Performance Comparison of Elliptic Curve and RSA Digital Signatures

Nicholas Jansma [njansma@engin.umich.edu]
Brandon Arrendondo [barrendo@engin.umich.edu]

April 28, 2004

Abstract. This paper compares the performance characteristics of two public key cryptosystems (RSA and ECC) used in digital signatures to determine the applicability of each in modern technological devices and protocols that use such signatures. Digital signatures are used in message transmission to verify the identity of the sender and to ensure that a message has not been modified after signing. The space and time efficiency of digital signature algorithms is essential to their widespread adoption in message transport systems.

1 Introduction

This paper compares the performance characteristics of RSA and elliptic curve digital signature algorithms by implementing each algorithm and comparing their experimental running-times in an effort to gauge the experimental time efficiencies of each.

Digital signatures are used in message transmission to verify the identity of the sender of the message and to ensure that the message has not been modified after signing. They are essential for verifying the authenticity of a message. The application of digital signatures is widespread in digital computing, taking the place of an ordinary hand-written signature. Because digital signatures are akin to hand-written signatures, they are used in many of the applications of signatures on the Internet (e.g. e-voting, online banking, online college applications, etc).

The importance of digital signatures in digital communications merits the research into relatively new cryptosystems such as elliptic curve cryptography (ECC), especially as the need for more efficient algorithms grows with the growing number of memory-limited mobile electronic devices. The increasing key sizes needed by RSA for security against brute force attacks by powerful computers or distributed computing also makes ECC more appealing, with its smaller secure key sizes [4].

After implementing and running ECC and RSA digital signatures with various key sizes on several test cases, we concluded that the results are mostly consistent with current academic knowledge comparing the two systems. RSA key generation is quite costly in terms of time, both cryptographic schemes are

comparative (up to 7680 bit RSA signing) for message signing, and RSA scales better than ECC in signature verification.

2 Preliminary Background

Public key cryptography is used in digital signatures to verify the identity of the sender of a message and the contents of the message. This must be done in such a way that the private key of the sender remains secret and an unknown adversary is not able to potentially forge the signature. Given a public key cryptosystem in which it is reasonably hard to obtain a user's private key, and public key exchange can take place with a high level of confidence of user identity, digital signatures can be created and used in the following manner:

Alice wants to send a message to Bob and Bob wants to ensure Alice is the actual sender of the message and that the message's contents have not been modified in transit.

1. Alice can generate a private key and public key and send her public key to Bob.
2. Alice then creates a hash of the message she wishes to send to Bob. She then encrypts this hash using her private key. She appends this signature to the message she sends to Bob.
3. Bob can then verify that Alice sent the message by decrypting the signature using Alice's public key. The result of the decryption will be the hash of the message Alice originally sent. Bob can then hash the message in the same way Alice did and compare the two hashes.

Using this method, Bob can prove whether Alice sent the message or not because only Alice's private key could encrypt the signature. He can also prove that the message is the original unmodified message Alice sent, for as long as hashing is relatively unique, any changes in the message would change the message hash (also called the message digest).

The underlying public key system used to generate digital signatures can make a considerable difference in the performance of the digital signature process. The two public key cryptosystems we compare in this paper are RSA and ECDSA.

3 Rivest Shamir Adelman (RSA)

RSA is one of the oldest and most widely used [14] public key cryptography algorithms. The algorithm was invented in 1977 by Ron Rivest, Adi Shamir, and Leonard Adleman.

The RSA cryptosystem is based on the assumption that factoring is a computationally hard task. This means that given sufficient computational resources and time, an adversary should not be able to "break" RSA (obtain a private key) by factoring. This does not mean that factoring is the only way to "break" RSA (in fact, breaking RSA may be easier than factoring, see [5]), but currently no other methods have been proposed to efficiently break RSA.

3.1 RSA Key Generation

A RSA public and private key pair can be generated using the algorithm below [15] :

1. Choose two random prime numbers p and q such that the bit length of p is approximately equal to the bit length of q.
2. Compute n such that $n = p * q$.
3. Compute $\phi(n)$ such that $\phi(n) = (p - 1)*(q - 1)$.
4. Choose a random integer e such that $e < \phi(n)$ and $\gcd(e, \phi(n)) = 1$, then compute the integer d such that: $e*d \equiv 1 \bmod \phi(n)$.
5. (n, e) is the public key, and d is the private key.

3.2 RSA Signature Generation

Signature of a message m is a straightforward modular exponentiation using the hash of the message and the private key. The signature s can be obtained by:

$$s = \text{hash}(m)^d \pmod{n}$$

A common hash algorithm used is SHA-1 (as described in FIPS 180-2 [9]).

3.3 RSA Signature Verification

To verify a signature s for message m, the signature must first be decrypted using the author's public key (n, e). The hash h is thus obtained by

$$h = s^e \pmod{n}$$

If h matches hash(m), then the signature is valid – the message was signed by the author, and the message has not been modified since signing.

4 Elliptic Curve Cryptography (ECC)

An elliptic curve is given by an equation in the form of:

$$y^2 = x^3 + ax + b$$

where $4a^3 + 27b^2 \neq 0$

Many interesting problems arise from the set of points on elliptic curves over a finite field under group operations. The finite fields that are commonly used are those over primes (F_p) and binary fields (F_2^n). The security of ECC is based on the elliptic curve discrete logarithm problem (ECDLP). This problem is defined as:

Given points X, Y on the elliptic curve, find z such that:

$$X = zY$$

The discrete logarithm problem over this group in a finite field is a good one-way function because there are currently no known polynomial time attacks for solving the problem [13]. The methods for computing the solutions to the ECDPL are much less efficient than that of factoring, so ECC can provide the same security as RSA with smaller key lengths.

ECC was developed independently by Neal Koblitz and Victor Miller in 1985.

4.1 ECC Key Generation

To generate a public and private key pair for use in ECC communications, an entity would perform the following steps:

1. Find an elliptic curve $E(K)$, where K is a finite field such as F_p or F_2^n, and a find point Q on $E(K)$. n is the order of Q. Recommended domain parameters for $E(K)$ are suggested in [11].
2. Select a pseudo random number x such that $1 \leq x \leq (n - 1)$.
3. Compute point $P = xQ$.
4. Your ECC key pair is (P, x), where P is your public key, and x is your private key.

4.2 ECC Digital Signatures (ECDSA)

The Elliptic Curve Digital Signature Algorithm (ECDSA) is defined in FIPS 186-2 [10] as a standard for government digital signatures, and described in ANSI X9.62. ECDSA was first proposed by Scott Vanstone [6] in 1992.

4.2.1 ECDSA Signature Generation

To create a signature S for a message m, using ECC key pair (P, x) over $E(K)$, an entity performs the following steps [8]:

1. Generate a random number k such that $1 \leq k \leq (n - 1)$.

2. Compute point $kQ = (x_1, y_1)$.
3. Compute $r = x_1 \pmod{n}$. If $r = 0$, go to step 1.
4. Compute $k^{-1} \pmod{n}$.
5. Compute SHA-1(m), and convert this to an integer e.
6. Compute $s = k^{-1}(e + xr) \pmod{n}$. If $s = 0$, go to step 1.
7. The signature for message m is $S = (r, s)$.

4.2.2 ECDSA Signature Verification

To verify a signature $S = (r, s)$ for message m over a curve $E(K)$ using the author's public key P, an entity performs following [8]:

1. Verify r and s are integers over the interval $[1, n - 1]$.
2. Compute SHA-1(m) and convert this to an integer e.
3. Compute $w = s^{-1} \pmod{n}$.
4. Compute $u_1 = ew \pmod{n}$ and $u_2 = rw \pmod{n}$.
5. Compute $X = u_1Q + u_2P$
6. If $X = \dot{O}$, reject S. Otherwise, compute $v = x_1 \pmod{n}$.
7. Accept if and only if $v = r$.

5 Run-time Comparisons

To test and compare the performance characteristics of the RSA and ECDSA signature algorithms, we independently tested each of the three main components: key generation, signature generation and signature verification.

Since ECC offers security equivalent to RSA using much smaller key sizes, the performances were tested according to the following table, suggested from [4].

Table 5-1: Comparable key sizes (in bits)

Symmetric	ECC	RSA
80	163	1024
112	233	2240
128	283	3072
192	409	7680
256	571	15360

Tests were performed on an Intel P4 2.0 GHz machine with 512MB of RAM. The message used for signing is a 100 KB text file.

5.1 Key Generation

Table 5-2: Key generation performance

Key Generation	Key Length		Time (s)	
	ECC	RSA	ECC	RSA
	163	1024	0.08	0.16
	233	2240	0.18	7.47
	283	3072	0.27	9.80
	409	7680	0.64	133.90
	571	15360	1.44	679.06

Key generation for ECC outperforms RSA at all key lengths, and is especially apparent as the key length increases. Since ECC does not have to devote resources to the computationally intensive generation of prime numbers, ECC can create the private/public key pair in superior speed to RSA comparable lengths.

ECC key generation time grows linearly with key size, while RSA grows exponentially.

5.2 Signature Generation

Table 5-3: Signature generation performance

Signing	Key Length		Time (s)	
	ECC	RSA	ECC	RSA
	163	1024	0.15	0.01
	233	2240	0.34	0.15
	283	3072	0.59	0.21
	409	7680	1.18	1.53
	571	15360	3.07	9.20

The performance of the two algorithms does not differ until the larger key sizes, where ECC outperforms RSA. One important consideration of the signature generation process is that some of the time for each algorithm is spent computing the SHA-1 hash of the message.

5.3 Signature Verification

Table 5-4: Signature verification performance

Verification	Key Length		Time (s)	
	ECC	RSA	ECC	RSA
	163	1024	0.23	0.01
	233	2240	0.51	0.01
	283	3072	0.86	0.01
	409	7680	1.80	0.01
	571	15360	4.53	0.03

Signature verification is where RSA pulls ahead of ECC in performance. The time to verify a message signed in RSA is negligible for the key lengths used, and does not even show a difference until you go from 7680 to 15360 bits. ECC lags behind in performance in every key length, showing nearly linear growth for increasing key sizes.

6 Implementation of RSA Signatures

The implementation of RSA signatures used in the run-time comparisons uses version 5.1 of the Crypto++™ Library [2]. For convenience and timing, the RSA signature process was divided into three programs, listed below.

6.1 rsaKeys.cpp (see Appendix A1 for source code)

rsaKeys is responsible for public and private RSA key generation. It generates the public and private keys, given key size in bits and the file locations to store these keys.

6.2 rsaSign.cpp (see Appendix A2 for source code)

rsaSign is responsible for signing a message using an RSA private key provided by the user. It also requires the location of the message file and location in which to store the signature file.

We chose to implement RSA signatures using PKCS #1 v2.1 Signature Scheme with appendix version 1.5 (as described in the RSA Cryptography Standard [12]). The hashing algorithm used to hash the messages in the signatures is SHA-1 (as described in FIPS 180-2 [9]).

6.3 rsaVerify.cpp (see Appendix A3 for source code)

rsaVerify verifies a signature made using an RSA private key by decrypting it with the public key. It requires, as input from the user, the location to the public key file, the location of the signature file, and the location of the message file that the signature is signing. After decrypting the signature, it performs a hash of the message and compares it to the hash decrypted to validate the signature.

7 Implementation of ECC Digital Signatures (ECDSA)

The implementation of ECC digital signatures follows the guidelines as specified for Elliptic Curve Digital Signatures (ECDSA) in ANSI X9.62. The ECDSA signature process was split into three programs: key generation, signature generation and signature verification. The open source borZoi library [1], version 1.02 was used to implement these programs.

7.1 ecdsaKeys.cpp (see Appendix A4 for source code)

ecdsaKeys is responsible for the generation of ECDSA keys for the specific NIST approved binary fields GF(2^n) of 163, 233, 283, 409 and 571. It generates a public and private key pair that is saved to disk in ASN.1 DER (Distinguished Encoding Rules) syntax according to ANSI X9.62 standards [7].

7.2 ecdsaSign.cpp (see Appendix A5 for source code)

ecdsaSign reads in a message stored on disk, and generates a signature file according to ANSI X9.62 standards based on the user's ECDSA private key file from ecdsaKeys. The signature file is stored on disk in ASN.1 DER encoding.

The hashing function used is SHA-1 (as described in FIPS 180-2 [9]).

7.3 ecdsaVerify.cpp (see Appendix A6 for source code)

ecdsaVerify performs the verification process to determine whether the message and its signature match the public key for the author of the message. It ensures that, if verification is successful, the author of the message was the signer of the message, and the message has not been modified during transmission.

8 Conclusions

Our findings suggest that RSA key generation is significantly slower than ECC key generation for RSA key of sizes 1024 bits and greater. Considering there are affordable devices than can break RSA keys smaller than 1024 bits in a matter of days [3], the cost of key generation can be considered as a factor in the choice of public key systems to use when using digital signatures, especially for smaller devices with less computational resources than our test machine.

Devices that do not need to generate RSA keys for each use, but rather have fixed RSA keys, will not have such a setback due to memory and time constraints compared to ECC, as our results show. RSA is comparable to ECC for digital signature creation in terms of time, and is faster than ECC for digital signature verification. Thus, for applications requiring message verification more often than signature generation, RSA may be the better choice.

ECC is still in its infancy, and thus has not received as much scientific analysis as the much older RSA scheme. The smaller key sizes of ECC potentially

allow for less computationally able devices such as smart cards and embedded systems to use cryptography for secure data transmissions, message verification and other means.

9 References

9.1 Software Libraries

[1] borzoi 1.02 – an open source Elliptic Curve Cryptography Library by Dragongate Technologies Ltd.
Available (28 April 2004) at:
http://www.dragongate-technologies.com/

[2] Crypto++™ Library 5.1 – a Free C++ Class Library of Cryptographic Schemes. Written by Wei Dai.
Available (28 April 2004) at:
http://www.eskimo.com/~weidai/cryptlib.html

9.2 Research Papers and Journals

[3] A. Shamir, and E. Tromer, Factoring Large Numbers with the TWIRL Device.
Available (28 April 2004) at:
http://psifertex.com/download/twirl.pdf

[4] A. Lenstra, and E. Verheul, "Selecting Cryptographic Key Sizes", Journal of Cryptology 14 (2001) 255-293.

[5] D. Boneh, and R. Venkatesan. Breaking RSA May be Easier Than Factoring. Available (28 April 2004) at:
http://theory.stanford.edu/~dabo/papers/no_rsa_red.pdf

[6] S. Vanstone, “Responses to NIST’s Proposal”, Communications of the ACM, 35, July 1992, 50-52

9.3 Standards

[7] Abstract Syntax Notation One Standard.
Available (28 April 2004) at:
http://asn1.elibel.tm.fr/en/

[8] Elliptic Curve Digital Signature Algorithm (ECDSA).
Available (28 April 2004) at:
http://www.comms.scitech.susx.ac.uk/fft/crypto/ecdsa.pdf

[9] FIPS 180-2: The Secure Hash Standard.
Available (28 April 2004) at:
http://csrc.nist.gov/publications/fips/fips180-2/fips180-2

[10] FIPS 186-2: The Digital Signature Standard.
Available (28 April 2004) at:
http://csrc.nist.gov/publications/fips/fips186-2/fips186-2-change1.pdf

[11] SEC 2: Recommended Elliptic Curve Domain Parameters.
Available (28 April 2004) at:
http://www.secg.org/collateral/sec2_final.pdf

[12] PKCS#1 v2.1: RSA Cryptography Standard, RSA Laboratories,
June 14, 2002

9.4 Books and Presentations

[13] I. Blake, G. Seroussi, and N. Smart. Elliptic Curves in Cryptography.
Cambridge University Press, 1999.

[14] P. Riikonen. RSA Algorithm.
Available (28 April 2004) at:
http://iki.fi/priikone/docs/rsa.pdf

[15] W. Mao. Modern Cryptography: Theory and Practice. © 2004. pp. 258

Appendix A1

```
/*****************************************************************************
  EECS 498:  Introduction to Cryptography
  Brandon Arrendondo [barendo@umich.edu]
    Nic Jansma [njansma@umich.edu]

  rsaKeys.cpp

  Generate a pair of RSA keys.
*****************************************************************************/
#include <string>
#include <iostream>
#include <time.h>
#include "base64.h"
#include "default.h"
#include "files.h"
#include "hex.h"
#include "randpool.h"
#include "rng.h"
#include "osrng.h"
#include "rsa.h"
using namespace CryptoPP;

int main(int argc, char *argv[])
{
    std::string privFilename, pubFilename;
  unsigned int keyLength;

    if (argc != 3) {
        std::cout << "RSA Key Generation\n";
        std::cout << "Usage:    rsa_genkeys keylen keyfile\n\n";
        std::cout << "  keyfile will have .pri and .pub appeneded to it\n";
        exit(1);
    }

    keyLength = atoi(argv[1]);
    privFilename = argv[2];
    privFilename.append(".pri");
    pubFilename = argv[2];
    pubFilename.append(".pub");

  AutoSeededRandomPool rng;

    RSAES_OAEP_SHA_Decryptor priv(rng, keyLength);
    HexEncoder privFile(new FileSink(privFilename.c_str()));
    priv.DEREncode(privFile);
    privFile.MessageEnd();

    RSAES_OAEP_SHA_Encryptor pub(priv);
    HexEncoder pubFile(new FileSink(pubFilename.c_str()));
    pub.DEREncode(pubFile);
    pubFile.MessageEnd();
}
```

Appendix A2

```
/*****************************************************************************
  EECS 498:  Introduction to Cryptography
  Brandon Arrendondo [barendo@umich.edu]
    Nic Jansma [njansma@umich.edu]

  rsaSign.cpp

  Sign a message using an RSA private key.
*****************************************************************************/
#include <iostream>
#include <time.h>
#include "base64.h"
#include "default.h"
#include "files.h"
#include "hex.h"
#include "randpool.h"
#include "rng.h"
#include "osrng.h"
#include "rsa.h"
using namespace CryptoPP;

int main(int argc, char *argv[])
{
    std::string priFilename, msgFilename, sigFilename;

  if (argc != 4) {
        std::cout << "RSA Verification\n";
        std::cout << "Usage:    rsa_sign prifile message sig\n\n";
        exit(1);
    }

    priFilename = argv[1];
    msgFilename = argv[2];
    sigFilename = argv[3];

    AutoSeededRandomPool rng;

  FileSource privFile(priFilename.c_str(), true, new HexDecoder);
  RSASSA_PKCS1v15_SHA_Signer priv(privFile);
  FileSource f(msgFilename.c_str(), true, new SignerFilter(rng, priv,
                        new HexEncoder(new FileSink(sigFilename.c_str()))));

  return 0;
}
```

Appendix A3

```
/*****************************************************************************
  EECS 498:  Introduction to Cryptography
  Brandon Arrendondo [barendo@umich.edu]
    Nic Jansma [njansma@umich.edu]

  rsaVerify.cpp

  Verify a RSA signed message
*****************************************************************************/
#include <string>
#include <iostream>
#include <time.h>
#include "base64.h"
#include "default.h"
#include "files.h"
#include "hex.h"
#include "randpool.h"
#include "rng.h"
#include "osrng.h"
#include "rsa.h"
using namespace CryptoPP;

int main(int argc, char *argv[])
{
    std::string pubFilename, msgFilename, sigFilename;

  if (argc != 4) {
        std::cout << "RSA Verification\n";
        std::cout << "Usage:    rsa_verify pubfile message sig\n\n";
        exit(1);
    }

    pubFilename = argv[1];
    msgFilename = argv[2];
    sigFilename = argv[3];

  FileSource pubFile(pubFilename.c_str(), true, new HexDecoder);
  RSASSA_PKCS1v15_SHA_Verifier pub(pubFile);

  FileSource signatureFile(sigFilename.c_str(), true, new HexDecoder);
  if (signatureFile.MaxRetrievable() != pub.SignatureLength()) {
        std::cout << "Incorrect public key\n";
    }
  SecByteBlock signature(pub.SignatureLength());
  signatureFile.Get(signature, signature.size());

  VerifierFilter *verifierFilter = new VerifierFilter(pub);
  verifierFilter->Put(signature, pub.SignatureLength());
  FileSource f(msgFilename.c_str(), true, verifierFilter);

  if(verifierFilter->GetLastResult())
  {
      std::cout << "Valid signature" << std::endl;
  }
  else
  {
      std::cout << "Invalid signature" << std::endl;
  }

return 0;
}
```

Appendix A4

```
/**************************************************************************
  EECS 498:  Introduction to Cryptography
  Brandon Arrendondo [barendo@umich.edu]
    Nic Jansma [njansma@umich.edu]

  ecdsaKeys.cpp

  Generate ECC private and public key pair
**************************************************************************/
#include "borzoi.h"
#include "nist_curves.h"
#include <fstream>
#include <string>

using namespace std;

void gen_keys(string pass, string oFile, int keylen);

int main (int argc, char* argv[]) {
    // send to key generation function

    if (argc != 4) {
        cout << "ECDSA Key Generation" << endl;
        cout << "Usage: ecdsa_genkeys \"password\" output_file keylen" << endl << endl;
        cout << "   Files output_file.pub and output_file.priv will be created" << endl;
        cout << "   keylen: can be 163, 233, 283, 409, 571 " << endl;
        exit(1);
    }

    // generate keys to a file
    gen_keys(argv[1], argv[2], atoi(argv[3]));
    return 0;
}

void gen_keys(string pass, string oFile, int keylen) {
    ECPrivKey *keyPriv;
    string sFilePri = oFile;

    ECPubKey *keyPub;
    string sFilePub = oFile;

    EC_Domain_Parameters dp;

    // generate file names
    sFilePri.append(".pri");
    sFilePub.append(".pub");

    // determine keysize
    switch (keylen) {
        case 163:
            use_NIST_B_163();
            dp = NIST_B_163;
            break;

        case 233:
            use_NIST_B_233();
            dp = NIST_B_233;
            break;

        case 283:
            use_NIST_B_283();
            dp = NIST_B_283;
            break;

        case 409:
            use_NIST_B_409();
            dp = NIST_B_409;
            break;
```

```
        case 571:
            use_NIST_B_571();
            dp = NIST_B_571;
            break;
    }

    cout << "Using keysize: " << keylen << " bits." << endl;

    // create private key
    cout << "Generating private key, please be patient... " << endl;
    keyPriv = new ECPrivKey(dp);
    cout << "done!" << endl;

    // encrypt this to DER and Hex (for file output and screen output)
    DER keyPrivDER(*keyPriv);
    HexEncoder oHex(keyPrivDER);

    // show the hex encoding on screen
    cout << oHex << endl << endl;

    // generate public key
    cout << "Generating public key... ";
    keyPub = new ECPubKey(*keyPriv);
    cout << "done:" << endl;

    // convert to DER and HEX
    DER keyPubDER(*keyPub);
    oHex = HexEncoder(keyPubDER);

    // print out public key to screen
    cout << oHex << endl << endl;

    // output DER for private key to file
    cout << "Creating file " << oFile << ".pri ...";
    ofstream outfile (sFilePri.c_str(), std::ios::binary);
    if (!outfile) {
        cout << "Couldn't open " << sFilePri << " for writing!" << endl;
        exit(1);
    }
    outfile << keyPrivDER;
    outfile.close();
    cout << "done." << endl;

    // output DER for public key to file
    cout << "Creating file " << oFile << ".pub ...";
    outfile.open(sFilePub.c_str(), ios::binary);
    if (!outfile) {
        cout << "Couldn't open " << sFilePub << " for writing!" << endl;
        exit(1);
    }
    outfile << keyPubDER;
    outfile.close();
    cout << "done." << endl;

    // all done!
    cout << endl << "Finished successfully!" << endl;

    return;
}
```

Appendix A5

```
/**************************************************************************
  EECS 498:  Introduction to Cryptography
  Brandon Arrendondo [barendo@umich.edu]
    Nic Jansma [njansma@umich.edu]

  ecdsaSign.cpp

  Sign a message with ECC private keys.
**************************************************************************/
#include "borzoi.h"
#include "nist_curves.h"
#include <fstream>
#include <string>

using namespace std;

void ecdsa_sign(string sInFile, string sOutFile, string sKeyFile);
void print_octetstr(OCTETSTR o);

int main (int argc, char* argv[]) {
    // send to key generation function

    if (argc != 4) {
        cout << "ECDSA Signature Generation" << endl;
        cout << "Usage: ecdsa_sign input_file sig_file priv_key" << endl << endl;
        exit(1);
    }

    // generate keys to a file
    ecdsa_sign(argv[1], argv[2], argv[3]);
    return 0;
}

void ecdsa_sign(string sInFile, string sOutFile, string sKeyFile) {
    OCTETSTR oStr;
    OCTET o;
    OCTETSTR hash;
    char c;

    // open file from disk
    cout << "Reading in data from " << sInFile << "... ";
    ifstream ifsInFile(sInFile.c_str(), std::ios::binary);
    if (!ifsInFile) {
        cout << "Error! Couldn't read input file " << sInFile << "!" << endl;
        exit(1);
    }

    // read file into memory
    while (ifsInFile.get(c)) {
        o = (unsigned char)c;
        oStr.push_back (o);
    }

    cout << "done." << endl << endl;

    // compute hash of message
    hash = SHA1(oStr);
    HexEncoder h(hash);
    cout << "Hash: " << h << endl;

    // read in private key
    oStr.clear();

    cout << "Reading in private key... " << endl;
    ifsInFile.close();
    ifsInFile.clear();
    ifsInFile.open(sKeyFile.c_str(), std::ios::binary);
    if (!ifsInFile) {
```

```
        cout << "Error! Couldn't read key file " << sKeyFile << "!" << endl;
        exit(1);
    }

    // read file into memory
    while (ifsInFile.get(c)) {
        o = (unsigned char)c;
        oStr.push_back (o);
    }
    ECPrivKey keyPriv = DER(oStr).toECPrivKey();
    cout << "done!" << endl;

    cout << "Generating signature, please wait..." << endl;
    // convert input to private key

    // create signature object
    ECDSA sig (keyPriv, OS2IP(hash));

    DER sigDER(sig);

    cout << "done!" << endl << endl;

    // output DER to sig file
    cout << "Creating signature file " << sOutFile << "...";
    ofstream outfile(sOutFile.c_str(), ios::binary);
    if (!outfile) {
        cout << "Couldn't open " << sOutFile << " for writing!" << endl;
        exit(1);
    }
    outfile << sigDER;
    outfile.close();
    cout << "done." << endl << endl;

    cout << "Signature:" << endl;
    h = HexEncoder(sigDER);
    cout << h << endl << endl;

    cout << "Finished successfull!" << endl;

    return;
}

void print_octetstr(OCTETSTR o) {
    for (int i = 0; i < o.size(); i++) {
        cout << o[i];
    }
}
```

Appendix A6

```
/*****************************************************************************
  EECS 498:  Introduction to Cryptography
  Brandon Arrendondo [barendo@umich.edu]
    Nic Jansma [njansma@umich.edu]

  ecdsaVerify.cpp

  Verify a message signed with ecdsaSign.
*****************************************************************************/

#include "borzoi.h"
#include "nist_curves.h"
#include <fstream>
#include <string>

using namespace std;

bool ecdsa_verify(string sInFile, string sSigFile, string sKeyFile);

int main (int argc, char* argv[]) {
    // send to key generation function

    if (argc != 4) {
        cout << "ECDSA Signature Generation" << endl;
        cout << "Usage: ecdsa_verify input_file sig_file pub_key" << endl << endl;
        exit(1);
    }

    // generate keys to a file
    if (ecdsa_verify(argv[1], argv[2], argv[3])) {
        cout << "Verification successful!" << endl;
    } else {
        cout << "Verification FAILED!" << endl;
    }
    return 0;
}

bool ecdsa_verify(string sInFile, string sSigFile, string sKeyFile) {
    OCTETSTR oStr;
    OCTET o;
    OCTETSTR hash;
    char c;

    // open file from disk
    cout << "Reading in data from " << sInFile << "... ";
    ifstream ifsInFile(sInFile.c_str(), std::ios::binary);
    if (!ifsInFile) {
        cout << "Error! Couldn't read input file " << sInFile << "!" << endl;
        exit(1);
    }

    // read file into memory
    while (ifsInFile.get(c)) {
        o = (unsigned char)c;
        oStr.push_back (o);
    }

    cout << "done." << endl << endl;

    // compute hash of message
    hash = SHA1(oStr);
    HexEncoder h(hash);
    cout << "Hash: " << h << endl << endl;

    // read in public key
    oStr.clear();

    cout << "Reading in public key... " << endl;
```

```
    ifsInFile.close();
    ifsInFile.clear();
    ifsInFile.open(sKeyFile.c_str(), std::ios::binary);
    if (!ifsInFile) {
        cout << "Error! Couldn't read key file " << sKeyFile << "!" << endl;
        exit(1);
    }

    // read file into memory
    while (ifsInFile.get(c)) {
        o = (unsigned char)c;
        oStr.push_back (o);
    }
    ECPubKey keyPub = DER(oStr).toECPubKey();
    cout << "   done!" << endl << endl;

    // read in public key
    oStr.clear();

    cout << "Reading in signature file... " << endl;
    ifsInFile.close();
    ifsInFile.clear();
    ifsInFile.open(sSigFile.c_str(), std::ios::binary);
    if (!ifsInFile) {
        cout << "Error! Couldn't read key file " << sKeyFile << "!" << endl;
        exit(1);
    }

    // read file into memory
    while (ifsInFile.get(c)) {
        o = (unsigned char)c;
        oStr.push_back (o);
    }

    ECDSA sig = DER(oStr).toECDSA();

    cout << "Verifying signature, please wait..." << endl;
    return (sig.verify(keyPub, OS2IP(hash)));
}
```