SE448: Information and Cyber Security	**Recitation 9**
Semester 2 5785	**29 May 2025**
Lecturer: Michael J. May	**Kinneret College**

Using Digital Signatures

In this recitation we develop a simple tool which uses the digital signature atoms studied in class this week.

We discussed digital signatures as part of a discussion about how we can verify data being sent over an insecure network. We talked about symmetric and asymmetric digital signatures, both of which amount to an encryption of a summary of a message. Java includes a digital signature implementation that does all of the work for us - hashing and signing at once.

1 Digital Signatures in Java

The Java interfaces for digital signatures is fairly straightforward and similar to what you used for hashing. The primary class you'll need to use is `Signature`. The following methods from the class are useful.

- `getInstance` - prepare a Signature with the algorithm that you want
- `initSign` - prepare the Signature instance for signing data
- `initVerify` - prepare the Signature instance for verifying data
- `update` - add bytes to the Signature instance to prepare for signing or verifying
- `verify` - perform the verification on the bytes provided before and a provided digital signature (in `byte[]` format)
- `sign` - create a digital signature on the bytes provided before (signature is a `byte[]`).

There are a few other helpful classes that you'll need to use as well for digital signatures.

KeyPair Represents a generic public/private key pair. This is an interface and can represent any type of the public/private key pairs that we will use.

PublicKey Represents a generic public key. This is an interface and can represent any type of the public keys that we will use.

PrivateKey Represents a generic private key. This is an interface and can represent any type of the private keys that we will use.

KeyPairGenerator Generates a key pair based on the parameters provided to it.

2 Digital Signatures Tool

We're going to make a command line tool as before. The tool has three operations that it can perform:

1. Generate a new key pair with a given algorithm and key length and store the key in an output file
2. Sign a file using a provided key pair and signature algorithm
3. Verify a signature on a file using a provided key pair and signature algorithm

We will support the following algorithms for key generation

- RSA
- EC

- DSA

We will support the following algorithms for signatures:

- SHA256withRSA
- SHA512withRSA
- SHA224withDSA
- SHA256withDSA
- SHA256withECDSA
- SHA512withECDSA

To support the operations and algorithms, the tool must support the following parameters:

infile The file to process (sign or verify)

keyfile The file containing the key (sign or verify) or to write the key out to (key generation)

keylength The length of the key to be generated (in bits)

alg The algorithm to use (see above lists)

operation Can be **sign, verify, generatekey**

signature The digital signature to verify (for verify) The signature will be provided in hexadecimal format (*e.g.* ABC123, 123DEF).

If any parameters are missing or invalid, output a usage message (*i.e.* Usage: DigitalSignatureCmd -operation= . . .).

2.0.1 Key Generation

When the operation is generatekey, the following parameters are required:

1. operation
2. keylength
3. keyfile

The key is written to the file keyfile.

2.0.2 Signing

When asking to sign a file, the following parameters are required:

1. infile
2. keyfile
3. alg
4. operation

The signature is output to standard output (*i.e.* System.out) in hexadecimal format.

2.0.3 Verifying

When asking to verify a file, the following parameters are required:

1. infile
2. keyfile
3. alg
4. operation
5. signature

The signature must be in hexadecimal format. If the signature is valid, the tool must output "Signature valid". Otherwise, the tool must output "Signature invalid".

2.1 The Key Objects

There are a few key classes that are crucial for Java support for the RSA, EC, and DSA algorithm:

KeyPairGenerator Generates key pairs for you. You just need to give it the algorithm that you want it to create key pairs for. In this case, we'll use "RSA", "EC", and "DSA". Before you can use the generator, you'll need to initialize it using the `initialize` method along with the key pair length that you want.

KeyPair Represents a public/private key pair. It's generated by the KeyPairGenerator using the `generateKeyPair` method.

PrivateKey The private part of the KeyPair. You can extract it from the KeyPair using the `getPrivate` method.

PublicKey The public part of the KeyPair. You can extract it from the KeyPair using the `getPublic` method.

2.2 Saving and Loading Keys

There are a few ways to store private and public keys on a computer. One option is to use a PEM file. Another is to use an X.509 digital certificate. Both options work, but involve significant additional work in the tool. To save time, I implemented a trivial save and restore mechanism for the private and public keys in hexadecimal format in a text file. The methods are already given to you in the starter code to save time. You can have a look at them to see what they do.

When loading a key, the code needs to know what algorithm the key is for. To make things, simple, the load algorithm looks at the file name.

- If the file name ends in .rsa.txt (*e.g.* oneKey.rsa.txt, rsa2048.rsa.txt) it's assumed to be RSA
- If the file name ends in .ec.txt (*e.g.* secondKey.ec.txt, ec384.ec.txt) it's assumed to be EC
- If the file name ends in .dsa.txt (*e.g.* thirdKey.dsa.txt, dsa2048.dsa.txt) it's assumed to be DSA

In our next recitation we'll see how to do key generation and storage in a more secure manner using KeyStore.

3 What to do

Your job is to take the empty code given and implement the logic behind key generation, file signing, and file signature verification.

Support the cryptographic hash algorithms listed above for the three key pair families - SHA256withRSA, SHA512withRSA, SHA224withDSA, SHA256withDSA, SHA256withECDSA, SHA512withECDSA.

4 Tests

To verify that you tool works correctly, I have provided a series of test files and cases on Moodle. Before you continue to the experiments, ensure that your tool works correctly.

When you check your tool, notice that the DSA and ECDSA algorithms include some randomness, so you will not get the precise values that I received for the signatures. Your tool must be able to verify my output, however.

5 What to do

5.1 Experiment 1: Signing a file with RSA

1. Generate a 2048 bit RSA key pair.
2. Create a text file and compute its signature using SHA256withRSA.
3. Then verify it using SHA512withRSA.

- Does it work?
- Can you find an input which will make the verification work under SHA512withRSA?

5.2 Experiment 2: Large Signatures with RSA

1. Create a large text file (*e.g.* a full Wikipedia article) and compute its signature using the 2048 bit RSA key pair.
2. Change one letter of the file and recompute the hash.
3. How much of the signature output changed under SHA256withRSA? Under SHA512withRSA?
4. What does that tell you about the sensitivity of the two algorithms to small changes in the input?

5.3 Experiment 3: Large Signature with DSA and RSA

1. Use a text file and compute a digital signature on it using the 2048 bit RSA key pair.
2. Create a new DSA key pair that's 2048 bits.
3. Verify the first signature using the DSA key pair.
4. What happens? Why?

5.4 Experiment 4: Key Size and Signature Size

1. Use your large text file to compute its signature using SHA256withRSA with the RSA 2048 bit key pair
2. Compute a digital signature using SHA256withDSA using the DSA 2048 bit key pair.
3. How do the lengths of the signatures compare?

5.5 Experiment 5: Using EC Signatures

1. Try to generate a 2048 bit EC key. Does it work?
2. Try to generate a 256 bit EC key.
3. Use the 256 bit EC key to sign a text file using SHA256withECDSA.
4. Sign the file again with SHA512withECDSA and the EC key.
5. How do the lengths of the signature vary?
6. Generate a 384 bit EC key.
7. Use the 384 bit EC key to sign a file using SHA256withECDSA.
8. How does the signature length compare to the length of the signature using the 256 bit key?