International Journal of Cryptology Research 6(1): 31 - 46 (2016)

Identity-Based Signature using Bilinear Pairings on iPhones

Wai-Choong Wong[1], **Tiong-Sik Ng**[1], and **Ji-Jian Chin** [*1]

[1]*Faculty of Engineering, Multimedia University*

E-mail: jjchin@mmu.edu.my
[*]*Corresponding author*

ABSTRACT

Many cryptographic protocols based on bilinear pairings were introduced over the years with various implementations such as identity-based identification (IBI) and identity-based signature (IBS). While Pairing Based Crypto (PBC) library is available as one of the most well-known open source pairing-based library based on elliptic curve cryptography (ECC), Tan et al. (2010) had developed a Java-based pairing library that functions in a similar manner as well as provides comparative performance based on pairing time. However, Tan et al. (2010)'s library does not support the development on Apple's iOS platform due to the distinct programming language it is written in. Recently Cheah et al. (2015) successfully simulated a pairing-based IBI scheme in iOS based on Tan et al. (2010)'s library by translating it but did not manage to extend his work as a mobile application due to linker error. In this work, we managed to overcome that problem and successfully implemented a pairing-based IBS scheme with Tan et al. (2010)'s library for mobile devices such as iPhones.

Keywords: Bilinear pairings, identity-based signature, iOS, Apple, iPhone

1 INTRODUCTION

According to the statistic shown by Pew Reserch Center, the ownership for mobile devices such as smartphones and tablets have been steadily growing over the recent years (Pew-Research-Center, 2015). People nowadays are relying on these smart devices to communicate with each other. Wireless connections between the mobile devices are convenient but have much to be desired in terms of security. Thus, identity authentication and verification processes between both parties are mandatory to ensure a secured environment for wireless communication. Cryptography plays a huge role in providing confidentiality and integrity in these situations.

Based on International Data Corporation (IDC)'s analysis in 2015, Google's Android OS is holding 82.8% of market share, followed by 13.9% for Apple's iOS, 2.6% for Windows Phone, and 0.7% for others (International-Data-Corporation, 2015). Even though iOS is having a large user base, it appears that pairing-based cryptographic libraries are almost non-existent and has yet to receive widespread support.

1.1 Motivations and related work

Our primary motivation for this work is Cheah et al. (2015)'s successful work that showed an efficient identity-based identification scheme using ECC pairing-based libraries can be implemented in iOS by translating the Java-based library by Tan et al. (2010). Their work is significant as it shows a working pairing-based security scheme on iOS platform which could be further implemented for various security applications in the industry. Also, pairing-based implementations for the iOS platform are still rare at this time. Our goal for conducting this work is add to the research and application of pairing-based cryptography on iOS mobile platform.

1.2 Contributions

In this work, the IBS scheme implemented was proposed by Cha and Cheon (2003). We simulate the scheme by translating Tan et al. (2010)'s library and develop a mobile application for iOS to implement the scheme mentioned. Our achievement is an extension of Cheah et al. (2015)'s results, where the authors only managed to implement a simulator for the Mac. In contrast, our implementation successfully runs on iPhones. To our knowledge, this is the first successful implementation of a pairing-based IBS scheme on iOS platform for mobile devices using Tan et al. (2010)'s library. In addition to the IBS, we also implemented the ECC version of Elgamal public key encryption (PKE) scheme Elgamal (1985) using the same ECC pairing-based library.

The advantage of running pairing-based cryptosystems is that the secret key length can be much shorter than pairing-free alternatives, such as RSA or DSA. We observe the running time for different stages of the scheme by using both simulator and mobile devices with efficient results as well. We describe the methods and procedures we used for conducting this work in detail.

The rest of the paper is organized as such. In section 2, we will first describe the Cha-Cheon IBS scheme and its stages in detail. In section 3, we will then describe the ECC ElGamal scheme and also its stages in detail. In section 4, we will show how we developed the mobile application using available tools. In section 5, we will show our results on both simulator and iOS devices. Section 6 is the conclusion of our work.

2 THE CHA-CHEON IBS SCHEME

2.1 Preliminaries

Notation-wise, let $\{0,1\}^*$ denote the set of all bit strings while $\{0,1\}^n$ the set of bit strings of length n. If a string $s \in \{0,1\}^*$ then $|s|$ denotes the length of s. If S is a set then $|S|$ denotes the size of S. Let $x \overset{\$}{\leftarrow} S$ denote a randomly and uniformly chosen element x from a finite set S. An elliptic curve E is set up over a field F_q. A public function $f : m \to P_m$ maps messages m to points P_m on E. Lastly let Z_p denote the set of positive integers modulo a large prime p.

Let G be a group of prime order p and let q be a large prime where $q = p/2-1$. G is a group in which Computational Diffie-Hellman problem (CDHP) is considered intractable to be solved. Let P be a random generator in G. The DDHP problem is defined as: Given (P, aP, bP) as a Diffie-Hellman tuple, where $a, b \overset{\$}{\leftarrow} Z_q$, calculate abP. The Cha-Cheon IBS scheme is proven to be secure if the CDHP is intractable.

2.2 Bilinear Pairings

Secondly, Cha-Cheon IBS scheme runs using a bilinear pairing function mapping elements from group G to group G_T, i.e. $e : G \times G \to G_T$. The bilinear pairing function e requires the following properties:

1. Bilinearity: $e(aP, bP) = e(P, P)^{ab}$.
2. Non-degeneracy: $e(P, P) \neq 1$
3. e is efficiently computable.

2.3 The Cha-Cheon IBS Scheme

An IBS scheme consists of 4 stages: Setup, Extract, Sign and Verify. Following this model, the Cha-Cheon IBS scheme is defined in detail as follows:

1. **Setup**: Choose a prime generator $P \overset{\$}{\leftarrow} G$ and a master secret key $s \overset{\$}{\leftarrow} Z_q$, then set $P_{pub} = sP$. Select hash functions $H_1 : \{0,1\}^* \times G \to Z_q$ and $H_2 : \{0,1\}^* \to G$. Lastly establish the pairing function $e : G \times G \to G_T$. Publish the system parameters as $(G_1, G_T, q, e, P, P_{pub}, H_1, H_2)$ and keep s secret.

2. **Extract**: Given an identity ID (e.g. email address, domain), calculate $Q = H_2(ID)$ and $D_{ID} = sQ$. Q_{ID} plays the role of the associated public key whereas D_{ID} is the user secret key.
3. **Sign**: Given a message m and user secret key D_{ID} as input, choose a random number $r \overset{\$}{\leftarrow} Z_q$, compute $U = rQ_{ID}, h = H_1(m, U)$ and $V = (r + h)D_{ID}$. The signature is generated as $\sigma = (U, V)$.
4. **Verify**: To verify the signature $\sigma = (U, V)$ of a message m with identity ID, check the validity of the tuple $(P, P_{pub}, U + hQ_{ID}, V$ by resolving $e(P, V) = e(P_{pub}, U + hQ_{ID})$.

For correctness, the following check equation should hold:

$$\begin{aligned} e(P_{pub}, U + hQ_{ID}) &= e(sP, rQ_{ID} + hQ_{ID}) && (1)\\ &= e(P, s(r + h)Q_{ID}) && (2)\\ &= e(P, (r + h)D_{ID}) && (3)\\ &= e(P, V) && (4) \end{aligned}$$

3 ECC ELGAMAL SCHEME

3.1 The ECC ElGamal Scheme

For transfering the user secret key securely from server to client, we use the ECC version of ElGamal public key encryption (PKE). The scheme consists of 3 stages: Key Generation stage, Encrypt stage, and Decrypt stage. In detail, the scheme is defined as follows:

1. **Key Generation**: Choose a prime generator based on the order of N, $P \leftarrow E_N$ and a secret key $x \leftarrow Z_q$, then set $Y = xP$. Publish the public key as (Y, P) and keep x as a secret.
2. **Encryption**: Generate a random $k \leftarrow Z_q$. Set $C_1 = kP$ and $C_2 = kY$. Then, map the message m as points $P_m = f(m)$. The ciphertext would be published as $(C_1, C_2 + P_m)$.

3. **Decryption**: The ciphertext is obtained as (C, D). Set $C' = xC$. The message points can be obtained by $P_m = D - C' = (k(xP) + P_m) - (x(kP))$. Then, the original message can be obtained by inversing the function $m = f^{-1}(P_m)$.

4 METHODOLOGY

In this section we describe the methodology of applying Tan et al. (2010)'s library in iOS mobile application development. Objective-C has been the native programming language for developing an iOS application ever since Apple acquired NeXT in 1996 (Cox and Love, 2015). Long before Apple gained its ground of market share, it has always been providing support for Java, for example the ”Java Bridge” which is the binding between Java and iOS's native application programming interface (API) known as Cocoa.

However, the unpopularity of ”Java bridge” among the Cocoa developers and incompatibility of Cocoa's key features with Java forced Apple to officially deprecate their support for ”Java bridge” in 2005 (Apple, 2015). Thus, all the new features Cocoa introduced later than Mac OS X version 10.4 were not available for Cocoa-Java programming interface. In 2014, another programming language developed by Apple known as Swift was introduced to replace the Objective-C language.

To apply the libraries introduced by Tan et al. (2010), translation from Java to Objective-C is a must. Hence, a different approach other than ”Java Bridge” must be made to link the libraries for both Java and Objective-C such as using third party translations tools. Besides that, the mobile application development also requires the knowledge of using the development tools provided by Apple.

4.1 Translation of Java libraries using J2ObjC

Java source code can be translated using a tool called J2ObjC which is an open-source command-line tool developed by Google (Ball, 2015). This tool allows developers to implement their Java source to be part of iOS application's build by translating them to Objective-C. It supports most of the Java language and runtime features such as generic types, threads, reflection and exceptions. The goal of this tool is shared the application and data models written in Java to other platforms such as GWT web apps, Android, and iOS applications.

However, platform-independent user-interface (UI) toolkit is not provided, so developers are required to write their own iOS UI code using the related Software Development Kit (SDK) provided by the platform they working on. In this work, we designed our UI using Xcode, a native iOS Application Development Tool (ADT) with code written in either Objective-C or Swift 2.1. The requirements of J2ObjC were stated in (Ball, 2015), where users need a Mac workstation installed with Mac OS X version of 10.9 or higher. Besides that, users are also required to install Java Development Kit (JDK) with version 1.7 or higher on their Mac (Oracle, 2015). Lastly, users need to install the Xcode version 6 or higher on their Mac.

Before we get started, we need to download and unzip the distribution provided by Ball (2015), the version of J2ObjC we used for this work was 0.9.8.2.1. After we tested the translation with some example code we wrote ourselves following the guide located here, we started working on the translation of pairing libraries.

There are a total of four libraries introduced by Tan et al. (2010): *CpxBiginteger*, *Line*, *Point* and *Curve*. Our first attempt at direct translation did not succeed due to the library's dependencies of data types from each other. Hence, our following attempts were done by merging all four libraries into a single one and the translation was successful by following the J2ObjC guide. Two files were generated in *.m* and *.h* format respectively. *.h* file is the header file while *.m* file contains all the method declarations. Both are needed for our implementation of the IBS scheme and also the PKE scheme using Xcode.

We used some exclusive Java objects (data types) in our original implementation in Java platform such as the *BigInteger* from *java.math* and *SecureRandom* from *java.security*. To reduce the complexities of our implementation, we decided to combine our IBS scheme written in Java with the merged library instead of finding third-party alternative Objective-C libraries that can provide similar functions. Now, we are only required to call the methods and pass in the desired input to test our IBS scheme and also the PKE scheme.

The size of the translated library is much larger compared to the original Java libraries due to syntax differences between two programming languages. However, no additional editing is needed to run our code.

4.2 Mobile application development using Xcode

Xcode is an integrated development environment (IDE) developed by Apple as a native software development tools for OS X and iOS. First of all, before we start to use our translated library, we need to link the J2ObjC to Xcode following the guide provided by Cheah et al. (2015), details as below:

1. Replace the J2ObjC's distribution directory to *$distribution-path*.
2. Select the project target in Xcode and click on the Build Rules tab.
3. Click the Add Build Rule button located at the bottom right of the panel.
4. Select "Java source files" in the new rule's Process option.
5. Add the following script in the Custom script text box:
    (a) *$distribution-path/j2objc -d ${DERIVED_FILES_DIR}*
    (b) *-sourcepath ${PROJECT_DIR}/$source-root*
    (c) *–no-package-directories ${INPUT_FILE_PATH};*
6. Click the + button in the Output Files panel, and add *${DERIVED_FILES_DIR}/${INPUT_FILE_BASE}.h*
7. Click the + button again, and add *$DERIVED_FILES_DIR/${INPUT_FILE_BASE}.m*
8. Update the Build Settings:
    (a) In User Header Search Paths, add
        i. *$distribution-path/include and*
        ii. *${DERIVED_FILES_DIR}*

   (b) In Library Search Paths, add *$distribution-path/lib*
   (c) In Other Linker Flags, add *-ljre_emul*

9. Select Build Phases tab, open the Link phase and add:
   (a) The *libz.dylib* library
   (b) The Security Framework
   (c) The *libicucore.dylib* library [1]

10. Add the *.m* and *.h* files generated with J2ObjC to Xcode's project directories

The updated build settings for our project allows Xcode to apply the translated libraries by simply importing the *.h* header file. The JRE emulation library from J2ObjC emulates a subset of Java runtime libraries and is essential for the translated library to work properly.

Since our main goal is to test the simulation of IBS scheme by implementing it as an iOS mobile application, our UI design is relatively simple as shown in the left part of Figure 1. The right part of Figure 1 illustrates the completed UI. First of all, four methods were implemented to represent the four stages of the BLS-IBS scheme respectively. Execution of each method is represented by two buttons. One of the buttons executes the respective method a single time and shows the output in string format including the time taken for running a single stage. Another button will execute the respective method multiple times to calculate the average time taken. Figure 3 shows some blocks of code that represent the functions of the buttons of our application. The flow of the program is following the IBS scheme discussed in Section 2.

After the implementation of the IBS simulation, we tested the functionality of the Elgamal PKE. Similar to the IBS, we also used a simple UI design, as shown in the left side of Figure 2. The right side of Figure 2 illustrates the completed UI of the PKE. The two buttons as shown were implemented to

[1]This is an extra step to link the JRE emulation library in step 8 above , thereby solving the linker error mentioned by Cheah et al. (2015)

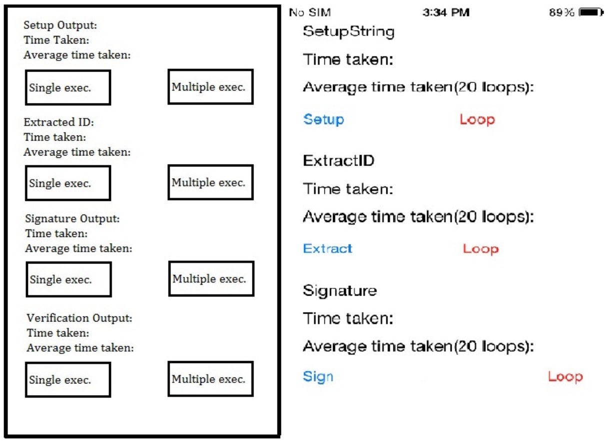


Figure 1: Mobile application UI design (left) and screenshot of iOS mobile application(right) for IBS

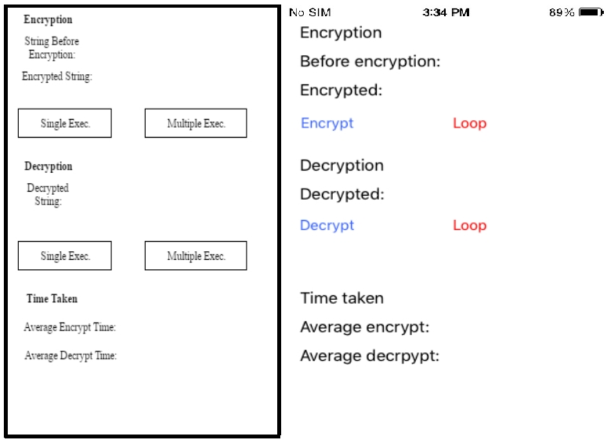


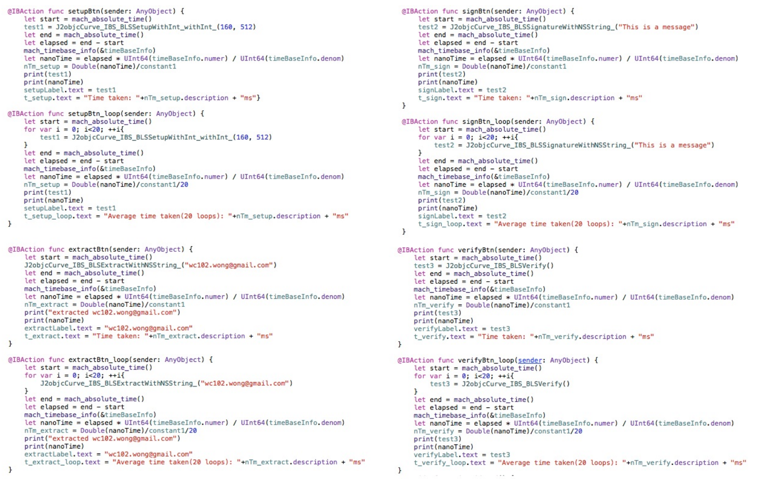
Figure 2: Mobile application UI design (left) and screenshot of iOS mobile application(right) for PKE

represent the Encrypt and Decrypt stages of the ECC ElGamal scheme respectively.

The setup stage based on the IBS was used to generate the point message as the string to be encrypted. The Encrypt button would run the Setup stage based on the IBS and then generate the original string to be encrypted, while generating the public keys and secret keys as well. The Decrypt button would then run the decryption of the ciphertext to show the original encrypted string. The time taken for the Encrypt excludes the Key Generation stage since it is similar to the IBS scheme. Figure 4 shows some blocks of code that represent the functions of the buttons of our application.

5 SIMULATION RESULTS

In this section, we present the simulation results for the BLS-IBS and ECC ElGamal scheme using translated ECC pairing libraries.

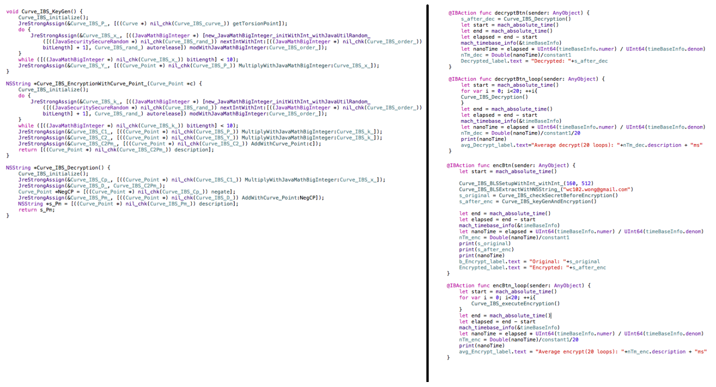


```
@IBAction func setupBtn(sender: AnyObject) {
    let start = mach_absolute_time()
    test1 = J2objcCurve_IBS_BLSSetupWithInt_withInt_(160, 512)
    let end = mach_absolute_time()
    let elapsed = end - start
    mach_timebase_info(&timeBaseInfo)
    let nanoTime = elapsed * UInt64(timeBaseInfo.numer) / UInt64(timeBaseInfo.denom)
    nTm_setup = Double(nanoTime)/constant1
    print(test1)
    print(nanoTime)
    setupLabel.text = test1
    t_setup.text = "Time taken: "+nTm_setup.description + "ms"}

@IBAction func setupBtn_loop(sender: AnyObject) {
    let start = mach_absolute_time()
    for var i = 0; i<20; ++i{
        test1 = J2objcCurve_IBS_BLSSetupWithInt_withInt_(160, 512)
    }
    let end = mach_absolute_time()
    let elapsed = end - start
    mach_timebase_info(&timeBaseInfo)
    let nanoTime = elapsed * UInt64(timeBaseInfo.numer) / UInt64(timeBaseInfo.denom)
    nTm_setup = Double(nanoTime)/constant1/20
    print(test1)
    print(nanoTime)
    setupLabel.text = test1
    t_setup_loop.text = "Average time taken(20 loops): "+nTm_setup.description + "ms"
}

@IBAction func extractBtn(sender: AnyObject) {
    let start = mach_absolute_time()
    J2objcCurve_IBS_BLSExtractWithNSString_("wc102.wong@gmail.com")
    let end = mach_absolute_time()
    let elapsed = end - start
    mach_timebase_info(&timeBaseInfo)
    let nanoTime = elapsed * UInt64(timeBaseInfo.numer) / UInt64(timeBaseInfo.denom)
    nTm_extract = Double(nanoTime)/constant1
    print("extracted wc102.wong@gmail.com")
    print(nanoTime)
    extractLabel.text = "wc102.wong@gmail.com"
    t_extract.text = "Time taken: "+nTm_extract.description + "ms"
}

@IBAction func extractBtn_loop(sender: AnyObject) {
    let start = mach_absolute_time()
    for var i = 0; i<20; ++i{
        J2objcCurve_IBS_BLSExtractWithNSString_("wc102.wong@gmail.com")
    }
    let end = mach_absolute_time()
    let elapsed = end - start
    mach_timebase_info(&timeBaseInfo)
    let nanoTime = elapsed * UInt64(timeBaseInfo.numer) / UInt64(timeBaseInfo.denom)
    nTm_extract = Double(nanoTime)/constant1/20
    print("extracted wc102.wong@gmail.com")
    print(nanoTime)
    extractLabel.text = "wc102.wong@gmail.com"
    t_extract_loop.text = "Average time taken(20 loops): "+nTm_extract.description + "ms"
}
```

```
@IBAction func signBtn(sender: AnyObject) {
    let start = mach_absolute_time()
    test2 = J2objcCurve_IBS_BLSSignatureWithNSString_("This is a message")
    let end = mach_absolute_time()
    let elapsed = end - start
    mach_timebase_info(&timeBaseInfo)
    let nanoTime = elapsed * UInt64(timeBaseInfo.numer) / UInt64(timeBaseInfo.denom)
    nTm_sign = Double(nanoTime)/constant1
    print(test2)
    print(nanoTime)
    signLabel.text = test2
    t_sign.text = "Time taken: "+nTm_sign.description + "ms"
}

@IBAction func signBtn_loop(sender: AnyObject) {
    let start = mach_absolute_time()
    for var i = 0; i<20; ++i{
        test2 = J2objcCurve_IBS_BLSSignatureWithNSString_("This is a message")
    }
    let end = mach_absolute_time()
    let elapsed = end - start
    mach_timebase_info(&timeBaseInfo)
    let nanoTime = elapsed * UInt64(timeBaseInfo.numer) / UInt64(timeBaseInfo.denom)
    nTm_sign = Double(nanoTime)/constant1/20
    print(test2)
    print(nanoTime)
    signLabel.text = test2
    t_sign_loop.text = "Average time taken(20 loops): "+nTm_sign.description + "ms"
}

@IBAction func verifyBtn(sender: AnyObject) {
    let start = mach_absolute_time()
    test3 = J2objcCurve_IBS_BLSVerify()
    let end = mach_absolute_time()
    let elapsed = end - start
    mach_timebase_info(&timeBaseInfo)
    let nanoTime = elapsed * UInt64(timeBaseInfo.numer) / UInt64(timeBaseInfo.denom)
    nTm_verify = Double(nanoTime)/constant1
    print(test3)
    print(nanoTime)
    verifyLabel.text = test3
    t_verify.text = "Time taken: "+nTm_verify.description + "ms"
}

@IBAction func verifyBtn_loop(sender: AnyObject) {
    let start = mach_absolute_time()
    for var i = 0; i<20; ++i{
        test3 = J2objcCurve_IBS_BLSVerify()
    }
    let end = mach_absolute_time()
    let elapsed = end - start
    mach_timebase_info(&timeBaseInfo)
    let nanoTime = elapsed * UInt64(timeBaseInfo.numer) / UInt64(timeBaseInfo.denom)
    nTm_verify = Double(nanoTime)/constant1/20
    print(test3)
    print(nanoTime)
    verifyLabel.text = test3
    t_verify_loop.text = "Average time taken(20 loops): "+nTm_verify.description + "ms"
}
```

Figure 3: Blocks of codes for IBS button functions.

*J2objcCurve_IBS_BLSSetupWithInt_withInt_* is the method called to run the Setup stage for IBS scheme. The passed arguments are 160 and 512 to represent the order and modulus bits of our elliptic curve respectively. This is equivalent to 1024-bit of RSA or discrete logarithms (DLOG) security.



```
void Curve_IBS_KeyGen() {
    Curve_IBS_initialize();
    JreStrongAssign(&Curve_IBS_P_, [((Curve *) nil_chk(Curve_IBS_curve_)) getTorsionPoint]);
    do {
        JreStrongAssign(&Curve_IBS_x_, [((JavaMathBigInteger *) [new_JavaMathBigInteger_initWithInt_withJavaUtilRandom_
            ([((JavaSecuritySecureRandom *) nil_chk(Curve_IBS_rand_)) nextIntWithInt:[((JavaMathBigInteger *) nil_chk(Curve_IBS_order_))
            bitLength] + 1], Curve_IBS_rand_) autorelease]) modWithJavaMathBigInteger:Curve_IBS_order_]);
    }
    while ([((JavaMathBigInteger *) nil_chk(Curve_IBS_x_)) bitLength] < 10);
    JreStrongAssign(&Curve_IBS_Y_, [((Curve_Point *) nil_chk(Curve_IBS_P_)) MultiplyWithJavaMathBigInteger:Curve_IBS_x_]);
}

NSString *Curve_IBS_EncryptionWithCurve_Point_(Curve_Point *c) {
    Curve_IBS_initialize();
    do {
        JreStrongAssign(&Curve_IBS_k_, [((JavaMathBigInteger *) [new_JavaMathBigInteger_initWithInt_withJavaUtilRandom_
            ([((JavaSecuritySecureRandom *) nil_chk(Curve_IBS_rand_)) nextIntWithInt:[((JavaMathBigInteger *) nil_chk(Curve_IBS_order_))
            bitLength] + 1], Curve_IBS_rand_) autorelease]) modWithJavaMathBigInteger:Curve_IBS_order_]);
    }
    while ([((JavaMathBigInteger *) nil_chk(Curve_IBS_k_)) bitLength] < 10);
    JreStrongAssign(&Curve_IBS_C1_, [((Curve_Point *) nil_chk(Curve_IBS_P_)) MultiplyWithJavaMathBigInteger:Curve_IBS_k_]);
    JreStrongAssign(&Curve_IBS_C2_, [((Curve_Point *) nil_chk(Curve_IBS_Y_)) MultiplyWithJavaMathBigInteger:Curve_IBS_k_]);
    JreStrongAssign(&Curve_IBS_C2Pm_, [((Curve_Point *) nil_chk(Curve_IBS_C2_)) AddWithCurve_Point:c]);
    return [((Curve_Point *) nil_chk(Curve_IBS_C2Pm_)) description];
}

NSString *Curve_IBS_Decryption() {
    Curve_IBS_initialize();
    JreStrongAssign(&Curve_IBS_Cp_, [((Curve_Point *) nil_chk(Curve_IBS_C1_)) MultiplyWithJavaMathBigInteger:Curve_IBS_x_]);
    JreStrongAssign(&Curve_IBS_D_, Curve_IBS_C2Pm_);
    Curve_Point *NegCP = [((Curve_Point *) nil_chk(Curve_IBS_Cp_)) negate];
    JreStrongAssign(&Curve_IBS_Pm_, [((Curve_Point *) nil_chk(Curve_IBS_D_)) AddWithCurve_Point:NegCP]);
    NSString *s_Pm = [((Curve_Point *) nil_chk(Curve_IBS_Pm_)) description];
    return s_Pm;
}
```

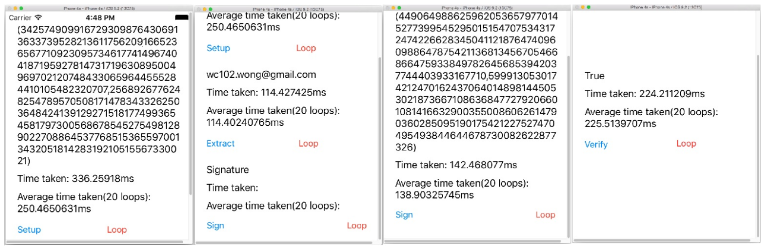
```
@IBAction func decryptBtn(sender: AnyObject) {
    s_after_dec = Curve_IBS_Decryption()
    let start = mach_absolute_time()
    let end = mach_absolute_time()
    let elapsed = end - start
    mach_timebase_info(&timeBaseInfo)
    let nanoTime = elapsed * UInt64(timeBaseInfo.numer) / UInt64(timeBaseInfo.denom)
    nTm_dec = Double(nanoTime)/constant1
    Decrypted_label.text = "Decrypted: "+s_after_dec
}

@IBAction func decryptBtn_loop(sender: AnyObject) {
    let start = mach_absolute_time()
    for var i = 0; i<20; ++i{
    Curve_IBS_Decryption()
    }
    let end = mach_absolute_time()
    let elapsed = end - start
    mach_timebase_info(&timeBaseInfo)
    let nanoTime = elapsed * UInt64(timeBaseInfo.numer) / UInt64(timeBaseInfo.denom)
    nTm_dec = Double(nanoTime)/constant1/20
    print(nanoTime)
    avg_Decrypt_label.text="Average decrypt(20 loops): "+nTm_dec.description + "ms"
}

@IBAction func encBtn(sender: AnyObject) {
    let start = mach_absolute_time()

    Curve_IBS_BLSSetupWithInt_withInt_(160, 512)
    Curve_IBS_BLSExtractWithNSString_("wc102.wong@gmail.com")
    s_original = Curve_IBS_checkSecretBeforeEncryption()
    s_after_enc = Curve_IBS_keyGenAndEncryption()

    let end = mach_absolute_time()
    let elapsed = end - start
    mach_timebase_info(&timeBaseInfo)
    let nanoTime = elapsed * UInt64(timeBaseInfo.numer) / UInt64(timeBaseInfo.denom)
    nTm_enc = Double(nanoTime)/constant1
    print(s_original)
    print(s_after_enc)
    print(nanoTime)
    b_Encrypt_label.text = "Original: "+s_original
    Encrypted_label.text = "Encrypted: "+s_after_enc
}

@IBAction func encBtn_loop(sender: AnyObject) {
    let start = mach_absolute_time()
    for var i = 0; i<20; ++i{
        Curve_IBS_executeEncryption()
    }
    let end = mach_absolute_time()
    let elapsed = end - start
    mach_timebase_info(&timeBaseInfo)
    let nanoTime = elapsed * UInt64(timeBaseInfo.numer) / UInt64(timeBaseInfo.denom)
    nTm_enc = Double(nanoTime)/constant1/20
    print(nanoTime)
    avg_Encrypt_label.text = "Average encrypt(20 loops): "+nTm_enc.description + "ms"
}
```

Figure 4: Blocks of codes for PKE button functions.

The method's running time is calculated by calling *mach_absolute_time()*. Results were taken in nanoseconds but converted to milliseconds afterwards by dividing the double type constant of 1,000,000.0. The loop button functions in a similar way except it is calling the methods 20 times to calculate the average time taken.

Other stages of the IBS scheme were implemented in a similar way as shown in Figure 3. For Extraction stage, we use an email as the ID input whereas some text that represent the message input were passed in for Signing stage. Inputs of both stages were passed in as strings. For Verification stage, the verify button will call the *J2objcCurve_IBS_BLSVerify()* method which returns a string of "True" or "False" depending on the validity of the tuple as explained in Section 2.

The PKE scheme also uses the same Setup stage that was used in the IBS scheme for the key generation. *Curve_IBS_KeyGen_* is called to generate the public and private keys. The arguments 160 and 512 are also passed to represent a 1024-bit of RSA security. The remaining stages of the PKE scheme were implemented as shown in Figure 4. For the Encrypt stage, the encrypt button would call the method to return the encrypted message point. For Decrypt stage, the decrypt button would then decrypt the encrypted point to return back the original message point.



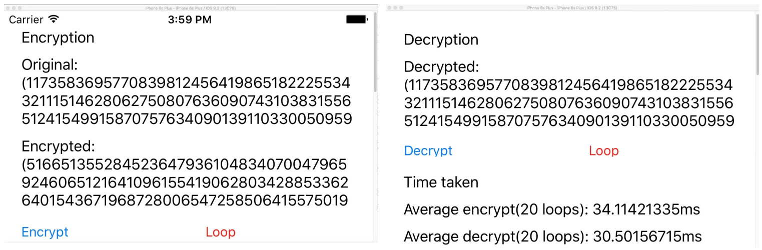
```
Carrier        4:48 PM
(342574909916729309876430691
3633739528213611756209166523
6567710923095734617741496740
4187195927814731719630895004
969702120748433065964455528
4410105482320707,256892677624
8254789570508171478343326250
3648424139129271518177499365
4581797300568678545275498128
9022708864537768515365597001
3432051814283192105155673300
21)
Time taken: 336.25918ms
Average time taken(20 loops):
250.4650631ms
Setup          Loop
```

```
Average time taken(20 loops):
250.4650631ms
Setup          Loop

wc102.wong@gmail.com
Time taken: 114.427425ms
Average time taken(20 loops):
114.40240765ms
Extract        Loop

Signature
Time taken:
Average time taken(20 loops):
Sign           Loop
```

```
(449064988625962053657977014
5277399545295015154707534317
2474226628345041121876474096
0988647875421136813456705466
866475933849782645685394203
7744403933167710,599913053017
4212470162437064014898144505
3021873667108636847727920660
1081416632900355008606261479
0360285095190175421227527470
495493844644678730082622877
326)
Time taken: 142.468077ms
Average time taken(20 loops):
138.90325745ms
Sign           Loop
```

```
True
Time taken: 224.211209ms
Average time taken(20 loops):
225.5139707ms
Verify         Loop
```

Figure 5: IBS result screenshots taken from iPhone 4s(iOS 8.4).



```
Carrier        3:59 PM
Encryption
Original:
(1173583695770839812456419865182225534
32111514628062750807636090743103831556
51241549915870757634090139110330050959
Encrypted:
(516651355284523647936104834070047965
9246065121641096155419062803428853362
6401543671968728006547258506415575019
Encrypt        Loop
```

```
Decryption
Decrypted:
(1173583695770839812456419865182225534
32111514628062750807636090743103831556
51241549915870757634090139110330050959
Decrypt        Loop
Time taken
Average encrypt(20 loops): 34.11421335ms
Average decrypt(20 loops): 30.50156715ms
```

Figure 6: PKE result screenshots taken from iPhone 6s(iOS 9.2).

The running time of each method is calculated using *mach_absolute_time()*. Similar to the IBS, the results were taken in nanoseconds but were converted to milliseconds. The loop button calls the methods 20 times to calculate the average time taken for each process.

This simulation experiment was conducted on multiple iOS devices with different hardware specifications and also on the Xcode simulator from the Mac Mini we were using. The results of the experiment are listed in Table 1 and Table 2. Figure 5 and Figure 6 consists of screenshots taken from various iPhone devices after running the experiment.

The translated library has been proven to have the same functionality as Tan et al. (2010)'s library. We also managed to obtain efficient results from the implementation as iOS mobile application. Our results also show that the simulator on Mac has the best results whilst the efficiency on mobile devices is determined by their hardware specifications. The upgrade on OS version only

Devices	OS Version	Setup(ms)	Extract(ms)	Sign(ms)	Verify(ms)
Simulator(Mac)	OS X 10.11	250.4650631	114.4024077	138.9032575	225.5139707
iPhone 4s	iOS 8.4	2760.469677	1059.206292	1343.33111	2261.52975
iPhone 4s	iOS 9.2.1	2569.235804	1001.158285	1228.748246	2224.738048
iPhone 5s	iOS 9.2.1	998.4872354	301.9295063	313.1813958	588.9880083
iPhone 6	iOS 9.2.1	469.7018542	165.122175	186.0620521	353.6549333

Table 1: Runtime Results for Cha-Cheon IBS Scheme.

Devices	OS Version	Encrypt(ms)	Decrypt(ms)
Simulator(Mac)	OS X 10.11	30.27794601	27.02684978
iPhone 4s	iOS 8.4	250.723468	231.157386
iPhone 4s	iOS 9.2.1	135.625709	127.8706345
iPhone 5s	iOS 9.2.1	53.76583235	51.97524613
iPhone 6	iOS 9.2.1	34.11421335	30.50156715

Table 2: Runtime Results for ECC ElGamal PKE.

brings a slight improvement in performance.

6 CONCLUSION

In this work, we successfully developed a working mobile application simulator for Cha-Cheon BLS-IBS and ECC ElGamal schemes by using the translated Java based libraries from Tan et al. (2010). The translated library was proved functional by running the Setup, Extract, Sign and Verification algorithms with reasonable running times simulated across multiple devices. Our future work will extending the simulator to a client-server architecture.

ACKNOWLEDGMENTS

The authors are thank the Ministry of Education Malaysia for financial assistance under The Fundamental Research Project Scheme (No.: FRGS/2/2013/ICT07/MMU/03/5)
.

REFERENCES

Apple (2015). *Cocoa API - Implementations and Bindings*. `https://en.wikipedia.org/wiki/Cocoa_(API)`.

Ball, T. (2015). *J2ObjC - Java to Objective-C Translator and Run-Time*. `https://github.com/google/j2objc/releases`.

Cha, J.-C. and Cheon, J.-H. (2003). An identity-based signature from gap diffie-hellman groups. In *ICCSA 2010*, volume 2567, pages 18–30.

Cheah, Z.-Y., Teh, T.-Y., Lee, Y.-S., and Chin, J.-J. (2015). Simulation of a pairing-based identity-based identification scheme in ios. In *ICSIPA 2015*, Pullman Hotel, Bangsar, Malaysia.

Cox, B. and Love, T. (2015). *Objective-C - Apple Development and Swift*. `https://en.wikipedia.org/wiki/Objective-C`.

Elgamal, T. (1985). A public key cryptosystem and a signature scheme based on discrete logarithms. In *Advances in cryptology*, volume 196, pages 10–18.

International-Data-Corporation (2015). *International Data Corporation - Smartphone OS Market Share for Q2 2015*. `http://www.idc.com/prodserv/smartphone-os-market-share.jsp`.

Oracle (2015). *Oracle - Java SE Downloads*. `http://www.oracle.com/technetwork/java/javase/downloads/index.html`.

Pew-Research-Center (2015). *Technology Device Ownership Statistic*. `http://www.pewinternet.org/2015/10/29/technology-device-ownership-2015/`.

Tan, S.-Y., Heng, S.-H., and Goi, B.-M. (2010). Java implementation for pairing-based cryptosystems. In *ICCSA 2010*, volume 6019, pages 188–198.