PandA: Pairings and Arithmetic

Peter Schwabe

Radboud University, Nijmegen, The Netherlands

Joint work with Chitchanok Chuengsatiansup, Michael Naehrig, Pance Ribarski

August 4, 2015

SIAM AG15, Mini Symposium on Pairings in Cryptography

Back in 2010...

- Naehrig, Niederhagen, Schwabe, 2010: New speed record for pairing over BN curves on Intel Core 2
  Software: http://cryptojedi.org/crypto/#dclxvi

Back in 2010...

- Naehrig, Niederhagen, Schwabe, 2010: New speed record for pairing over BN curves on Intel Core 2
  Software: http://cryptojedi.org/crypto/#dclxvi
- Soon beaten by Beuchat, González Díaz, Mitsunari, Okamoto, Rodríguez-Henríquez, Teruya (2010)

Back in 2010...

- Naehrig, Niederhagen, Schwabe, 2010: New speed record for pairing over BN curves on Intel Core 2
  Software: http://cryptojedi.org/crypto/#dclxvi
- Soon beaten by Beuchat, González Díaz, Mitsunari, Okamoto, Rodríguez-Henríquez, Teruya (2010)
- Soon beaten by Aranha, Karabina, Longa, Gebotys, López (2011)

Back in 2010...

- Naehrig, Niederhagen, Schwabe, 2010: New speed record for pairing over BN curves on Intel Core 2
  Software: http://cryptojedi.org/crypto/#dclxvi
- Soon beaten by Beuchat, González Díaz, Mitsunari, Okamoto, Rodríguez-Henríquez, Teruya (2010)
- Soon beaten by Aranha, Karabina, Longa, Gebotys, López (2011)
- Most pairing-based *protocols* do not use these improvements
- Reasons:
  - Protocols need more: arithmetic in groups, hashing to groups, etc.
  - Pairing computation is often not the bottleneck (anymore)
  - Protocols need easily accessible software
  - Protocols (sometimes) need Type-1 pairings

eBACS

- Widely used public benchmarking for cryptography: eBACS
  http://bench.cr.yp.to/

eBACS

- Widely used public benchmarking for cryptography: eBACS
  http://bench.cr.yp.to/
- eBACS supports bechmarking of cryptographic primitives, i.e.,
  - hash functions,
  - stream ciphers,
  - Diffie-Hellman key exchange,
  - signatures,
  - public-key encryption

eBACS

- Widely used public benchmarking for cryptography: eBACS
  http://bench.cr.yp.to/
- eBACS supports bechmarking of cryptographic primitives, i.e.,
  - hash functions,
  - stream ciphers,
  - Diffie-Hellman key exchange,
  - signatures,
  - public-key encryption
- "Low-level" benchmarking of scalar multiplication
- No benchmarking of pairings and full group arithmetic

Introducing PandA

- Idea: “extending eBACS API to low level”
- API and framework to test and benchmark
  - pairings
  - arithmetic in groups
  - arithmetic on scalars modulo the group order

Introducing PandA

- Idea: “extending eBACS API to low level”
- API and framework to test and benchmark
  - pairings
  - arithmetic in groups
  - arithmetic on scalars modulo the group order
- Benchmark results enable sensible protocol design
- Implementations in PandA can be compiled to a library
- Easy access to high-performance implementations for protocol implementations

Introducing PandA

- Idea: "extending eBACS API to low level"
- API and framework to test and benchmark
  - pairings
  - arithmetic in groups
  - arithmetic on scalars modulo the group order
- Benchmark results enable sensible protocol design
- Implementations in PandA can be compiled to a library
- Easy access to high-performance implementations for protocol implementations

Software bridge between two communities

API design, first considerations

C language

- Same language as eBACS API
- Typical language for high-speed implementations (+assembly)
- Use `bgroup` namespace (`group` for non-bilinear groups)

API design, first considerations

C language

- Same language as eBACS API
- Typical language for high-speed implementations (+assembly)
- Use `bgroup` namespace (`group` for non-bilinear groups)

Additive groups

- Pairing is map $e : G_1 \times G_2 \to G_3$
- Unify notation for G_1, G_2 and G_3: write them all additively
- Follows `crypto_scalarmult` API in eBACS

API design, first considerations

C language

- Same language as eBACS API
- Typical language for high-speed implementations (+assembly)
- Use `bgroup` namespace (`group` for non-bilinear groups)

Additive groups

- Pairing is map $e : G_1 \times G_2 \to G_3$
- Unify notation for G_1, G_2 and G_3: write them all additively
- Follows `crypto_scalarmult` API in eBACS

Type-I pairings

- Easily support Type-1 (symmetric) pairings:

```
#define BGROUP_TYPE1
```

Constant-time software

- Major threat to cryptographic software: **timing attacks**

Constant-time software

- Major threat to cryptographic software: **timing attacks**
- Osvik, Shamir, Tromer, 2006: 65 ms to steal Linux `dmcrypt` AES key used for hard-disk encryption

Constant-time software

- Major threat to cryptographic software: **timing attacks**
- Osvik, Shamir, Tromer, 2006: 65 ms to steal Linux `dmcrypt` AES key used for hard-disk encryption
- Brumley and Tuveri in 2011: A few minutes to steal OpenSSL ECDSA key *over the network*

Constant-time software

- Major threat to cryptographic software: **timing attacks**
- Osvik, Shamir, Tromer, 2006: 65 ms to steal Linux `dmcrypt` AES key used for hard-disk encryption
- Brumley and Tuveri in 2011: A few minutes to steal OpenSSL ECDSA key *over the network*
- Protection against such attacks: constant-time software
- Protection requires redesigning algorithms on all levels
- Hard to add later
- Starting with non-constant-time algorithms may lead to wrong protocol-design decisions

Constant-time software

- Major threat to cryptographic software: **timing attacks**
- Osvik, Shamir, Tromer, 2006: 65 ms to steal Linux `dmcrypt` AES key used for hard-disk encryption
- Brumley and Tuveri in 2011: A few minutes to steal OpenSSL ECDSA key *over the network*
- Protection against such attacks: constant-time software
- Protection requires redesigning algorithms on all levels
- Hard to add later
- Starting with non-constant-time algorithms may lead to wrong protocol-design decisions
- PandA has constant-time algorithms by default
- Faster non-constant-time versions through `_publicinputs` versions of functions

Data types, packing and unpacking

- Implementation defines 4 data types:

  `bgroup_g1e`
  `bgroup_g2e`
  `bgroup_g3e`
  `bgroup_scalar`

Data types, packing and unpacking

- Implementation defines 4 data types:

```
bgroup_g1e
bgroup_g2e
bgroup_g3e
bgroup_scalar
```

- This talk: consider G_1 functions, equivalent functions for G_2 and G_3
- Conversion to and from byte arrays ("wire format"):

```
void bgroup_g1e_pack(unsigned char r[BGROUP_G1E_PACKEDBYTES],
                     const bgroup_g1e *b);
int bgroup_g1e_unpack(bgroup_g1e *r,
                      const unsigned char b[BGROUP_G1E_PACKEDBYTES]);
```

Data types, packing and unpacking

- Implementation defines 4 data types:

```
bgroup_g1e
bgroup_g2e
bgroup_g3e
bgroup_scalar
```

- This talk: consider G_1 functions, equivalent functions for G_2 and G_3
- Conversion to and from byte arrays ("wire format"):

```
void bgroup_g1e_pack(unsigned char r[BGROUP_G1E_PACKEDBYTES],
                     const bgroup_g1e *b);
int bgroup_g1e_unpack(bgroup_g1e *r,
                      const unsigned char b[BGROUP_G1E_PACKEDBYTES]);
```

- Return nonzero for invalid byte array in `unpack`

Data types, packing and unpacking

- Implementation defines 4 data types:

```
bgroup_g1e
bgroup_g2e
bgroup_g3e
bgroup_scalar
```

- This talk: consider G_1 functions, equivalent functions for G_2 and G_3
- Conversion to and from byte arrays ("wire format"):

```
void bgroup_g1e_pack(unsigned char r[BGROUP_G1E_PACKEDBYTES],
                     const bgroup_g1e *b);
int bgroup_g1e_unpack(bgroup_g1e *r,
                      const unsigned char b[BGROUP_G1E_PACKEDBYTES]);
```

- Return nonzero for invalid byte array in `unpack`
- Define constants

```
bgroup_g1e bgroup_g1e_neutral;
bgroup_g1e bgroup_g1e_base;
```

Comparison

- Easy way to compare: `bgroup_g1e_pack` + `memcmp`
- Only efficient if packing is required anyway

Comparison

- Easy way to compare: `bgroup_g1e_pack` + `memcmp`
- Only efficient if packing is required anyway
- Faster alternative:

```
int bgroup_g1e_equals(const bgroup_g1e *a,
                      const bgroup_g1e *b);
```

Comparison

- Easy way to compare: `bgroup_g1e_pack` + `memcmp`
- Only efficient if packing is required anyway
- Faster alternative:

```
int bgroup_g1e_equals(const bgroup_g1e *a,
                      const bgroup_g1e *b);
```

- Non-constant-time version:

```
int bgroup_g1e_equals_publicinputs(const bgroup_g1e *a,
                                   const bgroup_g1e *b);
```

Comparison

- Easy way to compare: `bgroup_g1e_pack` + `memcmp`
- Only efficient if packing is required anyway
- Faster alternative:

```
int bgroup_g1e_equals(const bgroup_g1e *a,
                      const bgroup_g1e *b);
```

- Non-constant-time version:

```
int bgroup_g1e_equals_publicinputs(const bgroup_g1e *a,
                                   const bgroup_g1e *b);
```

- From now on don't mention non-constant-time versions anymore

Addition and doubling

- Addition, doubling and negation:

```
void bgroup_g1e_add(bgroup_g1e *r,
                    const bgroup_g1e *a,
                    const bgroup_g1e *b);
void bgroup_g1e_double(bgroup_g1e *r,
                       const bgroup_g1e *a);
void bgroup_g1e_negate(bgroup_g1e *r,
                       const bgroup_g1e *a);
```

Addition and doubling

- Addition, doubling and negation:

```
void bgroup_g1e_add(bgroup_g1e *r,
                    const bgroup_g1e *a,
                    const bgroup_g1e *b);
void bgroup_g1e_double(bgroup_g1e *r,
                       const bgroup_g1e *a);
void bgroup_g1e_negate(bgroup_g1e *r,
                       const bgroup_g1e *a);
```

- Functions need to work for *all* inputs in constant time

Scalar multiplication

- "Standard" scalar multiplication:

```
void bgroup_g1e_scalarmult(bgroup_g1e *r,
                           const bgroup_g1e *a,
                           const bgroup_scalar *s);
```

Scalar multiplication

- "Standard" scalar multiplication:

```
void bgroup_g1e_scalarmult(bgroup_g1e *r,
                           const bgroup_g1e *a,
                           const bgroup_scalar *s);
```

- Fixed base-point scalar multiplication:

```
void bgroup_g1e_scalarmult(bgroup_g1e *r,
                           const bgroup_scalar *s);
```

Scalar multiplication

- "Standard" scalar multiplication:

```
void bgroup_g1e_scalarmult(bgroup_g1e *r,
                           const bgroup_g1e *a,
                           const bgroup_scalar *s);
```

- Fixed base-point scalar multiplication:

```
void bgroup_g1e_scalarmult(bgroup_g1e *r,
                           const bgroup_scalar *s);
```

- Multi-scalar multiplication:

```
void bgroup_g1e_multiscalarmult(bgroup_g1e *r,
                                const bgroup_g1e *a,
                                const bgroup_scalar *s,
                                unsigned long long alen);
```

Hashing to G_1 and G_2

```
void bgroup_g1e_hashfromstr(bgroup_g1e *r,
                            const unsigned char *a,
                            unsigned long long alen);

void bgroup_g2e_hashfromstr(bgroup_g2e *r,
                            const unsigned char *a,
                            unsigned long long alen);
```

Hashing to G_1 and G_2

```
void bgroup_g1e_hashfromstr(bgroup_g1e *r,
                            const unsigned char *a,
                            unsigned long long alen);

void bgroup_g2e_hashfromstr(bgroup_g2e *r,
                            const unsigned char *a,
                            unsigned long long alen);
```

- Only functions where `_publicinputs` version does not need to compute the same result

Arithmetic on scalars

- Support addition, subtraction, multiplication, squaring, inversion, packing/unpacking of scalars modulo the group order
- Support setting scalar to a random value
- Example:

```
void bgroup_scalar_add(bgroup_scalar *r,
                       const bgroup_scalar *s,
                       const bgroup_scalar *t);
```

Arithmetic on scalars

- Support addition, subtraction, multiplication, squaring, inversion, packing/unpacking of scalars modulo the group order
- Support setting scalar to a random value
- Example:

```
void bgroup_scalar_add(bgroup_scalar *r,
                       const bgroup_scalar *s,
                       const bgroup_scalar *t);
```

- So far: No non-constant-time versions of scalar arithmetic
- We do not expect huge non-constant-time speedups
- Scalar arithmetic is *typically* not a performance bottleneck

Pairings

- "Standard" (single) pairing:

```
void bgroup_pairing(bgroup_g3e *r,
                    const bgroup_g1e *a,
                    const bgroup_g2e *b);
```

- Pairing of `bgroup_g1e_base` and `bgroup_g2e_base` must yield `bgroup_g3e_base`

Pairings

- "Standard" (single) pairing:

```
void bgroup_pairing(bgroup_g3e *r,
                    const bgroup_g1e *a,
                    const bgroup_g2e *b);
```

- Pairing of `bgroup_g1e_base` and `bgroup_g2e_base` must yield `bgroup_g3e_base`
- Product of multiple pairings

```
void bgroup_pairing_product(bgroup_g3e *r,
                            const bgroup_g1e *a,
                            const bgroup_g2e *b,
                            unsigned long long alen);
```

Pairings

- "Standard" (single) pairing:

```
void bgroup_pairing(bgroup_g3e *r,
                    const bgroup_g1e *a,
                    const bgroup_g2e *b);
```

- Pairing of `bgroup_g1e_base` and `bgroup_g2e_base` must yield `bgroup_g3e_base`
- Product of multiple pairings

```
void bgroup_pairing_product(bgroup_g3e *r,
                            const bgroup_g1e *a,
                            const bgroup_g2e *b,
                            unsigned long long alen);
```

- Currently no non-constant-time version of pairings
- Very easy to add but no speedups expected

Reference implementation setting

- Use 254-bit Barreto-Naehrig curve from Aranha, Karabina, Longa, Gebotys, López, 2011
- We call this setting `bn62550`
- Reference C implementation and AMD64 implementation

Reference implementation setting

- Use 254-bit Barreto-Naehrig curve from Aranha, Karabina, Longa, Gebotys, López, 2011
- We call this setting `bn62550`
- Reference C implementation and AMD64 implementation
- Pairing implementation follows previous implementations

Addition and doubling

- Points are represented in Jacobian coordinates
- Any point with $Z = 0$ is considered the point at infinity

Addition and doubling

- Points are represented in Jacobian coordinates
- Any point with $Z = 0$ is considered the point at infinity
- Doubling uses $2\text{M}+5\text{S}$ formulas
- Also works for point at infinity

Addition and doubling

- Points are represented in Jacobian coordinates
- Any point with $Z = 0$ is considered the point at infinity
- Doubling uses $2\text{M}+5\text{S}$ formulas
- Also works for point at infinity
- Addition needs to work for all inputs
- Perform addition and doubling and use constant-time selection of results

Addition and doubling

- Points are represented in Jacobian coordinates
- Any point with $Z = 0$ is considered the point at infinity
- Doubling uses $2\text{M}+5\text{S}$ formulas
- Also works for point at infinity
- Addition needs to work for all inputs
- Perform addition and doubling and use constant-time selection of results
- "Unified" formulas by Hışıl work for addition and doubling, but have other special cases
- Specifically: cannot add $(x_1, y_1) + (x_2, -y_1)$ for $x_1 \neq x_2$

Scalar multiplication

"Standard" case

- Use 2-dimensional GLV decomposition in G_1
- Use 4-dimensional GLS/GLV decomposition in G_2 and G_3
- Constant-time: joint signed fixed window ($w = 5$ for G_1, $w = 4$ for G_2, G_3)
- Non-constant-time: signed sliding window

Scalar multiplication

"Standard" case

- Use 2-dimensional GLV decomposition in G_1
- Use 4-dimensional GLS/GLV decomposition in G_2 and G_3
- Constant-time: joint signed fixed window ($w = 5$ for G_1, $w = 4$ for G_2, G_3)
- Non-constant-time: signed sliding window

Fixed base-point scalar multiplication

- Offline precomputation of 512 multiples of the base point
- 63 additions (no doublings) to compute scalar multiplication

Multi-scalar multiplication

- Again use GLV or GLS/GLV decomposition
- Constant-time: scalarmult and add up the results
- Variable-time:
  - Up to 16 inputs (G_2, G_3: 8 inputs): joint sliding window
  - Beyond that: Use Bos-Coster algorithm

Bos-Coster algorithm

- Computation of $Q = \sum_1^n s_i P_i$

Bos-Coster algorithm

- Computation of $Q = \sum_1^n s_i P_i$
- Idea: Assume $s_1 > s_2 > \cdots > s_n$. Recursively compute $Q = (s_1 - s_2)P_1 + s_2(P_1 + P_2) + s_3 P_3 \cdots + s_n P_n$
- Each step requires the two largest scalars, one scalar subtraction and one point addition
- Intuition: each step "eliminates" expected $\log n$ scalar bits

Bos-Coster algorithm

- Computation of $Q = \sum_1^n s_i P_i$
- Idea: Assume $s_1 > s_2 > \cdots > s_n$. Recursively compute $Q = (s_1 - s_2)P_1 + s_2(P_1 + P_2) + s_3 P_3 \cdots + s_n P_n$
- Each step requires the two largest scalars, one scalar subtraction and one point addition
- Intuition: each step "eliminates" expected $\log n$ scalar bits
- Requires fast access to the two largest scalars: put scalars into a heap

Selected performance results

- Benchmarks carried out on Intel Core i5-3210M "Ivy Bridge"
- Arithmetic in G_1:
  - `bgroup_g1e_scalarmult`: 347024 cycles
  - `bgroup_g1e_scalarmult_base`: 128596 cycles
  - `bgroup_g1e_scalarmult_publicinputs`: 288240 cycles
  - `bgroup_g1e_multiscalarmult` ($n = 128$): 45186732 cycles
  - `bgroup_g1e_multiscalarmult_publicinputs`: 14605364 cycles

Selected performance results

- Benchmarks carried out on Intel Core i5-3210M "Ivy Bridge"
- Arithmetic in G_1:
  - `bgroup_g1e_scalarmult`: 347024 cycles
  - `bgroup_g1e_scalarmult_base`: 128596 cycles
  - `bgroup_g1e_scalarmult_publicinputs`: 288240 cycles
  - `bgroup_g1e_multiscalarmult` ($n = 128$): 45186732 cycles
  - `bgroup_g1e_multiscalarmult_publicinputs`: 14605364 cycles
- Pairing performance:
  - Single pairing: 2567116 cycles
  - Pairing product ($n = 2$): 3832644 cycles
  - Pairing product ($n = 8$): 11381384 cycles

Selected performance results

- Benchmarks carried out on Intel Core i5-3210M "Ivy Bridge"
- Arithmetic in G_1:
  - `bgroup_g1e_scalarmult`: 347024 cycles
  - `bgroup_g1e_scalarmult_base`: 128596 cycles
  - `bgroup_g1e_scalarmult_publicinputs`: 288240 cycles
  - `bgroup_g1e_multiscalarmult` ($n = 128$): 45186732 cycles
  - `bgroup_g1e_multiscalarmult_publicinputs`: 14605364 cycles
- Pairing performance:
  - Single pairing: 2567116 cycles
  - Pairing product ($n = 2$): 3832644 cycles
  - Pairing product ($n = 8$): 11381384 cycles
- Many more benchmark results in the paper

BLS signatures

Key generation

- Let Q be the "base point" of G_2
- Choose random scalar s, compute $R = sQ$.
- Public key R, secret key s

Signing

- Hash message M to element P_M in G_1
- Compute signature $\sigma = sP_M$

Verification

- Accept signature σ on message M if $e(-\sigma, Q) \cdot e(P_M, R) = 1$

BLS key generation

```
int crypto_sign_keypair(
  unsigned char *pk,
  unsigned char *sk)
{
  /* private key */
  bgroup_scalar x;
  bgroup_scalar_setrandom(&x);
  bgroup_scalar_pack(sk, &x);

  /* public key */
  bgroup_g2e r;
  bgroup_g2_scalarmult_base(&r, &x);
  bgroup_g2_pack(pk, &r);

  return 0;
}
```

BLS signing

```
int crypto_sign(unsigned char *sm, unsigned long long *smlen,
  const unsigned char *m, unsigned long long mlen,
  const unsigned char *sk)
{
  bgroup_g1e p, p1;
  bgroup_scalar x;
  int i,r;

  bgroup_g1e_hashfromstr_publicinputs(&p, m, mlen);
  r = bgroup_scalar_unpack(&x, sk);
  bgroup_g1e_scalarmult(&p1, &p, &x);
  bgroup_g1e_pack(sm, &p1);

  for (i = 0; i < mlen; i++)
  sm[i + CRYPTO_BYTES] = m[i];
  *smlen = mlen + CRYPTO_BYTES;

  return -r;
}
```

BLS signature verification

```
int crypto_sign_open( unsigned char *m, unsigned long long *mlen,
  const unsigned char *sm, unsigned long long smlen,
  const unsigned char *pk)
{
  bgroup_g1e p[2];
  bgroup_g2e q[2];
  bgroup_g3e r;
  unsigned long long i; int ok;

  ok  = !bgroup_g1e_unpack(p, sm);
  bgroup_g1e_negate_publicinputs(p, p);
  q[0] = bgroup_g2e_base;
  bgroup_g1e_hashfromstr_publicinputs(p+1, sm + CRYPTO_BYTES,
                                           smlen - CRYPTO_BYTES);
  ok &= !bgroup_g2e_unpack(q+1, pk);
  bgroup_pairing_product(&r, p, q, 2);
  ok &= bgroup_g3e_equals(&r, &bgroup_g3e_neutral);

  ...
}
```

Paper and Software

Paper

http://cryptojedi.org/papers/#panda

Software

http://panda.cryptojedi.org/
(available soon)