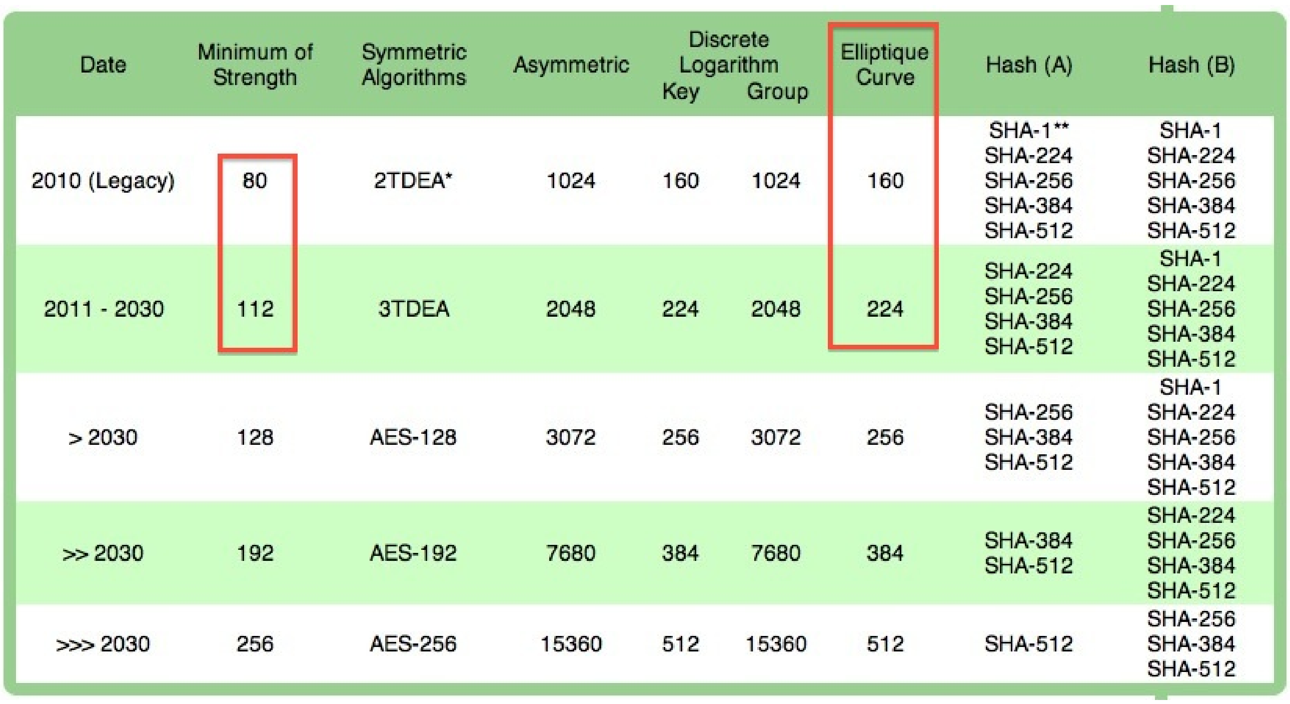
1 Charm-crypto Benchmark

Time unit is ms. Run 1000 trials and the average be recorded. I'm running the code on Ubuntu 12.04, which is a virtual machine running in VMWare fusion on my MACbook Air with 1.8 GHz Intel i5 and 4 GB memory. The virtual machine has access to one core of CPU and maximum 1 GB of memory. This is a Charm benchmark, no pre-processing, no Mul optimaztion. see table 1.

Table 1: Charm benchmark

Curves	element length					Average running-time in ms											
	G1	G2	GT	Dlog	NIST	G1 Mul	G1 Exp	G2 Mul	G2 Exp	GT Mul	GT Exp	Pairing	ZR Exp	R(G1)	R(G2)	G1 32bitE	G2 32bitE
SS512	512	512	1024	1024	80	0.0204	3.7503	0.0201	3.7833	0.0055	0.4844	3.9723	0.0269	4.2506	4.2333	0.7927	0.7854
MNT159	159	477	954	954	80	0.0077	1.1371	0.0523	10.5604	0.0182	2.6907	8.6728	0.0278	1.6193	10.6043	0.2443	2.1347
MNT224	224	672	1344	1344	112	0.0095	2.1293	0.0650	17.9085	0.0216	4.8034	15.7244	0.0523	2.5956	18.2118	0.3228	2.6971
BN	160	320	1920	1920	80	0.0078	1.1357	0.0136	2.3709	0.0623	11.0031	46.8283	0.0266	1.6140	2.8547	0.2482	0.5071

Fig 1 is a table of NIST recommendation, I found if from: http://www.keylength.com/en/4/



Date	Minimum of Strength	Symmetric Algorithms	Asymmetric	Discrete Logarithm Key	Discrete Logarithm Group	Elliptique Curve	Hash (A)	Hash (B)
2010 (Legacy)	80	2TDEA*	1024	160	1024	160	SHA-1** SHA-224 SHA-256 SHA-384 SHA-512	SHA-1 SHA-224 SHA-256 SHA-384 SHA-512
2011 - 2030	112	3TDEA	2048	224	2048	224	SHA-224 SHA-256 SHA-384 SHA-512	SHA-1 SHA-224 SHA-256 SHA-384 SHA-512
> 2030	128	AES-128	3072	256	3072	256	SHA-256 SHA-384 SHA-512	SHA-1 SHA-224 SHA-256 SHA-384 SHA-512
>> 2030	192	AES-192	7680	384	7680	384	SHA-384 SHA-512	SHA-224 SHA-256 SHA-384 SHA-512
>>> 2030	256	AES-256	15360	512	15360	512	SHA-512	SHA-256 SHA-384 SHA-512

Figure 1: NIST recommendation

Here are some explanation:

1. Dlog means Dlog Security bits, NIST means NIST symmetric security bits. R(G1) means generate a random element in G1. G1 32bitE means that Exp in G1 but the power is a 32 bit int.
2. Charm and PBC group name match: 'SS512':a, 'SS1024':a1, 'MNT159':d159, 'MNT201':d201, 'MNT224':d224.
3. SS512 group, the order is 160 bits and the base field is 512 bits long.
4. MNT curve, the base field size is n, n=159, 224. Dlog security is 6n.
5. The order of BN curve is 12, the element size in GT is larger and pairing is slower. Here is a quote from PBC lib:"Type F should be used when the top priority is to minimize bandwidth (e.g. short signatures). The current implementation makes them slow. If finite field discrete log algorithms improve further, type D pairings will have to use larger fields, but type F can still remain short, up to a point."
6. In PBC library, the MNT curve, because of a certain trick, elements of group G2 need only be 3 times longer, rather than 6 times long. Since Charm-crypto is based on PBC library, the elements in group G2 is also 3 times longer than G1 element.

2 Encryption scheme comparison: BB04ibe, Waters 05, DSE09 (Waters 09) and CLLWW12

2.1 Here are some basic facts about the the methodology:

1. All code are written in Python and based on Charm crypto lib.
2. Time unit is ms. Run 200 trials and the average be recorded.
3. The implementation was based on Charm-crypto. Notice that there is no pre-processing. Also, there is no optimization of Mul operation. Table 1 lists the running-time of each operation.
4. We count # of Exp and Pairing. For Mul, Div, Add and Sub, they are too small and we omit them.

2.2 Table 2 is about Identity-based Encryption schemes

Here are some explanations.

1. the first number in column setup() (and Keygen(), Enc(), Dec()) is the real-time running result. The number in the bracket is the estimation based on data in table 1
2. In the setup(), to generate a random generator, for example $g_1 = group.random(G_1)$, it actually takes a long time to generate such a random element. See table 1 for more info. $R(G_1)$ in setup() means you need to random an element in G_1.
3. In "# of Exp, Pairing", G_1(G_2, GT and ZR) means G_1(G_2, GT and ZR) Exponential operation. PP means Pairing operation.
4. For the size of public parameters, msk, sk and ct, G_1 means the size of an element in G_1. So does the G_2, GT and ZR.

2.3 Information about the IBE schemes

1. BBibe04:D. Boneh, X. Boyen. "Efficient Selective Identity-Based Encryption Without Random Oracles", Section 5.1.
   Implemented by Charm team. Type: Encryption.
   Notice: the size of sk should be $1ZR + 1G_2$. The implementation store user's ID as one of the element of the sk. In real life application, we can always use database to record the mapping between ID and the secret key.
2. N04(Waters 05): Brent Waters. Efficient identity-based encryption without random oracles. EUROCRYPT 05.
   However, the scheme that implemented is "David Naccache Secure and Practical Identity-Based Encryption Section 4", which is an improved version of Water 05. Original implementation: Charm team.
3. N05(Waters 05) improved: Improved by: Fan Zhang. Here are the improvements:
   (a) $e(g_1, g_2)$ is pre-calculated as part of public parameters.
   (b) Previous implementation was trying to multiply an element in G_1 with an element in ZR (which is the line: $d1 = mk[`U'][i] * *v[i]$), which sometimes cause the compiler to throw an error. It is a type error. And elements in U should be a group element instead of in ZR. I fixed the problem by having U_z as a vector in ZR and $U = g^{U\_z}$ as U.
   (c) I stored U_z and u as part of msk. This will speed up the extract() a lot. The trick is that, instead of doing exponential operation and then multiply all together, I compute the exponent first and then do one exponential operation
   (d) I have two copies of U and u' now. The reason is that we want the scheme to work perfectly under asymmetric groups and make sk in G_2 and ct in G_1
   (e) sk are in G_2 and ct are in G_1 now. Before that, we have 1 element in G_1 and the other in G_2 in both sk and ct.
4. Waters 09 improved: Dual System Encryption: Realizing Fully Secure IBE and HIBE under Simple Assumptions. CRYPTO 2009. Original implementation: Charm team. Improved by: Fan Zhang. Here are the improvements:
   (a) It works under MNT curve now. However, the size of pk and msk are larger since I need have some duplicate elements in G_2.
   (b) u, w, and h has two copies now. One in G_1, the other one in G_2. They all stored as public params
   (c) pre-calculated $g_2^{-\alpha}$, g_2^b and stored in msk. This makes the keygen() faster.
   (d) The size of public param and msk should be minimal now.

   Since there is no huge improvement in terms of performance, We didn't compare the improved version with the original version

5. CLLWW12: J. Chen, H. Lim, S. Ling, H. Wang, H. Wee Shorter IBE and Signatures via Asymmetric Pairings", Section 4. Published in: Pairing 2012. Implementation: Fan Zhang.

6. CLLWW12 improved: Instead of store $MK = \{\alpha, g_2^{d_1^*}, g_2^{d_2^*}\}$ as the master secret key, I store $MK = \{\alpha, d_1^*, d_2^*\}$. And the computation of SK_{ID}, I first compute $(\alpha + rID)d_1^* - rd_2^*$ first and then apply the exponential operation. This reduce the G_2 pairing from 8 to 4. This is the similar trick I played in N04(Waters05) improved version.

Table 2: Identity-based Encryption

Scheme	# of Exp, Pairing				Group	Average running-time in ms			
BBibe04 ibenc_bb03.py	Setup()	Keygen()	Enc()	Dec()		setup()	Keygen()	Enc()	Dec()
	$1R(G_1)+1R(G_2)+2G_1+1PP$	$1G_2$	$3G_1+1GT$	$1G_1+1PP$	SS512	20.87 (19.96)	4.20 (3.78)	12.18 (11.74)	7.67 (7.72)
	public param	master secret key	secret key	ciphertext	MNT159	23.56 (23.17)	10.35 (10.56)	6.43 (6.1)	9.61 (9.81)
	$3G_1+1GT$	$2ZR+1G_2$	$2ZR+1G_2$	$2G_1+1GT$	MNT224	41.38 (40.79)	17.98 (17.91)	11.63 (11.19)	17.88 (17.85)
					BN	58.00 (53.57)	3.03 (2.27)	15.59 (14.41)	51.31 (47.96)
N04(Waters05) ibenc_waters05.py	Setup()	Keygen()	Enc()	Dec()		setup()	Keygen()	Enc()	Dec()
	$1R(G_1)+2R(G_2)+1G_1+1G_2$	$5ZR+1G_1+2G_2$	$5ZR+1G_1+1G_2+1GT$	$2PP$	SS512	23.44 (20.25)	18.43 (11.45) [1]	22.91 (8.15)	7.86 (7.94))
	public param	master secret key	secret key	ciphertext	MNT159	36.24 (34.53)	39.70 (22.4)	50.84 (14.53)	17.09 (17.35)
	$5ZR+2G_1+2G_2+1GT$	$1G_2$	$1G_1+1G_2$	$1G_1+1G_2+1GT$	MNT224	60.45 (59.06)	66.43 (38.21)	86.76 (25.1)	31.00 (31.45)
					BN	14.13 (10.73)	10.74 (5.81)	69.84 (14.54)	95.74 (93.66)
N04(Waters05) improved ibenc_waters05 _improved.py	Setup()	Keygen()	Enc()	Dec()		setup()	Keygen()	Enc()	Dec()
	$1R(G_1)+1R(G_2)+7G_1+1G_2+1PP$	$2G_2$	$5G_1^{32bitnumber}+2G_1+1GT$	$2PP$	SS512	45.77 (42.49)	8.03 (7.57)	10.63 (11.95)	7.98 (7.94)
	public param	master secret key	secret key	ciphertext	MNT159	41.31 (39.42)	20.14 (21.12)	6.00 (6.19)	17.01 (17.35)
	$8G_1+1G_2+1GT$	$6ZR+1G_2$	$2G_2$	$2G_1+1GT$	MNT224	71.15 (69.35)	34.98 (35.82)	10.26 (10.68)	31.11 (31.45)
					BN	63.67 (61.52)	5.18 (4.54)	13.99 (14.52)	91.91 (93.66)
Waters 09 improved ibenc_waters09 _improved.py	Setup()	Keygen()	Enc()	Dec()		setup()	Keygen()	Enc()	Dec()
	$1R(G_1)+1R(G_2)+15G_1+9G_2+1GT+1PP$	$12G_2$	$14G_1+1GT$	$1GT+9PP$	SS512	106.68 (103.24)	47.50 (45.4)	54.33 (52.99)	38.56 (36.24)
	public param	master secret key	secret key	ciphertext	MNT159	134.05 (135.69)	121.77 (126.72)	20.28 (18.61)	80.25 (80.75)
	$13G_1+4G_2+1GT$	$6G_2$	$1ZR+8G_2$	$1ZR+9G_1+1GT$	MNT224	232.41 (234.45)	210.28 (214.9)	35.79 (34.61)	145.14 (146.32)
					BN	103.32 (99.77)	30.41 (27.25)	27.96 (26.9)	426.99 (432.46)
CLLWW12 ibenc_cllww12.py	Setup()	Keygen()	Enc()	Dec()		setup()	Keygen()	Enc()	Dec()
	$1R(G_1)+1R(G_2)+8G_1+8G_2+1GT+1PP$	$8G_2$	$8G_1+1GT$	$4PP$	SS512 [2]	81.81 (73.21)	30.32 (30.27)	31.00 (30.49)	15.73 (15.89)

[1] There are inconsistency in keygen() and Enc() in all the curves. Look at the following line of code: $d_1* = (pk['U'][i])^{v[i]}$, d_1 is an element in G1 and pk is in ZR, the Exp is in ZR. After the Exp operation, an element in ZR should be somehow cast to an element in G_2 (I guess what they did is g_2^{ZR}). This is actually an type error and should not be a right implementation. I fixed it in the improved version. This 'cast' takes time. This is the reason for the inconsistency. Here are some test result of cast: ZR to G_1 in SS512/MNT159/MNT224: 2.3340/0.6903/1.2846; ZR to G_2: 2.2580/6.1770/10.7333. If we take the time of cast into consideration. It is consistent.

[2] The CLLWW12 scheme only secure under asymmetric groups. It indeed works under SS512, but not secure.

	public param	master secret key	secret key	ciphertext	MNT159	120.91 (117.17)	79.99 (84.48)	12.09 (11.79)	34.33 (34.69)
	$8G_1 + 1GT$	$1ZR + 8G_2$	$1ZR + 4G_2$	$4G_1 + 1GT$	MNT224	205.17 (201.64)	139.45 (143.27)	21.92 (21.84)	62.30 (62.9)
					BN	97.96 (89.55)	19.34 (18.17)	20.07 (20.09)	186.05 (187.31)
CLLWW12 _improved ibenc_cllww12 _improved.py	Setup()	Keygen()	Enc()	Dec()		setup()	Keygen()	Enc()	Dec()
	$1R(G_1) + 1R(G_2) + 8G_1 + 1GT + 1PP$	$4G_2$	$8G_1 + 1GT$	$4PP$	SS512	51.75 (42.94)[3]	15.39 (15.13)	30.76 (30.49)	15.74 (15.89)
	public param	master secret key	secret key	ciphertext	MNT159	41.05 (32.68)	39.89 (42.24)	12.09 (11.79)	34.31 (34.69)
	$8G_1 + 1GT$	$9ZR + 1G_2$	$1ZR + 4G_2$	$4G_1 + 1GT$	MNT224	66.75 (58.37)	70.01 (71.63)	21.90 (21.84)	62.31 (62.9)
					BN	80.01 (71.39)	10.04 (9.08)	20.25 (20.09)	186.43 (187.31)

3 Signature scheme comparison: BLS, Waters 05, DSE09 (Waters 09) and CLLWW12

3.1 Table 3 is about Identity-based Signature schemes.

Time unit is ms. Run 200 trials and the average be recorded.
The implementation was based on Charm-crypto.

3.2 Information about the signature schemes

1. BLS: Implemented by Charm team. The public parameter actually has 4 parts. However, the ‘identity’ should has the same length as g^x and the ‘secparam’ can be stored some where else. Actually ’secparam’ is very short and we can ignore it.

2. N04(Waters 05): The same paper as the one in encryption scheme. Original implementation: Charm team.

3. N04(Waters 05) improved: Implemented by Fan Zhang. Here are the improvements:

   (a) The same trick in ibenc_waters05_improved has been used here too.

   (b) Also, I swapped g_1 and g_2 to make the signature happens in G_1. It’s much more faster now.

4. Waters 09 improved: The same paper as the one in encryption scheme. Original implementation: Charm team. Improved by: Fan Zhang. Here are the improvements: delete the alpha from msk and add $g_2^{-\alpha}$ into it. Since there is no huge improvement in terms of performance
   Note: The original implementation by Charm team support the asymmetric groups. What I did is just some trivial modification. And We can swap G_1 and G_2 to achieve better performance too. However, I didn’t swap them here in Waters09 scheme.

5. CLLWW12: The same paper as the one in encryption scheme. J. Chen, H. Lim, S. Ling, H. Wang, H. Wee Shorter IBE and Signatures via Asymmetric Pairings”, Section 5. Published in: Pairing 2012. Implementation: Fan Zhang.

6. CLLWW12 swap: Implementation: Fan Zhang. Simply by swap g_1 with g_2, and in the pair(), swap the first and the second param. Its done! Notice: now, even if we call it g_1, its now an element in G_2, so does the g_2. And the signature is much more faster after the swap.

7. CLLWW12 swap improved: Implementation: Fan Zhang. This is a improved version of pksig_cllww12_swap. The trick in ibenc_cllww12_improved has been used here. One has to notice that we already swapped g_1 with g_2 This improved version is 2 times faster than it’s predecessor.

Table 3: Identity-based Signature

[3]The generation of DPVS in CLLWW12 scheme takes certain amount of time.

Scheme	# of Exp, Pairing			Group	Average running-time in ms		
	Keygen()	Sign()	Verify()		Keygen()	Sign()	Verify()
BLS pksig_bls04.py	$1R(G_2) + 1G_2$	$1G_1$	$2PP$	SS512	8.55 (8.02)	12.29 (3.75) [4]	16.44 (7.94)
	public param	secret key	Signature	MNT159	21.02 (21.16)	1.28 (1.14)	17.26 (17.35)
	$3G_2$	$1ZR$	$1G_1$	MNT224	36.15 (36.12)	2.44 (2.13)	31.75 (31.45)
				BN	6.24 (5.13)	1.33 (1.14)	98.88 (93.66)
	Keygen()	Sign()	Verify()		Keygen()	Sign()	Verify()
N04(Waters05) pksig_water05.py	$1R(G_1) + 2R(G_2) + 1G_1 + 1G_2 + 2GT + 2PP$	$5ZR + 1G_1 + 1G_2$	$5ZR + 2PP$	SS512	31.96 (29.16)	18.73 (7.67)	18.68 (8.08))
	public param	secret key	Signature	MNT159	58.35 (57.25)	38.45 (11.84) [5]	43.90 (17.48)
	$5ZR + 2G_1 + 2G_2 + 1GT$	$1G_1$	$1G_1 + 1G_2$	MNT224	101.88 (100.11)	69.36 (20.3)	80.46 (31.71)
				BN	133.52 (126.39)	11.09 (3.54)	103.22 (93.79)
	Keygen()	Sign()	Verify()		Keygen()	Sign()	Verify()
N04(Waters05) improved(swap) pksig_water05 _improved.py	$1R(G_1) + 1R(G_2) + 1G_1 + 7G_2 + 1PP$	$2G_1$	$5G_2^{32bitnumber} + 2PP$	SS512	45.46 (42.69)	7.94 (7.5)	10.06 (11.87)
	public param	secret key	Signature	MNT159	94.22 (95.96)	2.80 (2.27)	22.90 (28.02)
	$1G_1 + 8G_2 + 1GT$	$1G_1$	$2G_1$	MNT224	164.33 (164.02)	4.71 (4.26)	38.97 (44.93)
				BN	73.02 (68.33)	2.77 (2.27)	94.76 (96.19)
	Keygen()	Sign()	Verify()		Keygen()	Sign()	Verify()
Waters 09 improved pksig_waters09 _improved.py	$1R(G_1) + 1R(G_2) + 15G_1 + 9G_2 + 1GT + 1PP$	$12G_2$	$14G_1 + 2GT + 9PP$	SS512	105.66 (103.24)	46.59 (45.4)	89.86 (89.22)
	public param	secret key	Signature	MNT159	134.24 (135.69)	121.54 (126.72)	100.35 (99.36)
	$13G_1 + 4G_2 + 1GT$	$6ZR + 1G_1$	$1ZR + 8G_2$	MNT224	233.10 (234.45)	211.28 (214.9)	181.30 (180.94)
				BN	105.16 (99.77)	30.96 (27.25)	469.56 (459.36)
	Keygen()	Sign()	Verify()		Keygen()	Sign()	Verify()
CLLWW12 pksig_cllww12.py	$1R(G_1) + 1R(G_2) + 8G_1 + 8G_2 + 1GT + 1PP$	$8G_2$	$4G_1 + 4PP$	SS512	81.12 (73.21) [6]	30.41 (30.27)	30.76 (30.89)
	public param	secret key	Signature	MNT159	120.64 (117.17)	82.99 (84.48)	38.76 (39.24)
	$8G_1 + 1GT$	$1ZR + 8G_2$	$4G_2$	MNT224	206.60 (201.64)	140.68 (143.27)	71.24 (71.41)
				BN	104.68 (89.55)	19.42 (18.17)	195.03 (191.86)

[4] The inconsistency in Sign() and Verify() are both caused by the following line of code: $group.hash(M, G_1)$. One should map the message into G_1 and then raise to the power of x. $group.hash(M, G_1)$ takes roughly 9 ms. Also, one has to notice that in MNT159/224, this group.hash also happens. However, the mapping process only takes roughly 0.08/0.25 ms in it respectively.

[5] Inconsistency in Sign() and Verify() in all curves caused by the same reason explained in Table **??**

[6] The generation of DPVS in CLLWW12 scheme takes certain amount of time.

	Keygen()	Sign()	Verify()		Keygen()	Sign()	Verify()
CLLWW12 swap pksig_cllww12 _swap.py	$1R(G_1) + 1R(G_2) + 8G_1{+}8G_2{+}1GT{+}1PP$	$8G_1$	$4G_2 + 4PP$	SS512	85.13 (73.21)	30.25 (30)	30.90 (31.02)
	public param	secret key	Signature	MNT159	120.31 (117.17)	9.47 (9.1)	74.16 (76.93)
	$8G_2 + 1GT$	$1ZR + 8G_1$	$4G_1$	MNT224	206.07 (201.64)	17.20 (17.03)	134.74 (134.53)
				BN	101.79 (89.55)	9.68 (9.09)	201.93 (196.4)

	Keygen()	Sign()	Verify()		Keygen()	Sign()	Verify()
CLLWW12 swap improved pksig_cllww12 _swap_improved.py	$1R(G_1) + 1R(G_2) + 8G_2 + 1GT + 1PP$	$4G_1$	$4G_2 + 4PP$	SS512	51.34 (43.21)	15.22 (15)	30.60 (31.02)
	public param	secret key	Signature	MNT159	111.66 (108.07)	5.05 (4.55)	74.55 (76.93)
	$8G_2 + 1GT$	$9ZR + 1G_1$	$4G_1$	MNT224	190.05 (184.6)	8.90 (8.52)	132.45 (134.53)
				BN	91.55 (80.47)	5.12 (4.54)	199.92 (196.4)

4 Exponential, Multiplication and Pairing in pre-processing, PBC library

The observation was: in Charm, Exp in G2 takes longer than pairing, which is unusual. In usual case, Pairing should be 30 times slower than Exp. Due to implementation issues, in practice, it should be 10 times slower.
The reason: There is no optimization in Charm, both in Exp and pairing.
How about the pre-processing optimization? Charm is based on PBC library and PBC library does provide a pre-processing mode.
Table 4 is the result of pre-processing: Now, the Exp is much more faster! However, one may notice that the pre-processing itself takes

Table 4: Pre-processing

average, ms	MNT224	MNT159	SS512
G1 Exp	2.2324	1.1787	4.2456
After pre-processing	0.2946	0.15996	0.58522
Pre-processing itself	10.3266	5.4665	20.3675
G2 Exp	17.7188	10.06818	4.0805
After pre-processing	2.6019	1.44612	0.5485
Pre-processing itself	84.0911	46.7426	19.0134
Pairing	16.0519	8.4755	4.32726.
After pre-processing	12.8999	6.92033	1.8323
Pre-processing itself	3.2822	1.68439	3.86498

a long time, especially in Exp. To understand what Pre-processing truly means, I looked into the pre-processing code of Exp operation. It turns out that the pre-processing is a process of build k-bit base table for n-bit exponentiation. And later, the Exp operation will do a look up first, and then do a normal Mul operation. My understanding is that, the pre-processing of Exp is taken a n-bit number x and build x^2, x^4, x^8... ... and all of them will be stored for further lookup.

It seems that the so called “pre-processing” is not very effective when we take the “pre-processing” itself into consideration. And now, we know that even the Exp is much more faster, its a result of pre-processing and the pro-precessing give no answer to the question: “Why TinyPBC is faster in Mul than PBC library”.

My guess now is that, the reason that Exp takes a long time in PBC library when comparing with RELIC-tinyPBC is not caused by exp itself. It should caused by the implementation of MUL operation.

In RELIC-tinyPBC, 80-bit security, the MUL takes 11727us, and the pairing takes 14,000,000us, which is a ratio of 1200:1 In PBC, When I was using MNT 159, which is rough 70 bits of security. Pairing : G1 MUL = 1790:1. Pairing : G2 MUL = 216:1. The ratio between Pairing and G2 MUL indicates that the MUL in PBC library is roughly 6 times slower than TinyPBC. Table 5 is a table about ratio.

Table 5: Pairing Mul ratio

ratio	Tiny PBC	MNT224	MNT159	SS512
Pairing:MUL	1200:1	-	-	-
Pairing:MUL(G1)	-	1650:1	1790:1	160:1
Pairing:MUL(G2)	-	220:1	216:1	250:1

I went through TinyPBC paper and they use LpezDahab algorithm. I found the paper of the algorithm: High-speed software multiplication in$GF(2^m)$

According to the paper: the proposed method is about 2-5 times faster than standard multiplication. I think this explains why PBCs MUL is 6 times slower than TinyPBC because TinyPBC use optimization.

5 Improved schemes

5.1 Waters 09_improved, ibe scheme

Setup(): The authority first chooses group $\mathbb{G}_1$ and $\mathbb{G}_2$ of prime order p. Next, it chooses generators $g_1 \in \mathbb{G}_1$ and $g_2 \in \mathbb{G}_2$ respectively. Then it chooses v_z, v_{1z}, v_{2z}, w_z, u_z, $h_z \in \mathbb{Z}_p$ and exponents a_1, a_2, b, $\alpha \in \mathbb{Z}_p$. Calculate $v_{G1} = g_1^{v_z}$, $v_{1G1} = g_1^{v_{1z}}$, $v_{2G1} = g_1^{v_{2z}}$, $w_{G1} = g_1^{w_z}$, $u_{G1} = g_1^{u_z}$, $h_{G1} = g_1^{h_z}$. Also, calculate $v_{G2} = g_2^{v_z}$, $v_{1G2} = g_2^{v_{1z}}$, $v_{2G2} = g_2^{v_{2z}}$, $w_{G2} = g_2^{w_z}$, $u_{G2} = g_2^{u_z}$, $h_{G2} = g_2^{h_z}$.
In the same time, let $\tau_1 = v_{G1}(v_{1G_1})^{a_1}$, $\tau_2 = v_{G1}(v_{2G_1})^{a_2}$. It publishes the public parameters PK as the group description $\mathbb{G}_1, \mathbb{G}_2$ along with:
$g_1, g_2, g_1^b, g_1^{a_1}, g_1^{a_2}, g_1^{b\cdot a_1}, g_1^{b\cdot a_2}, \tau_1, \tau_2, \tau_1^b, \tau_2^b, w_{G_1}, u_{G_1}, h_{G_1}, w_{G_2}, u_{G_2}, h_{G_2}, e(g_1, g_2)^{\alpha\cdot a_1\cdot b}$
And the master secret key MSK consists of : $g_2^{-\alpha}, g_2^{\alpha\cdot a_1}, v_{G_2}, v_{1G_2}, v_{2G_2}, g_2^b$.
Enc($PK, \mathcal{I}, M$): The encryption algorithm chooses random s_1, s_2, t, and $tag_c \in \mathbb{Z}_p$, Let $s = s_1 + s_2$. It then blinds $M \in \mathbb{G}_T$ as $C_0 = M \cdot (e(g_1, g_2)^{\alpha\cdot a_1\cdot b})^{s_2}$ and creates:
$C_1 = (g_1^b)^{s_1+s_2}, C_2 = (g_1^{b\cdot a_1})^{s_1}, C_3 = (g_1^{a_1})^{s_1}, C_4 = (g_1^{b\cdot a_2})^{s_2}, C_5 = (g_1^{a_1})^{s_2}, C_6 = \tau_1^{s_1}\tau_2^{s_2}, C_7 = (\tau_1^b)^{s_1}(\tau_2^b)^{s_2}w_{G_1}^{-t}, E_1 = (u_{G_1}^{\mathcal{I}} w_{G_1}^{tag_c} h_{G_1})^t, E_2 = g_1^t$.
The Ciphertext is $CT = C_0, C_1, ..., C_7, E_1, E_2, tag_c$.
KeyGen($MSK, PP, \mathcal{I}$): The authority chooses random $r_1, r_2, z_1, z_2, tag_k \in \mathbb{Z}_p$. Let $r = r_1 + r_2$. Then it creates:
$D_1 = g_2^{\alpha\cdot a_1}, D_2 = g_2^{-\alpha} v_{1G_2}^r g_2^{z_1}, D_3 = (g_2^b)^{-z_1}, D_4 = v_{2G_2}^r g_2^{z_2}, D_5 = (g_2^b)^{-z_2}, D_6 = g_2^{r_2\cdot b}, D_7 = g_2^{r_1}, K = (u_{G_2}^{\mathcal{I}} w_{G_2}^{tag_k} h_{G_2})^{r_1}$.
The secret key is $SK = D_1, ..., D_7, K, tag_k$.
Dec($CT, K_{\mathcal{I}}$): Nothing has been changed in Dec().

5.2 Waters 09_swap_improved, signature scheme

Swap means to swap g_1 and g_2. It makes the signature much more faster when we are using asymmetric groups. There is no need to change the code except in every pairing function: Pair(param1, param2), you need to swap param1 with param2. The following paragraphs describe the improved scheme before swap.

5.3 N04(Waters 05)_improved, ibe scheme

5.4 N04(Waters 05)_swap_improved, signature scheme