Chapter 7:
Enhancement
The theory of my developed applications can be a great pattern that by implement it can get a new feature on these applications.

In general, we can have different types of maintenance for each application that can make it for interactive.

One of the features that we can add to all of developed applications is make it, as a trial application as well as full application. For that we should consider to make it along with the secure license for and also consider the business part for the buyer companies, and manufacturer.
Another part that is needed for an application like, Omid – Hit Brick ver1.0, is to add more level of games to it, which should consider the more number of bricks in each level and as well as Omid – Air Force Drop ver1.0 in each level make speed up and for this purpose, there is no need to change all the codes and classes part, just we should make the different classes which are the hierarchy of the different classes that can have all the specification of other classes along with the new feature that is the new level and make it more interactive and encourage different player to come in this competition once at least.

Another feature, that I think more about we implement it, is to have a special part of a server on the specific IP and try to get the best game score result in every month, and to become more active and interact by different users, monthly give a good prize or award to the first most scored player, that it will be more interest in the game markets, and in the society at least, who interest in game playing will try it once.

So, one another important part which can be consider in my main application, is to have a secure connection between our application and our access to the specify server, make it more secure along with using digital signature which this concept and its feature illustrates as follow.

Make it more secure:

XML digital signature technology can help you implement lightweight and flexible security solutions for wireless Web services applications. By walks through the digital signature APIs of the secure cryptography package, providing examples in the context of secure XML messaging between a J2ME/MIDP wireless front end and a JSP page back end.

Java-based Web services and wireless Java development were the two most prominent topics at JavaOne 2002. They represent the future back- and front-end Java technologies in a world of pervasive computing.
Web services are loosely coupled; interoperable software components based on standard XML communication protocols. The use of Web services enables a vendor to offer services in a specific market where its core competency lies. Customers can then choose to purchase services from multiple vendors, according to their various needs. This convenience means that Web services are perfectly suited to serve wireless front ends. The convenience and dynamic nature of wireless information devices allow mobile users to take advantage of modularized and dynamically re-configurable back-end services.

The Java platform can play several important roles in wireless Web services application development. On the wireless end, Java 2 Micro Edition (J2ME) offers cross-device compatibility, advanced language features, and extensive libraries to all wireless devices from cell phones to complex home wireless information appliances. A key component of J2ME is the Mobile Information Device Profile (MIDP), which defines Java APIs and runtime environments on cell phones and low-end PDAs. Due to the large number of low-end devices, MIDP is expected to become widely deployed in the future.

From the Web services end, Java 2 Enterprise Edition (J2EE) already has all the necessary APIs and libraries to handle Web services XML messages. Core J2EE functions implemented in EJB technology, the JDBC API, and the RMI API can easily be made available to the outside world through Web services interfaces or gateways. To bring these features together and enable wireless Web services applications, the J2ME Web services specification has also been proposed and is currently in the Java Community Process (JSR 172).

Although Java-based wireless Web services have a bright future in the world of pervasive mobile commerce, the current technology is not yet mature. Security is among the remaining issues yet to be resolved. Wireless communications are easy targets for air wave interception, and wireless devices rarely have the computing power to support strong encryption of all communication data. Moreover, on the back end, Web services run outside corporate firewalls and interact with each other using open messaging protocols. Wireless Web services are likewise vulnerable targets for various cracking attacks. Well developed point-to-point security technologies such as SSL/TLS and HTTPS are not suitable for the multiple vendor, multiple intermediary Web services network topography -- the focus needs to be on securing the contents themselves rather than the connections over which they travel. Despite the new
challenges. Web services themselves, however, can be used to enhance mobile commerce security. Emerging Web services security specifications enable you to use Web services as security utilities.

In the paragraphs below, I'll discuss a commonly used security technique: digital signature. I will show you how digital signatures are used in XML messages to guarantee end-to-end data integrity. I'll cover some examples that illustrate how to implement XML digital signatures using the popular J2ME/MIDP platform on the wireless end and JavaServer Pages (JSP) technology on the back end. Finally, I will discuss performance issues and the feasibility of using digital signatures on current MIDP devices.

Let's suppose that you are a stock trader and that you use a cell phone to track stock price changes when you are not on the trading floor. In the middle of your commute to work, your phone alerts you that the price of a stock you have been monitoring has dropped below a threshold. Now, should you buy it based on this tip and take advantage of the low price? Before you take any action, you want to be absolutely sure that the tip itself is authentic. If a competitor could intercept and change the message (for example, change the stock symbol), he could lure you into buying the wrong stock and dump his high-priced equity to you. How do you know that your message has not been tampered with en route from your monitoring service to your phone?

Indeed, data integrity is one of the most important aspects of communication security. Physically secure networks are very expensive and do not cover a wide geographic area. If your business has to rely on the Internet to communicate, you have to face the fact that the Internet offers little security by itself. Internet data packets have to travel through multiple routers and hosts that are controlled by neither party in the conversation before they reach their destinations. Data communications are especially vulnerable over the wireless Internet. The tools that come to our rescue are the Public Key Infrastructure (PKI) and digital signature. (As with MIDP programming, digital signature is beyond the scope of this part; in a nutshell, in a PKI digital signature scheme, each party has two cryptography keys: the public key is available to anyone and the private key is kept secret to oneself. A message encrypted by the private key can only be correctly decrypted by the corresponding public key. When a sender sends out a message, he can accompany the message with a private key.
encrypted version of the same message, as well as his public key. The recipient decrypts the
encrypted version using the sender's public key. If it matches the clear text message, the
recipient can know for sure that the message is authentic. The private key encrypted version
of the message serves as an integrity verification token and it is called a "digital signature."
Because the original message might be quite long, and public key algorithms to generate and
verify digital signatures are resource intensive, the sender normally calculates a short version
of the original message called the "digest" and digitally signs only this version. The digest is
a fixed-length, one-way hash of any-length input message; its calculation is very fast. The
recipient first verifies that the message received produces the correct digest. If the digest does
not match, the message is rejected before any public key algorithm is performed. That could
help prevent a clogging attack, in which the attacker overwhelms a server's computational
resources by flooding it with fake public key requests.
In most practical applications, public keys themselves are digitally signed by trusted
authorities, becoming "digital certificates" to validate the sender's identification. Digital
certificate handling is beyond the scope of this part, however, so in the following examples, I
will assume that the senders are trusted and use unsigned public keys to illustrate the
approaches.

IBM alphaWorks develops a Java package called XML Security Suite, which supports the
latest XML digital signature specification. JSR 105 is a community effort to standardize a set
of Java APIs to process XML digital signatures. However, they only work for Java 2
Standard Edition (J2SE), which means that you can use XML Security Suite or JSR 105 Java
XML digital signature APIs on the server side, but not on the MIDP wireless device side.
To handle XML digital signatures, the wireless devices being used need to support the
following functions:

- Read and write data from/to an XML document. In our example MIDP
  applications, XML documents and elements are parsed into Java objects by the kXML
  parser. In addition to kXML, there are several other MIDP XML parsers available
  under different license terms.
Sign the message and verify the signature. These functions require a cryptography API that is not part of the current MIDP 1.0 specification.

Bouncy Castle is an open source, lightweight cryptography package for the Java platform. It supports a large number of cryptography algorithms and provides an implementation for JCE 1.2.1. Because Bouncy Castle is designed to be lightweight, it can run from J2SE 1.4 to J2ME (including MIDP) platforms. It is the only complete cryptography package that runs on MIDP.

However powerful, there is one major problem with the Bouncy Castle package: the lack of documentation. Online documentation is non-existent and its JavaDoc is not very well written. Similar to many other advanced cryptography packages, the Bouncy Castle package uses type polymorphism extensively to separate general concepts from implementing algorithms. It's hard for beginners to decipher the relationships between classes and the correct types for methods arguments and return values. Often, the developer has to peek into the source code and test cases to investigate correct ways to do things. Clearly, a guide for the Bouncy Castle package is very much in order.

In the rest of this article, we will walk through the XML digital signature specification and the usage of several different Bouncy Castle key generators, encoding engines, digital signature singers, and a digest engine.

As I mentioned earlier, to improve performance and avoid clogging attacks, you actually sign the message digest rather than the message itself. Listing 1 illustrates how to compute an encoded digest from a piece of text message using the SHA1Digest digest engine.

```java
static public String getDigest( String msg ) throws Exception {
    SHA1Digest digEng = new SHA1Digest();
    byte[] msgBytes = msg.getBytes();
```
In the next part, we'll see how to sign and verify digital signatures using Bouncy Castle's DSA, ECC, and RSA signers. Those signers use different algorithms and different keys, and need different parameters. We'll also discuss how to embed security information (signatures, digests, and public keys) in XML documents. At the end, I'll compare the three signers and suggest future improvements.

The method DSASigUtil.generateKeys() generates key pairs. As I have discussed, this step is normally done offline by a central certificate authority, as shown in Listing 2:

```java
SecureRandom sr = new SecureRandom();

DSAParametersGenerator DSAParaGen = new DSAParametersGenerator();
DSAParaGen.init(1024, 80, sr);
DSAPara = DSAParaGen.generateParameters();
```

```java
// Get DSA key generation parameters.
DSAKeyGenerationParameters DSAKeyGenPara =
    new DSAKeyGenerationParameters(sr, DSAPara);
```

```java
// Generate keys.
```

```
// AnalysiS of wireless devices platform and professional developing in J2ME applications

DSAKeyPairGenerator DSAKeyPairGen = new DSAKeyPairGenerator();
DSAPublicKeyGen.init( DSAKeyGenPara );
AsymmetricCipherKeyPair keyPair = DSAKeyPairGen.generateKeyPair();

prvKey = (DSAPrivateKeyParameters) keyPair.getPrivate();
pubKey = (DSAPublicKeyParameters) keyPair.getPublic();

The generated public key is characterized by a parameter Y, and it is retrieved by the
pubKey.getY() method. Parameters G, P, and Q describe the model. The following methods
in class DSAUtil retrieve the model and key parameters, which are necessary to reconstruct
the public key object:

public static String getG() throws Exception {
    return (new String(Base64.encode(DSAPara.getG().toByteArray())));
}

public static String getP() throws Exception {
    return (new String(Base64.encode(DSAPara.getP().toByteArray())));
}

public static String getQ() throws Exception {
    return (new String(Base64.encode(DSAPara.getQ().toByteArray())));
}

public static String getY() throws Exception {
    return (new String(Base64.encode(pubKey.getY().toByteArray())));
}

Using the generated private key, the utility class DSASigUtil can get a two-part DSA
signature, R and S, from a digest:

static public String [] getSignature (String digest) throws Exception {

    // Sign
```
```
DSA signer = new DSA signer();
signer. init( true, privKey );
BigInteger [] sigarray = signer.generateSignature( digest.getBytes() );

String [] result = new String [2];
// Signature R
result[0] = new String( Base64.encode( sigArray[0].toByteArray() ));
// Signature S
result[1] = new String( Base64.encode( sigArray[1].toByteArray() ));

return result;
}
```

The server encodes the digest, signature, and key parameters into ASCII text form and embed the text in the XML digital signature format, as shown in Listing 5:

```
<SignedMsg>
  <msg>Hello World</msg>
  <Signature>
    <SignedInfo>
      <SignatureMethod
        Algorithm="http://www.sun.org/xmldsig#dsa-sha1" />
      <DigestValue>CkIVqM45QhvoJAIz8XYQLeEhGA=</DigestValue>
    </SignedInfo>
    <SignatureValue>
      <R>AHYtYlWgWyPDvUOgJoU+NH9KQc2z</R>
      <S>RwhqpppZx/jbMlYkh8dtYOlaH4=</S>
    </SignatureValue>
    <KeyInfo>
    </KeyInfo>
  </Signature>
</SignedMsg>
```
The verification MIDP application parses the digest, key parameters, and signature out of the XML document, reconstructs the public key, and uses the following method to validate the signature:

<DSAKeyValue>
<KeyInfo>
<SignedMsg>

The verification MIDP application parses the digest, key parameters, and signature out of the XML document, reconstructs the public key, and uses the following method to validate the signature:
static public boolean verify (String digest, 
        String sig_r, String sig_s, 
        String key_g, String key_p, 
        String key_q, String key_y) {

        BigInteger g = new BigInteger(Base64.decode(key_g));
        BigInteger p = new BigInteger(Base64.decode(key_p));
        BigInteger q = new BigInteger(Base64.decode(key_q));
        BigInteger y = new BigInteger(Base64.decode(key_y));
        BigInteger r = new BigInteger(Base64.decode(sig_r));
        BigInteger s = new BigInteger(Base64.decode(sig_s));

        DSAParameters DSAPara = new DSAParameters(p, q, g);
        DSAPublicKeyParameters DSAPubKeyPara = new DSAPublicKeyParameters(
                DSAPara);

        // Verify
        DSASigner signer = new DSASigner();
        signer.init(false, DSAPubKeyPara);
        boolean result = signer.verifySignature(digest.getBytes(), r, s);
        return result;
    }

The RSA algorithm only has one model parameter, Exponent:

private static BigInteger pubExp = new BigInteger("11", 16);
The `RSASigUtil.generateKeys()` method generates random key pairs using `Exponent`. Again, this step is normally done offline by a central certificate authority.

```java
SecureRandom sr = new SecureRandom();
RSAKeyGenerationParameters RSAKeyGenPara =
    new RSAKeyGenerationParameters(pubExp, sr, 1024, 80);
RSAPairGenerator RSAKeyPairGen = new RSAPairGenerator();
RSAKeyPairGen.init(RSAKeyGenPara);
AsymmetricCipherKeyPair keyPair = RSAKeyPairGen.generateKeyPair();

private = (RSAPrivateCrtKeyParameters) keyPair.getPrivate();
pubKey = (RSAKeyParameters) keyPair.getPublic();
```

The public key is characterized by a parameter, `Modulus`, and it is retrieved by the `pubKey.getModulus()` method. Listing 14 shows the methods in the `RSAUtil` class. These methods retrieve `Exponent` and `Modulus`, the model and key parameters, which are necessary to reconstruct the public key object.

```java
// Public key specific parameter.
public static String getMod() throws Exception {
    return (new String(Base64.encode(pubKey.getModulus().toByteArray())));
}

// General key parameter. pubExp is the same as pubKey.getExponent()
public static String getPubExp() throws Exception {
    return (new String(Base64.encode(pubExp.toByteArray())));
}
```

Using the generated private key, the utility class `RSASigUtil` can get a byte array RSA signature from a digest:

```java
static public String getSignature(String msg) throws Exception {
```
The server encodes the digest, signature, and key parameters into ASCII text form and embed the text in XML digital signature format:

```xml
<SignedMsg>
  <msg>Hello World</msg>
  <Signature>
    <SignedInfo>
      <SignatureMethod Algorithm="rsa-sha1" />  
      <DigestValue>Ck4YNd45Qvq3AId8XYQLvEhtA=</DigestValue>
    </SignedInfo>
    <SignatureValue>
      lHJ/rMfGK73WbahnGRrU6DfZbnfUobivemG9yfG
      WcQ3cUzsg2zdclWJX7w5h2Qyj5d8+hPj5ASC6r
      0jyj3Bd6pwjHkJ2kxWx7hS1f699/Qr/yaHsY6anaZ
      3oHeBfz2bEbrbYzeln3KeCvmvE22idN7kh88Do=
    </SignatureValue>
  </Signature>
</SignedMsg>
```
The verification MIDP application parses the digest, key parameters, and signature out of the XML document, reconstructs the public key, and uses the following method to validate the signature:

```java
static public boolean verify(String msg, String signature,
                            String mod, String pubExp) {

    BigInteger modulus = new BigInteger(Base64.decode(mod));
    BigInteger exponent = new BigInteger(Base64.decode(pubExp));

    SHA1Digest digEng = new SHA1Digest();
    RSAEngine rsaEng = new RSAEngine();

    RSAPublicKeyParameters publicKey = new RSAPublicKeyParameters(false, modulus, exponent);

    PSSSigner signer = new PSSSigner(rsaEng, digEng, 64);
```
My tests show that XML parsing and digest generation on wireless devices are both very fast. The main performance bottleneck, as expected, is the slow public key algorithms.

The Bouncy Castle Crypto package provides several signer classes using DSA, RSA, and ECC algorithms to sign and verify messages. However, not all of them are practical in real-world devices. Because the Bouncy Castle Crypto package is based purely on the Java language, it relies on the slow JVM to perform even the most intensive big integer mathematical operations without special optimization.

As a result, only the RSA algorithm gives an acceptable performance, and it is barely acceptable. It can verify a simple digital signature with a 1024-bit public key in slightly more than a minute on a 16MHz Palm VII device. The performance can be improved by choosing a weaker key. But even so, the verification process must run as a background thread in any real-world application to avoid user interface lockup.

DSA and ECC algorithm performances are completely unacceptable in their current implementations. A DSA signature with a 1024-bit key and an ECC signature with a 192-bit key take more than an hour to verify on standard Palm VII MIDP.

The performance problems strongly suggest that we need JVMs optimized for big integer mathematical operations and public key algorithms. The JVM must also take advantage of available special hardware and underlying OS features to accelerate security-related math operations. Public key algorithms are used at the handshakes in secure connections such as HTTPS. Many current MIDP VMs can support the HTTPS protocol with reasonable performances. The MIDP4Palm VM can make use of Palm OS's underlying inethttps
protocol to establish secure connections. It is conceivable that future VMs and core language libraries will not only optimize public key operations associated with secure connections, but also make the optimization available to general security functions such as digital signature.