International Journal of Cryptology Research 6(1): 16 - 30 (2016)

An Optimized Pairing-Based Cryptography Library for Android

Syh-Yuan Tan[1], **Chee-Siang Wong**[2], and **Hoon-Herk Ng**[1]

[1]*Faculty of Information Science and Technology, Multimedia University, Malaysia*
[2]*Faculty of Information and Communication Technology, Universiti Tunku Abdul Rahman, Malaysia*

E-mail: sytan@mmu.edu.my, wongcs@utar.edu.my, malco_ng@live.com.my

ABSTRACT

There are numerous pairing-based cryptography (PBC) libraries available for desktop-based applications. However, these libraries are mostly not compatible or not optimized for mobile phone operating systems (OS) such as Android. In this paper, we show the inconsistency on benchmarking result for Java-based PBC libraries between Java Virtual Machine (JVM) and Dalvik Virtual Machine (DVM). Identifying the root cause, we present a new PBC library for Android, namely, mobile-PBC (mPBC) which combines the strengths of several Java PBC libraries and optimized for Android's DVM. The mPBC library outperforms the existing Java-based PBC libraries in DVM, yet as efficient as the fastest PBC library to date in the JVM. In particular, the pre-processed Tate pairing operation in mPBC uses 426.11ms in DVM (Samsung GT-N7000) and 4.50ms in JVM (Sager NP5160).

Keywords: Pairing-Based Cryptography, Elliptic Curve cryptography, Java, Android, Library

1 INTRODUCTION

There has been much interest in recent years in reducing the processing power of cryptographic schemes. Due to the rise of mobile platform and networking ability, a cryptography library which is suitable for mobile phone applications may help in increasing the efficiency of cryptographic computations. Even though there are a lot of cryptography libraries well developed for desktop-based applications, not much were done for mobile devices which have limited battery life and processing power.

Up to date, there are only a few elliptic curve cryptography libraries that support pairing operations written in Java (De Caro and Iovino, 2011, Dong, 2010, Tan et al., 2010) which work in both Dalvik Virtual Machine (DVM) and Java Virtual Machine (JVM). In 2014, Liu et al. (2014) ported the PBC library (Lynn, 2010) from C to Java by using Java Native Interface (JNI) and Android Native Development Kit (NDK). Compared to jPBC by De Caro and Iovino (2011), Liu et al. (2014) is a more complete Java version of PBC. However, Liu et al.'s library runs on Android 4.0.3 or higher and they did not manage to provide performance comparison with jPBC which can only be run on Android 2.2 or lower. This is the main drawback of using C libraries on Android in which they loss the platform independent feature. For instance, even though works by De Caro and Iovino (2011) and Liu et al. (2014) performed significantly faster than the other Java-based pairing libraries, their compilations are tied to the underlying hardware, e.g., the library compiled for ARM architecture cannot run on x86 architecture.

In short, one can view performance and platform independent as the trade-off to each other. On the other hand, we prefer to maintain the platform independent feature of the pairing libraries, so that pairing cryptosystems can run on any device that supports Java.

1.1 Motivation

Our motivation of this work came after the benchmarking on the Java pairing libraries in JVM and DVM. The benchmarks were done in a loop of 1100 times

Platform	JVM1	JVM2	DVM
Device	Sager NP5160	RaspBerry Pi II	Samsung GT-N7000
CPU	Quad-core 2 GHz Intel Core i7-2630M	Quad-core 900Mhz ARM Cortex-A7	Dual-core 1.4 GHz ARM Cortex-A9
RAM	4 GB	1GB	1 GB
OS	Windows 8.1	Raspbian Wheezy	Android 4.2.2
JDK/SDK Version	JDK 1.8.0.25	JDK 1.8.0.25	SDK 23.0.5

Table 1: Test Environment.

for JVM tests with the first 100 times neglected to avoid the caching of processor. The specification of our machines to carry out the target benchmarks are shown in Table 1.

Table 2 shows the timing performance (in nanoseconds) of the libraries under JVM1, JVM2 and DVM on the Type-A curve $y = x^3 + x \bmod p$ where p is a 512-bit prime with 160-bit Solinas prime order q (Lynn, 2010). If an operation is not available under a library, we mark it with the symbol '-'. For example, times of pre-processing multiplication and pre-processing pairing are recorded for jPBC which is the only library provides such features.

We discovered that existing Java PBC libraries perform differently in DVM (a register based virtual machine) as compared to JVM (a stack based virtual machine). For instance, Jpair (Dong, 2010) scores the highest among others (De Caro and Iovino, 2011, Tan et al., 2010) in point scalar multiplication under JVM1 and JVM2 but scores the lowest under DVM. Moreover, THG-PBC is slower than jPBC in JVM1 and JVM2 but faster in DVM despite the fact that jPBC uses k-bit Windows method which is theoretically faster. The results under JVM2 rule out the possibility of processor architecture as both RaspBerry Pi II and Samsung GT-N7000 use ARM-Cortex processors. The remaining causes of the inconsistencies may come from many aspects, ranging from the class structures to the virtual machine architecture.

Table 2: Timing (ns) of Group Operations for Java PBC Libraries

Operations		JVM1			JVM2			DVM		
		THG-PBC	jPBC	Jpair	THG-PBC	jPBC	Jpair	THG-PBC	jPBC	Jpair
Negation		2,183	2,005	5,904	36,352	13,936	33,871	39,430	30,004	65,571
Addition		73,658	63,513	74,477	1,178,562	1,177,218	1,272,717	693,915	671,874	768,134
Doubling		74,737	63,094	75,467	1,250,657	1,255,628	1,368,083	1,107,691	1,081,931	1,200,287
D&A	Affine	15,394,156	-	-	282,096,335	-	-	269,322,363	-	-
	k-Affine	-	13,143,754	-	-	243,576,846	-	-	282,808,693	-
	Jacobian	-	-	5,853,007	-	-	177,507,191	-	-	523,765,047
P.P Mult.		-	1,902,335	-	-	34,925,697	-	-	34,246,110	-
Pairing		24,737,596	9,037,093	9,047,344	542,602,143	291,230,173	289,206,823	1,037,003,196	930,931,258	898,596,388
P.P. Pairing		-	4,426,967	-	-	143,932,848	-	-	466,375,265	-

1.2 Contribution

These inconsistencies indicate that the existing Java libraries (De Caro and Iovino, 2011, Dong, 2010, Tan et al., 2010) are not optimized for Android platform and inspired us to build an optimized PBC library primarily for Android's Dalvik Virtual Machine (DVM) with optimizations on Java Virtual Machine (JVM) come as a by-product. In this paper, we realize an optimized PBC library in Java, namely, mobile-PBC (mPBC), which outperforms the existing libraries (De Caro and Iovino, 2011, Dong, 2010, Tan et al., 2010) in Android's DVM.

1.3 Organization

The rest of the paper are organized as follows. In Section 2, we briefly discuss the algorithms of the point operations for pairing based cryptography. In Section 3, we present the optimization techniques for mPBC in terms of programming approach and cryptography approach. In Section 4 we discuss the benchmark results of mPBC and end the work with a case study in Section 5.

2 PRELIMINARIES

2.1 Point

A point P on elliptic curve $E(\mathbb{F}_p)$ under the finite field $\mathbb{F}_p$ with prime modulus p in the affine coordinate (x, y) can be represented in the format of Jacobian coordinate $(x/z^2, y/z^3, z)$ to get rid of the calculation on multiplicative inverse during point addition and point doubling.

2.2 Point Scalar Multiplication

THG-PBC uses Double and Add (Tan et al., 2010) while jPBC (De Caro and Iovino, 2011) and Jpair (Dong, 2010) use Double and Add with k-bit window as the point scalar multiplication algorithm.

2.2.1 Double and Add Algorithm

THG-PBC (Tan et al., 2010) and jPair (Dong, 2010) use the Double and Add algorithm as shown in Algorithm 1 as the point scalar multiplication. The difference between these two libraries is that the former use the algorithm in Affine coordinate while the latter is in Jacobian coordinate.

Algorithm 1 Double and Add

Require: m, P
Ensure: $Q = mP$
1: $Q \leftarrow P$
2: **for all** $i \leftarrow (\lg(m)) - 2$ to 0 **do**
3: $\quad Q \leftarrow 2Q$
4: $\quad$ **if** i is 1 **then**
5: $\quad\quad Q \leftarrow Q + P$
6: $\quad$ **end if**
7: **end for**
8: **return** Q

2.2.2 Double and Add Algorithm with k-bit Window

jPBC (De Caro and Iovino, 2011) uses an enhanced version of Algorithm 1, namely, Double and Add with k-bit Window. The parameter k defines the lookup table size which contains the precomputed points.

3 OPTIMIZATION TECHNIQUES

In this section, we examine the possible causes of the bizarre benchmarks and present the optimization techniques used in producing the mPBC library.

3.1 Global Methods

At the first glance, the performance of Jpair in JVM is benefited from its class structure which pushes all point operations to the `Curve` class and yields the global methods for `Point` objects. This may indicate that the execution of a global method is faster than a local method in JVM but the result is conversed in DVM. However, our quick experiments showed that this hypothesis is not true. The result showed that global methods work better than local methods in DVM by approximately 10% but no significant difference in JVM. This concludes that global variables help in reducing the execution time, but not as significant as shown in Table 2.

3.2 Point Operations

The optimizations on point operations can be categorized into two main categories, namely, mathematical enhancements and coding optimizations. For the latter, we code the library in such a way that global variables will be utilized whenever it is possible to avoid the creation of local variables; for the former, we make use of bit operation functions supported in `java.math.BigInteger` to replace some simple yet repetitive mathematical operations;

3.2.1 Point Negation

Point negation is a simple point operation that negates the value y under $\mathbb{F}_p$. The implementations of most of the libraries are $-y \bmod p$, but this can be done more efficiently by calculating $p - y$. In the programming aspect, the

point negation function `negate(BigInteger p)` can be optimized by removing the point validations (i.e. check $y < p$) because this function is always called from a valid `Point` object generated by the `Curve` object.

3.2.2 Point Addition and Doubling

In mPBC, we optimize the computation through the use of methods from `BigInteger` libraries to reduce the execution time in DVM. For example, we replace `equals(BigInteger.ZERO)` with `signum()` method which is able to check for zero values with reduced time.

For the algorithms which execute point addition and point doubling frequently, we set the temporary variables as `static` where these variables will be constructed only once. The reuse of temporary variables will save the constructions of new variables at each round of point addition and point doubling. Furthermore, the point addition and point doubling were placed under the `Curve` class as global methods. Additionally, the multiplication and exponentiation operation of `BigInteger` objects are also optimized by using bitwise operation such as `shiftLeft(int n)` whenever possible.

3.2.3 Point Scalar Multiplication

It is well known that point operation in Jacobian coordinate is faster than that of Affine coordinate but inconsistency of benchmark results were identified as well between Double and Add algorithm with k-bit Window and the original Double and Add algorithm, i.e., without k-bit Window in Table 1.1. Under JVM, Jpair's point multiplication is the fastest followed by jPBC's and lastly THG-PBC's; while in DVM, the sequence is totally reversed. It is obvious that the number of temporary variables is directly proportional to the speed of scalar multiplication algorithm in JVM, but inversely proportional to that in DVM.

In view of this, we modified point addition algorithm in Jacobian coordinate in such a way that the first three `BigInteger` variables are taken from

Algorithm 2 Optimized Point Addition in Jacobian Coordinate

Require: P, Q
Ensure: $R = P + Q$
1: $t_1 \leftarrow x_1 z_2^2$
2: $t_2 \leftarrow x_2 z_1^2$
3: $t_3 \leftarrow t_1 - t_2$
4: $t_4 \leftarrow y_1 z_2^3$
5: $t_5 \leftarrow y_2 z_1^3$
6: $t_1 \leftarrow t_1 + t_2$
7: $t_2 \leftarrow t_4 - t_5$
8: $x_3 \leftarrow t_2^2 - t_1 t_3^2$
9: $y_3 \leftarrow ((t_1 t_3^2 - 2x_3)t_2 - (t_4 + t_5)t_3^3)/2$
10: $z_3 \leftarrow z_1 z_2 t_3$
11: **return** $R = (x_3, y_3, z_3)$

the global variables which are shared by all the algorithms in the `Curve` class for calculation purposes. Moreover, by manipulating the algorithm steps, we can reduce four temporary variables from nine to five as shown in Algorithm 2. Therefore, in the implementation, we only need to create two temporary variables instead of nine as in the original algorithm.

3.2.4 Pre-Processing Point Scalar Multiplication

Pre-processing point multiplication which is only found in jPBC (De Caro and Iovino, 2011) uses more memory to construct a pre-processing table, on top of the lookup table of k-bit Window. In mPBC, the pre-processing algorithm used is similar to that of jPBC with optimization done in the programming aspect as discussed. The pre-processing algorithm is described as in Algorithm 3 and 4.

Although the pre-processing algorithm gives great speed increments, it comes with sacrifice in memory to store $n \times 2^k$ points. In the Type-A curve of 80 bits security, the order is 160 bits, the modulus p is 512 bits in length. At $k = 5$, the pre-processing table contains $160/6 \times 2^5 = 832$ points where each point costs 1024 bits and the total memory needed is 832×1024 bits$= 104$KB. If point compression is used, the pre-processing table still cost 52KB.

Algorithm 3 Pre-Processing Table Initialization

Require: $order, k, P$
Ensure: $Table_{PP}$
1: $n \leftarrow \log(order)/(k+1)$
2: $Q \leftarrow P$
3: **for all** $i \leftarrow 0$ to $n-1$ **do**
4: $\quad Table_{PP}[i][0] \leftarrow \mathcal{O}$
5: $\quad$ **for all** $j \leftarrow 1$ to $2^k - 1$ **do**
6: $\quad\quad Table_{PP}[i][j] \leftarrow Q + Table_{PP}[i][j-1]$
7: $\quad$ **end for**
8: $\quad Q \leftarrow Q + Table_{PP}[i][j-1]$
9: **end for**
10: **return** $Table_{PP}$

Algorithm 4 Pre-Processing Point Scalar Multiplication

Require: $m, P, k, Table_{PP}$
Ensure: $Q = mP$
1: $n \leftarrow \log(m)/(k+1)$
2: $Q = \mathcal{O}$
3: **for all** $i \leftarrow 0$ to n **do**
4: $\quad j \leftarrow 0$
5: $\quad$ **for all** $s \leftarrow 0$ to k **do**
6: $\quad\quad j \leftarrow j \mid (m_{ki+s} \times 2^s)$
7: $\quad$ **end for**
8: $\quad$ **if** j > 0 **then**
9: $\quad\quad Q \leftarrow Q + Table_{PP}[i][j]$
10: $\quad$ **end if**
11: **end for**
12: **return** Q

3.3 Bilinear Pairing

The bilinear pairing of mPBC relies on the Type-1 pairing algorithm used by Jpair (Dong, 2010) is the fastest in DVM compared to the other two. Thus, we optimize the pairing using the faster point operation algorithms in Section 3.2.

3.3.1 Pre-Processing Bilinear Pairing

Given that the point addition and point doubling were executed in the pre-processing phase, it does not help in reducing the actual computation time of the final pairing operation. In mPBC, the pre-processing algorithm used is similar to that of jPBC with optimization done in the programming aspect as discussed.

3.4 Class Structure

mPBC consists of seven classes only to keep the library simple, namely, `Curve`, `CpxBigInteger`, `Point`, `JcbPoint`, `PointMulPreProcessing`, `Pairing`, `PairingPreProcessing`. Among all, only `JcbPoint` inherits `Point` and there is no inheritance relationship among other classes.

We reassert that all optimizations are targeted on the performance in DVM, while the improvements in JVM come as byproduct.

4 BENCHMARKS

In this section, we show the benchmarks for mPBC in Table 3 with the same environment as in Section 1.1 where 'J' represents Jacobian and 'A' represents Affine.

Table 3: Timing (ns) of Group Operations for mPBC.

Operations		mPBC		
		JVM1	JVM2	DVM
Negation		899	13,870	17,811
J-Addition		12,475	479,858	603,285
J-Doubling		12,665	487,670	962,549
A-Addition		64,353	1,215,114	523,073
A-Doubling		63,444	1,248,487	592,711
D&A	Affine	15,591,693	283,056,439	310,566,204
	k-Affine	13,023,479	242,890,806	268,979,259
	Jacobian	5,199,897	169,289,050	443,377,476
	k-Jacobian	4,014,078	130,038,010	372,981,140
P.Proc. Mult.		1,937,780	34,740,619	26,003,816
Pairing		8,429,780	299,451,947	809,407,243
P.Proc. Pairing		4,501,117	162,697,997	426,115,736

4.1 Discussion

From Table 3, we can see that the more variables needed, the slower the algorithms execute in DVM. This explain why Jacobian coordinates system performs well in JVM but affine coordinates system performs well in DVM. Unlike JVM, Android's DVM uses garbage collector to dispose the unused variables (And) and the disposal cost is greater than speed-up gained from the gap of algorithm complexity between Jacobian coordinates and affine coordinates.

Secondly, simple class structure delays lesser than complex class structure in DVM. Although it is confirmed from Table 3 that Double and Add algorithm is faster with the presence of k-bit Window in DVM, some may noticed from Table 2 that THG-PBC uses Double and Add algorithm without k-bit Window but it is faster than that of jPBC. The cause of this situation is not due to the algorithm used, but the class structures of the two libraries. THG-PBC comprises of only four classes without any inheritance or interfaces but jPBC on the other hand, uses a relatively complex class structures.

Thirdly, the use of global methods can speed up the execution time in

DVM. Every point operations in Jpair are the slowest among all but its pairing operation is still the fastest after mPBC. This is because the multiplication under $\mathbb{F}_{p^2}$ in Jpair is done by the `ComplexField` object, which provides all the field operations. So, instead of calling the multiplication method from a new $\mathbb{F}_{p^2}$ object, DVM calls the same `ComplexField` object in every round of the Miller algorithm.

5 CASE STUDY

In this section, we provide a proof of concept for mPBC by implementing the Identity-Based Signature scheme by Cha and Cheon (2002) (CC-IBS). We describe CC-IBS scheme as follows:

Setup(1^k). Generate an additive group $\mathbb{G}$ with prime order q. Choose random $P \in \mathbb{G}$ and $s \in \mathbb{Z}_q^*$. Let the master public key be $mpk = (P, P_{pub} = sP, H)$ and master secret key be $msk = s$ where $H : \{0,1\}^* \rightarrow \mathbb{G}$.

Extract(mpk, msk, ID). Given a public identity ID, compute user secret key as $D = sQ$ where $Q = H(ID)$.

Sign(mpk, msk, m). Select a random $r \in \mathbb{Z}_q^*$ and compute $U = rQ$. Next, compute $V = (r + h)D$ where $h = H(m, U)$. The signature σ is a tuple $\sigma = (U, V)$.

Verify(mpk, σ, ID). Compute $Q = H(ID)$ and accept the signature if $e(P, V) = e(P_{pub}, U + hQ)$ where $h = H(m, U)$; reject otherwise.

Setting ID as "user@gmail.com", m as random string and $k = 5$ for lookup table in window method, we benchmark the CC-IBS scheme in DVM with the same specification as mentioned in Section 1.1. The results are as depicted in Table 4.

The Setup and Extract of CC-IBS are the same as that of BLS-IBI (Kurosawa and Heng, 2004) scheme, which was presented in the case study of THG-PBC (Tan et al., 2010). Without pre-processing, the Setup and Extract from

Table 4: Timing (ns) of CC-IBS using mPBC in DVM

Algo	Without Pre-Processing	With Pre-Processing
Setup	764,974,711	1,397,281,960
Extract	399,742,484	2,321,424,874
Sign	239,167,466	42,021,442
Verify	1,286,640,424	1,099,376,141

THG-PBC recorded 1330ms and 229ms respectively which is 50% slower than that of mPBC. Furthermore, the BLS-IBI scheme of THG-PBC was benchmarked on a computer with Pentium M 1.73 Ghz processor while CC-IBS was only benchmarked on an Android smart phone with ARM Cortext-A9 1.4 Ghz dual core processor.

The results obtained also show the obvious difference between timing of pre-processing and without pre-processing. The initiations of pre-processing for pairing lies in the Setup function while the construction of pre-processing table for point multiplication is located in the Extract function. By doing so, we pushed the heavy processes to Setup and Extract as well as attaching $Table_{PP}$ into mpk. The user is left with efficient Sign and Verify but with larger system parameters. Unless the Android device has very limited memory, we believe the 52KB-larger mpk is acceptable in practice. Furthermore, the Setup and Extract algorithms are normally run by server and so the slow processing time does not affect the usability on Android devices.

Remark 5.1. *Although the Sign algorithm from BLS signature (Boneh et al., 2004) scheme is similar to the Extract algorithm of CC-IBS, we do not compare the benchmark of CC-IBS to that of the BLS signature in De Caro and Iovino (2011) and Liu et al. (2014). This is because De Caro and Iovino (2011) did not provide the benchmark on BLS signature scheme while Liu et al. (2014) is not a full java library as the core library is still in C language.*

ACKNOWLEDGEMENTS

This work is supported by the FRGS grant (FRGS/2/2014/ICT04/MMU/03/1).

REFERENCES

Android developers: Performance tips. http://developer.android.com/training/articles/perf-tips.html.

Boneh, D., Lynn, B., and Shacham, H. (2004). Short signatures from the weil pairing. *Journal of Cryptology*, 17(4):297–319.

Cha, J. C. and Cheon, J. H. (2002). *Public Key Cryptography — PKC 2003: 6th International Workshop on Practice and Theory in Public Key Cryptography Miami, FL, USA, January 6–8, 2003 Proceedings*, chapter An Identity-Based Signature from Gap Diffie-Hellman Groups, pages 18–30. Springer Berlin Heidelberg, Berlin, Heidelberg.

De Caro, A. and Iovino, V. (2011). jpbc: Java pairing based cryptography. In *Proceedings of the 16th IEEE Symposium on Computers and Communications, ISCC 2011*, pages 850–855, Kerkyra, Corfu, Greece, June 28 - July 1. IEEE.

Dong, C. (2010). Jpair: A quick introduction. https://personal.cis.strath.ac.uk/changyu.dong/jpair/intro.html.

Kurosawa, K. and Heng, S.-H. (2004). *Public Key Cryptography – PKC 2004: 7th International Workshop on Theory and Practice in Public Key Cryptography, Singapore, March 1-4, 2004. Proceedings*, chapter From Digital Signature to ID-based Identification/Signature, pages 248–261. Springer Berlin Heidelberg, Berlin, Heidelberg.

Liu, W., Liu, J., Wu, Q., and Qin, B. (2014). Android pbc: A pairing based cryptography toolkit for android platform. In *Communications Security Conference (CSC 2014), 2014*, pages 1–6.

Lynn, B. (2010). The pairing-based cryptography library. http://crypto.stanford.edu/pbc/.

Tan, S.-Y., Heng, S.-H., and Goi, B.-M. (2010). Java implementation for pairing-based cryptosystems. In *Computational Science and Its Applications ICCSA 2010*, volume 6019 of *Lecture Notes in Computer Science*, pages 188–198. Springer Berlin Heidelberg.