whereas the application to the multiplication of Q by u is standard, the impact on the line functions is not entirely obvious.
- The first implementation was I believe by myself, as mentioned in the pre-print S. [2005] "Scaling security in pairing-based protocols"

Let's window

- The details were soon after worked out and published by Kobayishi et al. [2006] "Efficient Algorithms for Tate pairing".

Let's window

- The details were soon after worked out and published by Kobayishi et al. [2006] "Efficient Algorithms for Tate pairing".
- The performance benefits were researched in greater detail in the paper by Kiyomura and Takagi [2012] "Efficient Algorithm for Tate Pairing of Composite Order" (which is behind a pay-wall, has attracted 0 citations, so I think its fair to say that these results are not widely known)

Let's window

- The details were soon after worked out and published by Kobayishi et al. [2006] "Efficient Algorithms for Tate pairing".
- The performance benefits were researched in greater detail in the paper by Kiyomura and Takagi [2012] "Efficient Algorithm for Tate Pairing of Composite Order" (which is behind a pay-wall, has attracted 0 citations, so I think its fair to say that these results are not widely known)
- Indeed an early paper appeared to overlook the possible benefits of windowing when applied to composite order pairings (Guillevic [2013] "Comparing the pairing efficiency over composite order and prime order elliptic curves")

Let's window

- The details were soon after worked out and published by Kobayishi et al. [2006] "Efficient Algorithms for Tate pairing".
- The performance benefits were researched in greater detail in the paper by Kiyomura and Takagi [2012] "Efficient Algorithm for Tate Pairing of Composite Order" (which is behind a pay-wall, has attracted 0 citations, so I think its fair to say that these results are not widely known)
- Indeed an early paper appeared to overlook the possible benefits of windowing when applied to composite order pairings (Guillevic [2013] "Comparing the pairing efficiency over composite order and prime order elliptic curves")
- We can exploit the fact that negation of elliptic curve points cost nothing. Similarly inversion of line functions cost little, as inversion can be replaced by conjugation (DE).

Let's window

- The details were soon after worked out and published by Kobayishi et al. [2006] "Efficient Algorithms for Tate pairing".
- The performance benefits were researched in greater detail in the paper by Kiyomura and Takagi [2012] "Efficient Algorithm for Tate Pairing of Composite Order" (which is behind a pay-wall, has attracted 0 citations, so I think its fair to say that these results are not widely known)
- Indeed an early paper appeared to overlook the possible benefits of windowing when applied to composite order pairings (Guillevic [2013] "Comparing the pairing efficiency over composite order and prime order elliptic curves")
- We can exploit the fact that negation of elliptic curve points cost nothing. Similarly inversion of line functions cost little, as inversion can be replaced by conjugation (DE).
- Hence a windowing strategy based on a NAF (Non-Adjacent Form) is appropriate. Since u is a public parameter constant-time considerations are not an issue, hence a sliding-windows algorithm can be used.

Line functions

- The key identity that arises from divisor theory is $f_{i+j} = f_i f_j l_{iQ,jQ}(P)$, with $f_1 = 1$.

Line functions

- The key identity that arises from divisor theory is $f_{i+j} = f_i f_j l_{iQ,jQ}(P)$, with $f_1 = 1$.
- To minimize algorithmic clutter, we will drop the fixed parameter (P)

Line functions

- The key identity that arises from divisor theory is $f_{i+j} = f_i f_j l_{iQ,jQ}(P)$, with $f_1 = 1$.
- To minimize algorithmic clutter, we will drop the fixed parameter (P)
- For use in a double-and-add left-to-right context we will consider this identity in two particular cases

$$f_{m+m} = {f_m}^2 . l_{mQ,mQ}$$

$$f_{m+1} = f_m . l_{mQ,Q}$$

Line functions

- The key identity that arises from divisor theory is $f_{i+j} = f_i f_j l_{iQ,jQ}(P)$, with $f_1 = 1$.
- To minimize algorithmic clutter, we will drop the fixed parameter (P)
- For use in a double-and-add left-to-right context we will consider this identity in two particular cases

$$f_{m+m} = {f_m}^2 . l_{mQ,mQ}$$

$$f_{m+1} = f_m . l_{mQ,Q}$$

- Observe that the "squaring" step is more expensive than the "multiply" step.

Line functions

- The key identity that arises from divisor theory is $f_{i+j} = f_i f_j l_{iQ,jQ}(P)$, with $f_1 = 1$.
- To minimize algorithmic clutter, we will drop the fixed parameter (P)
- For use in a double-and-add left-to-right context we will consider this identity in two particular cases

$$f_{m+m} = {f_m}^2 . l_{mQ,mQ}$$

$$f_{m+1} = f_m . l_{mQ,Q}$$

- Observe that the "squaring" step is more expensive than the "multiply" step.
- Which is bad news, as windowing (which reduces the number of multiplies) works best when squaring is cheaper.

Working out the details

- Consider the case where two set bits of u are being processed... Instead of calculating

$$
\begin{aligned}
f_{2m} &= {f_m}^2 . l_{mQ,mQ} \\
f_{2m+1} &= f_{2m} . l_{2mQ,Q} \\
f_{4m+2} &= {f_{2m+1}}^2 . l_{2mQ+Q,2mQ+Q} \\
f_{4m+3} &= f_{4m+2} . l_{4mQ+2Q,Q}
\end{aligned}
\tag{1}
$$

Working out the details

- Consider the case where two set bits of u are being processed... Instead of calculating

$$\begin{aligned} f_{2m} &= {f_m}^2 . l_{mQ,mQ} \\ f_{2m+1} &= f_{2m} . l_{2mQ,Q} \\ f_{4m+2} &= {f_{2m+1}}^2 . l_{2mQ+Q,2mQ+Q} \\ f_{4m+3} &= f_{4m+2} . l_{4mQ+2Q,Q} \end{aligned} \tag{1}$$

- We will calculate

$$\begin{aligned} f_{2m} &= {f_m}^2 . l_{mQ,mQ} \\ f_{4m} &= {f_{2m}}^2 . l_{2mQ,2mQ} \\ f_{4m+3} &= f_{4m} . l_{4mQ,3Q} . f_3 \end{aligned} \tag{2}$$

Working out the details

- Consider the case where two set bits of u are being processed... Instead of calculating

$$
\begin{aligned}
f_{2m} &= {f_m}^2.l_{mQ,mQ} \\
f_{2m+1} &= f_{2m}.l_{2mQ,Q} \\
f_{4m+2} &= {f_{2m+1}}^2.l_{2mQ+Q,2mQ+Q} \\
f_{4m+3} &= f_{4m+2}.l_{4mQ+2Q,Q}
\end{aligned} \tag{1}
$$

- We will calculate

$$
\begin{aligned}
f_{2m} &= {f_m}^2.l_{mQ,mQ} \\
f_{4m} &= {f_{2m}}^2.l_{2mQ,2mQ} \\
f_{4m+3} &= f_{4m}.l_{4mQ,3Q}.f_3
\end{aligned} \tag{2}
$$

- which will require the precomputation of $3Q$ and f_3

Getting ready for a NAF

- It is also easy to show that

$$f_{8m-3} = f_{8m}.l_{8mQ,-3Q}/f_3$$

Getting ready for a NAF

- It is also easy to show that

$$f_{8m-3} = f_{8m}.l_{8mQ,-3Q}/f_3$$

- which due to DE can be replaced by

$$f_{8m-3} = f_{8m}.l_{8mQ,-3Q}.\bar{f}_3$$

Getting ready for a NAF

- It is also easy to show that

$$f_{8m-3} = f_{8m}.l_{8mQ,-3Q}/f_3$$

- which due to DE can be replaced by

$$f_{8m-3} = f_{8m}.l_{8mQ,-3Q}.\bar{f}_3$$

- Extending the idea, a sliding window of size w bits will require the precomputation of a table E of size M, containing the precomputed points $Q, 3Q, ..(2M-1)Q$ and a table F containing $f_1, f_3, ..f_{2M-1}$, where $F_0 = f_1 = 1$.

Precomputation

- The line function table is precomputed as

$$F_i = F_{i-1}.l_{Q,Q}.l_{E_i,2Q}$$

Precomputation

- The line function table is precomputed as

$$F_i = F_{i-1}.l_{Q,Q}.l_{E_i,2Q}$$

- and the table size M is

$$M = 1 + \Sigma_{i=1}^{(w-1)/2} \; 2^{2i-(w \bmod 2)}$$

Precomputation

- The line function table is precomputed as

$$F_i = F_{i-1}.l_{Q,Q}.l_{E_i,2Q}$$

- and the table size M is

$$M = 1 + \Sigma_{i=1}^{(w-1)/2} \; 2^{2i-(w \bmod 2)}$$

- To facilitate the sliding window, assume a function `naf_window`, which given $s = 3u \oplus u$ (the bit-by-bit exclusive or) and a pointer i to the current bit position scans bits from left-to-right returning the tuple $\{n, b, z\}$ where n is the odd signed window value, b is the number of bits processed and z is the number of subsequent zero bits.

Windowed Miller Loop

Algorithm 4: Windowed Miller Loop for Tate pairing

Input: $P \in \mathbb{G}_1$, $Q \in \mathbb{G}_2$, curve parameter u
Output: $f \in \mathbb{F}_{p^k}$

$f \leftarrow 1$
$T \leftarrow P$
$s \leftarrow 3u \oplus u$
$i \leftarrow \lfloor \log_2(u) \rfloor$
while $i > 0$ **do**
    $n, b, z \leftarrow \texttt{naf\_window}(s, i)$
    for $j \leftarrow 0$ **to** b **do**
        $f \leftarrow f^2.l_{T,T}$, $T \leftarrow 2T$
    if $n > 0$ **then**
        $f \leftarrow f.l_{T,E[n/2]}.F[n/2]$, $T \leftarrow T + E[n/2]$
    if $n < 0$ **then**
        $f \leftarrow f.l_{T,-E[-n/2]}.\overline{F[-n/2]}$, $T \leftarrow T - E[-n/2]$
    for $j \leftarrow 0$ **to** z **do**
        $f \leftarrow f^2.l_{T,T}$, $T \leftarrow 2T$
    $i \leftarrow i - b - z$
return f

Thoughts

- Again the loop can be "split", and the contribution of the line functions accumulated and stored, one for each window.

Thoughts

- Again the loop can be "split", and the contribution of the line functions accumulated and stored, one for each window.
- The accumulated outputs from a multi-pairing could finally be fed into something like our algorithm 3, where the loop length would be shortened to the number of windows required for a particular u.

Thoughts

- Again the loop can be "split", and the contribution of the line functions accumulated and stored, one for each window.
- The accumulated outputs from a multi-pairing could finally be fed into something like our algorithm 3, where the loop length would be shortened to the number of windows required for a particular u.
- We omit the details

Bottom line

- For a Tate pairing over a 1024-bit supersingular curve with embedding degree $k = 2$, where the group order is a 1022-bit RSA public key, we find that the optimal window size is between 5 and 6. The performance improvement from using a window of size 5 is approximately 8%.

Bottom line

- For a Tate pairing over a 1024-bit supersingular curve with embedding degree $k = 2$, where the group order is a 1022-bit RSA public key, we find that the optimal window size is between 5 and 6. The performance improvement from using a window of size 5 is approximately 8%.
- For the Tate pairing on a 160-bit MNT $k = 6$ curve we find that the the optimal window size is 3. The performance improvement to be expected is about 3%. For the Ate pairing over the same curve again the optimal window size is 3, but improvement is a nearly negligible 1%. Clearly the larger the exponent, the greater the gains to be expected from windowing.

Any Questions?

- Any questions?

Any Questions?

- Any questions?
- Thank you for your attention.