ANDROID PBC: A Pairing Based Cryptography Toolkit for Android Platform

Weiran Liu[1,3], Jianwei Liu[1,2] , Qianhong Wu[1,2] , Bo Qin[2,3]

[1]*The School of Electronisc and Information Engineering, Beihang University, Beijing, China*
[2]*The Academy of Satellite Application, Beijing, 100086, China*
[3]*The School of Information, Renmin University of China, Beijing, China*

Keywords: Pairing-Based Cryptography, Java, Android

Abstract

There is an increasing demand and necessity to deploy advanced cryptographic approaches to secure mobile platforms, especially for Android, which is one of the most popular platforms with continual growth rate in Market. Among newly developed cryptographic libraries, Pairing-Based Cryptography (PBC) is a versatile and well-known cryptography library written in C language that has been widely used both in academia and industry. Unfortunately, there is no public Java PBC library available to Android 2.3 or higher versions. For our own research purpose, we have developed a fully functional PBC library wrapping cryptographic algorithms in Java, called Android PBC, which is suitable to all Android versions. Great efforts have been made so that PBC library can run as fast as possible in source-limited Android platforms. We provide the details of our Android PBC with benchmark tests performed to measure the gap between Android PBC and Linux PBC, together with a typical example, so that one can easily deploy cryptographic algorithms and protocols on Android platforms by following the examples.

1 INTRODUCTION

In the last decade, finite cyclic groups equipped with efficient bilinear maps (i.e., bilinear pairings) have been shown a powerful mathematic tool in cryptography. The initial use of bilinear pairings was negative, i.e., to break cryptosystems [1-2] that are built from supersingular elliptical curves. Joux [3] presented the first constructive scheme of a one-round tripartite Diffie-Hellman protocol based on bilinear pairings In 2001, Boneh *et al.* constructed an Identity-Based Encryption scheme using bilinear maps [4]. Since then, numerous versatile and fantastic cryptosystems have been built based on bilinear pairings with additional useful properties, such as Identity-Based Proxy Verifiably Encrypted Signature [5], Asymmetric Group Key Agreement [6-7], Contributory Broadcast Encryption [8], Hierarchical Identity-Based Encryption [9], Identity-Based Broadcast Encryption [10], Identity-Based Signature [11], Identity-Based Authenticated Key Agreement [12], etc.

For implementing bilinear maps in practical usage, Ben Lynn [13-14] provided a complete and freely available implementation of the bilinear maps for cryptography. The library, named Pairing-Based Cryptography (PBC), provides the implementation of different types of elliptic curves that are efficiently computable and cryptographically secure. It is a versatile and well-known cryptography library that has been widely used both in academia and industry.

In the last ten years, with more and more mobile electronic devices connected to Internet and continual progressing in embedded CPUs [15-16], there is an increasing demand and necessity to deploy cryptographic approaches to secure mobile platforms. Among them, Android platform, which have been widely used in mobile devices, becomes the most popular platform in practice. It is preferable to implement advanced cryptographic approaches based on bilinear pairings to protect communication and storage models in Android.

It has been shown challenging to implement Pairing-Based Cryptography on Android. The well-known PBC library is implemented by C language, while applications on Android platform are implemented by Java. The gap between the two programming languages makes the PBC hard to be applied in Android. Separately implementing Pairing-Based Cryptography in Java seems reasonable. However, the code executing in Java is far slower than running in C. Note that pairing and exponent operations on bilinear pairing groups are relatively slow, the total time processing pairing and exponent operations in Java is considerably unacceptable, even on a high performance mobile device [17].

To the best of our knowledge, the only public PBC in Java is provided by Angelo De Caro *et al.* [18-19]. Their project is called Java Pairing-Based Cryptography (jPBC). jPBC is a Java wrapper on the PBC library, that is, jPBC provides full interfaces and classes to simplify the use of the bilinear maps in Java by using PBC as the basic calculating library. The result shows that jPBC supports all functions provided by PBC in Java.

There are limitations in jPBC. In practice, Many Pairing-Based cryptosystems have been realized using original PBC library. It is preferable if one can port these instantiations to embedded systems with minor modifications. However, if an engineer uses jPBC to implement these cryptosystems on Android, he has to recode all algorithms in Java instead of modifying code writing in C, which incurs considerable repetitive workloads. Also, implementing cryptosystems in Java using jPBC leads to unnecessary extra operations, since jPBC causes frequently forwarding and submitting data from Java to C. This is not a serious problem to personal computer. However, it is unacceptable in Android-like embedded systems since resources are rather limited in such systems. Moreover, to the best of our knowledge, the Android version of jPBC can only be used on Android version 2.1 or lower.

Up to now, there exist no PBC libraries on Android platform that can be used on Android 2.3 or higher versions.

In this paper, we implement a fully functional PBC library on Android platform, named Android PBC. We have received a number of requests for Android PBC from peer research groups and questions on their usage. To avoid repetitive development and share with more researchers and security practitioners, we would like to publish the challenges and solutions in realizing PBC in Java for Android-like platforms, as an opportunity to answer frequently asked questions. Instead of fully wrapping PBC library like jPBC, Android PBC just wrap algorithms for cryptosystems. Developers can easily port implemented Pairing-Based cryptosystems writing in C to Android by just slightly modifying the source code, instead of fully recoding the algorithms in Java. Also, wrapping algorithms at a cryptosystem level can reduce extra operations caused by forwarding and submitting data between Java and C. By modifying the source codes that do not satisfy standard C in PBC library, Android PBC can be used on any version of Android, including Android 2.3 or higher versions. We perform extensive benchmark tests to measure the gap between Android PBC and Linux PBC. We further implement all examples provided by PBC library and test the performances. By following these examples, one can easily deploy cryptographic algorithms and protocols on Android platforms to develop new security applications.

2 BACKGROUND

2.1 Bilinear Maps

Let $\mathbb{G}_1$, $\mathbb{G}_2$ and $\mathbb{G}_T$ be cyclic groups of prime order p. Let g_1 be a generator of $\mathbb{G}_1$ and g_2 be the generator of $\mathbb{G}_2$. A bilinear map $e: \mathbb{G}_1 \times \mathbb{G}_2 \rightarrow \mathbb{G}_T$ is an efficiently computable function satisfying the following properties:

- *Bilinearity*: for all a, $b \in \mathbb{Z}_p$, $e(g_1^a, g_2^b) = e(g_1, g_2)^{ab}$;
- *Non-degeneracy*: $e(g_1, g_2)$ has order p in $\mathbb{G}_T$;
- *Computability*: There exists an efficient algorithm to compute $e(u, v)$ for all $u \in \mathbb{G}_1$ and v $\in \mathbb{G}_2$.

The tuple $(p, g_1, g_2, \mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T)$ is called an asymmetric bilinear setting. If $\mathbb{G}_1$ and $\mathbb{G}_2$ are the same cyclic groups with a generator g, we say the tuple $(p, g, \mathbb{G}_1, \mathbb{G}_T)$ is a symmetric bilinear setting. For simplicity, in this paper we only consider the symmetric bilinear settings.

The bilinear map was first used for breaking decisional Diffie-Hellman assumptions in groups with such maps. The decision Diffie-Hellman problem states that: for a cyclic group $\mathbb{G}$ with a generator g, given $(g^a, g^b, h) \in \mathbb{G}^3$, determine whether or not $h = g^{ab}$. Indeed, if there exists an efficient bilinear map $e: \mathbb{G} \times \mathbb{G} \rightarrow \mathbb{G}_T$, it is possible to solve decision Diffie-Hellman problem in polynomial time by checking $e(g, h) = e(g^a, g^b)$.

It is known that one can apply elliptic curve over a finite field to construct bilinear maps [18]. Given the points of an elliptic curve over a finite field, it is possible to define a multiplication operation so that the set of points on this curve constitute a cyclic group, on which there may be possible to define a bilinear map. Galbraith *et al.* [20] summaries the properties for different curves equipped with bilinear maps.

Following the studies of Lynn [13], and the manual of PBC libraries [14], the curves supporting bilinear maps are listed as follows:

- **Type A**: Type A pairings are constructed on the curve $y^2 = x^3 + x$ over the field F_q for some prime $q = 3 \bmod 4$, with embedding degree 2. Type A pairing is symmetric, since both $\mathbb{G}_1$ and $\mathbb{G}_2$ are the group of points $E(F_q)$.
- **Type A1**: Type A1 also uses the curve $y^2 = x^3 + x$, but with a specific order, e.g. $N = p \cdot q$, where p and q are large primes so that N is hard to factorize.
- **Type B**: The curve $y^2 = x^3 + 1$ over the field F_q for some prime $q = 2 \bmod 3$, which implies cube roots in F_q are easy to compute. This type of curve is unimplemented in PBC library.
- **Type C**: The supersingular curves $y^2 = x^3 + 2x + 1$ and $y^2 = x^3 + 2x - 1$ over a field of characteristic 3. This type of curve is unimplemented in PBC library.
- **Type D**: The type D curve is defined over some field F_q and has order $h * r$ where r is a prime and h is a small constant. Over the field F_{q^6} its order is a multiple of r^2. This type of curve is constructed by using the Complex Multiplication (CM) method.
- **Type E**: The CM method of constructing elliptic curves of embedding degree 1.
- **Type F**: Type F is a $k = 12$ pairings using carefully crafted polynomials.
- **Type G**: It is another construction based on the CM method with embedding degree 10.

2.2 Android NDK

Android NDK is a toolset that allows Android developers to implement parts of their application using native code languages such as C and C++.

There are advantages and disadvantages of using native code in Android. For certain types of applications, developers can reuse existing code libraries written in C/C++ to reduce unnecessary developing workloads. Also, since native codes are all run in native level, its running time is always faster than running in Java level [21]. However, using Android NDK in an application inevitably increases application complexity.

Android NDK Revision 9 was released in July 2013. This version of NDK provides additional tools to help developers for coding in C/C++. We recommend the reader to get more details about Android NDK in [22].

2.3 Java Native Interface

Java Native Interface (JNI) [23] is a standard Java programming interface for writing Java native methods using C/C++ and embedding the Java virtual machine into native applications.

It is also possible to use the JNI interface in Android by applying Android NDK to embed native code libraries in Android applications. Through the help of JNI, the native functions can be called from the application running on Java virtual machine in Android.

3 ANDROID PBC: IMPLEMENTATIONS

3.1 Porting GMP

Since the PBC library is built on top of the GMP library and the PBC API is strongly influenced by the GMP API, we should first port GMP into Android. Fortunately, there exists a prebuilt copy of GMP compiled with the Android NDK r8e, called *GMP 5 for Android* [24]. Following the method presented in [24], we can directly include and install the GMP library on Android devices.

3.2 Modifying PBC Source Code

After including GMP on Android, we should slightly modify the PBC source code, or the application would be crushed whenever calling PBC functions. This is because Android native C and Linux C have different memory management mechanisms. Android native code does not automatically allocate memories for strings or other arrays with unknown size, which will lead to "*pointer out of size*" error. This kind of error does not happen in Linux C, while leads crash in Android due to the limitation of memories in embedded system.

As an example, the following code in PBC library (/ecc/a_param.c) writing in C would make the application crash in Android, while it can be successfully executed in Linux C. In this example, the variable *tab* is a function pointer with a char pointer as input. The char pointer has unknown size without allocating memories beforehand. Directly assigning values of p, q, r using pointer *tab* causes "*pointer out of size error*", thus crushes the application.

```
int pbc_param_init_a(pbc_param_ptr par,
  const char *(*tab)(const char *)) {
  a_param_init(par);
  a_param_ptr p = par->data;

  int err = 0;
  err += lookup_mpz (p->q, tab, "q");
  err += lookup_mpz (p->r, tab, "r");
  err += lookup_mpz (p->h, tab, "h");
  err += lookup_int (&p->exp2, tab, "exp2");
  err += lookup_int (&p->exp1, tab, "exp1");
  err += lookup_int (&p->sign1, tab, "sign1");
  err += lookup_int (&p->sign0, tab, "sign0");
  return err;
}
```

We fix these kinds of problems when porting PBC in Android by directly writing system parameters into source code, instead of reading parameters from local file like original PBC library. This is reasonable since reading file may cause additional operations, which can be reduced with the consideration of limited resources in embedded systems. The following code shows the fixed version.

```
int pbc_param_init_a(pbc_param_ptr par,
  const char *(*tab)(const char *)) {
  a_param_init(par);
  a_param_ptr p = par->data;

  int err = 0;
  char q[] =
  "87807107996633125224377819847540498
  15806883199414208211028653399266475 6
  30880222957078625179422662221423155 8
  58769582317459277713367317481324925 1
  29998224791";
  char h[] =
  "12016012264891146079388821366740534
  20480295440125131182291961513104720 7
  28935970453110284480218390653778677 6";
  char r[] = "73075081866545162136111 9
  24557150490140597655961 7";
  char exp2[] = "159";
  char exp1[] = "107";
  char sign1[] = "1";
  char sign0[] = "1";
  err += new_lookup_mpz(p->q, q);
  err += new_lookup_mpz(p->r, r);
  err += new_lookup_mpz(p->h, h);
  err += new_lookup_int(&p->exp2, exp2);
  err += new_lookup_int(&p->exp1, exp1);
  err += new_lookup_int(&p->sign1, sign1)
  err += new_lookup_int(&p->sign0, sign0);
  return err;
}
```

The similar modification should also be done in source codes listed below:

- /ecc/d_param.c
- /ecc/e_param.c
- /ecc/f_param.c
- /ecc/g_param.c

3.3 Porting PBC

After modifying the PBC source codes, we need to further write necessary files for PBC that are used to help Android NDK compile and generate static library for Android platforms.

Each folder in PBC source code that containing C files should include a file named *source.mk*, which helps to include all C files into Android NDK for building static library. The following code shows the *source.mk* in the folder pbc/arith/.

```
LOCAL_SRC_FILES +=
arith/dlog.c\
arith/field.c\
```

```
arith/init_random.c\
arith/multiz.c\
arith/random.c\
arith/z.c\
arith/fasterfp.c\
arith/fieldquadratic.c\
arith/naivefp.c\
arith/fastfp.c\
arith/fp.c\
arith/montfp.c\
arith/poly.c\
arith/tinyfp.c
```

Similarly, folders in PBC source code listed below should also contain *source.mk* files.

- pbc/benchmark/
- pbc/ecc/
- pbc/guru/
- pbc/misc/
- pbc/pbc/

Then, we should write a make file named *Android.mk*, which helps Android NDK compiler to collect all *source.mk* files for building and generating static library for PBC. The following code displays the example *Android.mk* file.

```
LOCAL_PATH:=$(call my-dir)
ROOT_PATH:=$(LOCAL_PATH)

include$(call all-subdir-makefiles)
include$(CLEAR_VARS)

LOCAL_PATH:=$(ROOT_PATH)
LOCAL_CFLAGS:=-Wall -Wextra

include$(LOCAL_PATH)/arith/source.mk
include$(LOCAL_PATH)/benchmark/source.mk
include$(LOCAL_PATH)/ecc/source.mk
include$(LOCAL_PATH)/gen/source.mk
include$(LOCAL_PATH)/guru/source.mk
include$(LOCAL_PATH)/misc/source.mk
include$(LOCAL_PATH)/pbc/source.mk
LOCAL_MODULE:=pbc
LOCAL_LDLIBS:=-L$(SYSROOT)/usr/lib -llog
LOCAL_SHARED_LIBRARIES:=gmp
LOCAL_C_FLAGS:=-g
include $(BUILD_SHARED_LIBRARY)
```

Finally, Android NDK compiler can build and generate the PBC static library file *libpbc.so* by running shell script *ndk-build*. Now we finish porting PBC on Android platform.

3.4 Benchmark

A benchmark comparison between Android PBC and Linux PBC has been conducted to measure the gap between the two libraries. Table 1 shows the test platforms of Android PBC and Linux PBC libraries.

Test Platform	*PBC in Linux*	*Android PBC*
Platform	PC	Samsung I9100
CPU Series	Inter Core i5-2400	Samsung Exynos 4210
CPU Clock Speed	3.10GHz * 4	1.2GHz * 2
RAM	2GB	1GB
Operation System	Ubuntu 12.04	Android 4.0.3
JDK && SDK Version	Open JDK 1.6.0	SDK Tools 22.0.4

Table 1: Test Platform

Table 2 demonstrates the timing performance of the Android PBC and Linux PBC libraries on a Type A curve with base field size of 512 bits. It can be seen that there is a significant gap between the two libraries: Android PBC is nearly 5 times slower than Linux PBC. Unfortunately, the impossibility of running jPBC [19] on Android 2.3 or higher version prevents us from comparing the performance of our library with that of the jPBC.

Operation(millisecond)	*PBC in Linux*	*Android PBC*
Pairing#Pairing$(\cdot,\cdot)$	3.5700	27.039
Pairing#Pow$(\cdot,\cdot)$ in $\mathbb{G}$	3.3156	13.589
Pairing#Pow$(\cdot,\cdot)$ in $\mathbb{G}_T$	0.4120	2.5320
Pairing#Pow$(\cdot,\cdot)$ in $\mathbb{Z}_r$	0.0264	0.1230

Table 2: Benchmark on Android and Linux

4 Implementing BLS Short Signature Using Android PBC

In this section we introduce how to implement the BLS short signature scheme using Android PBC. We provide implementation details to explain the approach to port BLS scheme from the C source code in PBC library [14] to Android application, which indicates a guideline so that researchers, security engineers can build their security applications by following this example.

4.1 BLS Signature Scheme

The BLS signature scheme [25] comprises four algorithms: **Setup**, **KeyGen**, **Signing** and **Verification**. It makes use of a full-domain hash function $H: \{0,1\}^* \rightarrow \mathbb{G}^*$.

- **Setup**. It chooses a system parameter to generate two groups $\mathbb{G}$, $\mathbb{G}_T$ of order p, and a bilinear map $e: \mathbb{G} \times \mathbb{G} \rightarrow \mathbb{G}_T$. The parameter is $param \leftarrow (p, g, \mathbb{G}, \mathbb{G}_T)$
- **KeyGen**. The algorithm picks a random $x \leftarrow \mathbb{Z}_r$ and computes $v \leftarrow g^x$. The public key is $v \in \mathbb{G}^*$. The corresponding secret key is $x \in \mathbb{Z}_r$.
- **Signing**. Given a secret key $x \in \mathbb{Z}_r$, and a message $M \in \{0,1\}^*$, it computes $h \leftarrow H(m) \in \mathbb{G}^*$ and $\sigma \leftarrow h^x$. The signature is $\sigma \in \mathbb{G}^*$.
- **Verification**. Given a public key $v \in \mathbb{G}^*$, a message $M \in \{0,1\}^*$, and a signature $\sigma \in \mathbb{G}^*$, the algorithm computes $h \leftarrow H(m) \in \mathbb{G}^*$ and verifies that (g, v, h, σ) is a valid Diffie-Hellman tuple by checking whether $(g, \sigma) = e(v, h)$ holds. If the equality holds, the algorithm accepts the signature and returns VALID. Otherwise, it returns INVALID.

Boneh *et al.* show that the BLS scheme is secure against existential forgery under a chosen message attack in the random oracle model assuming that the Computational Diffie-Hellman assumption holds in $\mathbb{G}$ [25].

4.2 Implementation

We can use the C source code for BLS signature scheme in PBC source code [14] to implement BLS on Android platform. Compared with jPBC, we do not need to recode BLS in Java. Instead, we just need to slightly modify the C source code of BLS.

Figure 2 shows the BLS architecture on Android PBC. The functions are all placed in the class *BLS.java*, which contains four native functions: *Setup*, *KeyGen*, *Signing*, *Verification*. Note that these four functions are identical with algorithms defined in [25]. In addition, the fifth function, *InitBLSJni*, which will be executed statically, is included to help the system make necessary initialization operations.

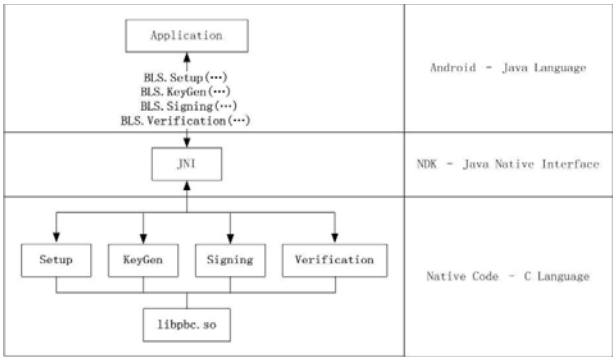

Figure 1: BLS Architecture

When executing these functions, Java directly forwards the parameters from Java to C by using JNI. JNI unpackages the parameters, and then calls PBC functions in the library *libPBC.so* to do the calculation. Finally, the results are packaged and returned to Java.

The source code below shows the implementation of *BLS.java*. As expected, *BLS.java* includes five native functions linked with five C functions in the JNI level.

```
package com.androidpbc.bls;

public class BLS{
  public static native void
    InitBLSjni();
  public static native BLS_Parameter
    Setup();
  public static native BLS_Keys
    KeyGen(BLS_Parameter parameter);
  public static native BLS_Signature
    Sign(BLS_Prikey key, String msg);
  public static native boolean
    Verify(BLS_Parameter parameter,
    BLS_Pubkey pubkey, String msg,
    BLS_Signature signature);
}
```

By compiling *BLS.java* with the javah command, we can obtain a header file in *C*, named com_androidpbc_bls_BLS.h, which contains five *C* functions corresponding to the five functions in *BLS.java*:

- JNIEXPORT void JNICALL \\
  Java_com_androidpbc_bls_BLS_InitBLSjni \\
  (JNIEnv *, jclass);
- JNIEXPORT jobject JNICALL \\
  Java_com_androidpbc_bls_BLS_Setup \\
  (JNIEnv *, jclass);
- JNIEXPORT jobject JNICALL \\
  Java_com_androidpbc_bls_BLS_KeyGen \\
  (JNIEnv *, jclass, jobject);
- JNIEXPORT jobject JNICALL \\
  Java_com_androidpbc_bls_BLS_Sign \\
  (JNIEnv *, jclass, jobject, jstring);
- JNIEXPORT jboolean JNICALL \\
  Java_com_androidpbc_bls_BLS_Verify \\
  (JNIEnv *, jclass, jobject, jobject, jstring, jobject);

After creating a source code file and including the header file *com_androidpbc_bls_BLS.h* into it, we can then copy and modify the source code provided by PBC to the source code file for implementing these functions using the precompiled library *libPBC.so*.

Figure 2 shows the result obtained by running BLS signature scheme on Android platform. The log information displayed in figure 2 proves that BLS signature scheme can be successfully run on Android 2.3 or higher version.

4.3 Performance Measurements

We now compare the performance achieved by the BLS scheme we developed on Android platform comparing with the scheme running on Linux platform. Table 3 displays measurements of BLS scheme. We separately list the signature time and verification time. The result shows that BLS scheme running on Android is about 4 times slower than on Linux, which is consistent with the benchmarks shown in Section 3.

Function (second)	*Linux PBC*	*Android PBC*
BLS Sign	0.0108	0.0478
BLS Verification	0.0070	0.0325

Table 3: Performance Measurements on Android and Linux

5 Conclusion

In this paper, we implemented a fully functional PBC library on Android platform, called Android PBC. Unlike jPBC, Android PBC just wraps algorithms for cryptosystems so that developers can easily port implemented Pairing-Based cryptosystems writing in C to Android by just slightly modifying the source code, instead of fully rewriting the algorithms in Java. Wrapping algorithms at a cryptosystem level can also reduce extra operations when forwarding and

returning data between Java and C. We performed extensive benchmark tests to measure the gap between Android PBC and Linux PBC. We further implemented BLS signature scheme and test the performances, for the purpose to help researchers to execute their own cryptographic algorithms and protocols on Android platforms, and help security practitioners to easily develop their own security applications using cryptographic schemes encoded in Linux C. Our Android PBC can be used on any version of Android, including Android version 2.3 or higher versions.

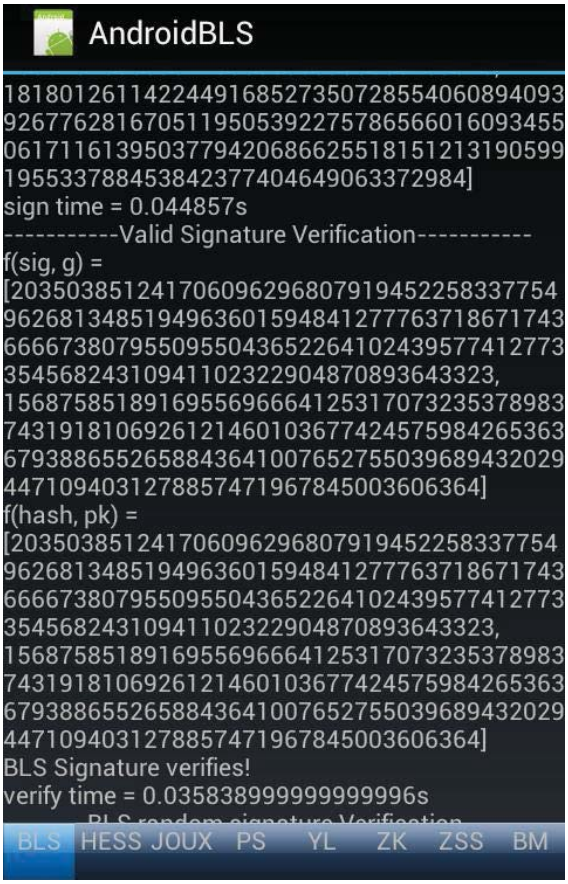

Figure 2: BLS Running on Android

ACKNOWLEDGMENTS

This paper is partially supported by the National Key Basic Research Program (973 program) through project 2012CB315905, the Natural Science Foundation of China through projects 61272501, 61370190, 61173154, 61003214, 61202465 and 60970116, the Beijing Natural Science Foundation through project 4132056, the Fundamental Research Funds for the Central Universities through the Research Funds of Renmin University of China, the Open Research Fund of The Academy of Satellite Application and the Open Research Fund of Beijing Key Laboratory of Trusted Computing.

REFERENCES

[1] M. Alfred J, T. Okamoto, S.A. Vanstone: "Reducing elliptic curve logarithms to logarithms in a finite field", In: IEEE Trans. on Info. Theo., 1993, 39, (5), pp. 1639-1646

[2] F. Gerhard, M. Muller, H-G. Ruck: "The Tate pairing and the discrete logarithm applied to elliptic curve cryptosystems", IEEE Trans. on Info. Theo., 1999, 45, (5), pp. 1717-1719

[3] A. Joux: "A one round protocol for tripartite Diffie–Hellman", Proc. Int. Symp. Algorithmic number theory, Leiden, The Netherlands, Jul. 2000, pp. 385-393

[4] D. Boneh, M. Franklin: "Identity-based encryption from the Weil pairing", Proc. Inc. Conf. Crypto, CA, USA, Aug. 2001, pp. 213-229

[5] J.H. Liu, J.W. Liu, X.F. Qiu: "Identity-based proxy verifiably encrypted signature scheme", China Com. 2012, 9, 11, pp. 137-149

[6] Q.H. Wu, X.Y. Zhang, M. Tang, P. Yin, Z.L. Qiu: "Extended asymmetric group key agreement for dynamic groups and its applications", China Com. 2011, 8, (4), pp. 32-40

[7] Q.H. Wu, Y. Mu, W. Susilo, B. Qin, J. Domingo-Ferrer: "Asymmetric group key agreement", Int. Conf. Eurocrypt, Cologne, Germany, Apr. 2009, pp. 153-170

[8] Q.H. Wu, B. Qin, L. Zhang, J. Domingo-Ferrer, O. Farras: "Bridging broadcast encryption and group key agreement", Proc. Int. Conf. Asiacrypt, Seoul, South Korea, Dec. 2011, pp. 143–160

[9] C. Gentry, A. Silverberg: "Hierarchical ID-based cryptography", Int. Conf. Asiacrypt, Queenstown, New Zealand, Dec. 2002, pp. 548-566

[10] C. Delerablée: "Identity-based broadcast encryption with constant size ciphertexts and private keys", Proc. Int. Conf. Asiacrypt, Kuching, Malaysia, Dec. 2007, pp. 200-215

[11] F. Hess: "Efficient identity based signature schemes based on pairings", Int. Work. Selected Areas in Cryptography, Newfoundland, Canada, Aug. 2002 pp. 310-324

[12] N.P. Smart: "Identity-based authenticated key agreement protocol based on Weil pairing", Elec. letters 2002, 38, (13), pp. 630-632

[13] B. Lynn: "On the implementation of pairing-based cryptography", PhD Thesis, Stanford University, 2007

[14] http://crypto.stanford.edu/pbc, accessed July 2013

[15] Q.H. Wu, B. Qin, L. Zhang, J. Domingo-Ferrer, J.A. Manjón: "Fast transmission to remote cooperative groups: a new key management paradigm", IEEE/ACM Trans. on Net. 2012, 21, (2), pp. 621-633

[16] Q.H. Wu, J. Domingo-Ferrer, Ú. González-Nicolás: "Balanced trustworthiness, Safety and Privacy in Vehicle-to-Vehicle Communications", IEEE Trans. on Vehi. Tech. 2010, 59, (2), pp. 559–573

[17] S.Y. Tan, S.H. Heng, B.M. Goi: "Java implementation for pairing-based cryptosystems", Int. Conf. Computational Science and Its Applications, Fukuoka, Japan, Mar. 2010, pp. 188-198

[18] A.D. Caro, V. Iovino: "jPBC: Java pairing based cryptography", IEEE Symp. on Comp. and Comm. 2011, pp. 850-855

[19] http://gas.dia.unisa.it/projects/jpbc, accessed July 2013

[20] S.D. Galbraith, K.G. Paterson, N.P. Smart: "Pairings for cryptographers", Disc. App. Math. 2008, 156, (16), pp. 3113-3121

[21] S. Lee, J.W. Jeon: "Evaluating performance of Android platform using native C for embedded systems", IEEE Int. Conf. Control Automation and Systems, Oct. 2010, pp. 1160-1163

[22] http://developer.android.com/tools/sdk/ndk/index.html, accessed July 2013

[23] http://docs.oracle.com/javase/6/docs/technotes/guides/jni/, accessed July 2013

[24] https://github.com/Rupan/gmp, accessed July 2013

[25] D. Boneh, B. Lynn, H. Shacham: "Short signatures from the Weil pairing", Int. Conf. Asiacrypt, Gold Coast, Australia, Dec. 2001, pp. 514-532