Digital Signatures

MATH 493 Final Report

Turid Herland

Supervisors:
Dr. Ernst Kani & Dr. Andrew D. Lewis

Queen's University, Kingston, Ontario, Canada

April 2, 2004

Abstract

Digital signatures are implemented using public key cryptography, and in this project three different mathematical algorithms for digital signatures have been implemented. These three algorithms are first described, and then their implementations are discussed. Finally, some known attacks against the algorithms are discussed, and the implemented solutions are given.

Acknowledgments

I would like to thank my husband Johan Herland for support and understanding, and for giving me all the help I needed with C. I also wish to thank Dr. Ernst Kani for giving me the opportunity to do this project and for considerable help and guidance. Finally, I would like to thank Dr. Andrew D. Lewis for guidance in design and help with LaTeX.

Contents

List of Figures

List of Tables

1 Introduction

In our society, an individual's signature on a document proves that this individual acknowledges the contents of that document, whether it a personal letter, a credit card transaction, a contract, or any other type of document. But how does one sign a digital document? Digital signatures should provide the same functionality and security for digital documents as handwritten signatures do for paper documents.

Since any digital information can be easily copied and altered, a digital version of a handwritten signature will not do. To make digital signatures difficult to forge, one person's signature must be different on different documents. So, the signature must be tied both to the signer and to the message or document that is to be signed. Also, it is important that anyone can verify that the signature belongs to the person it claims to belong to, and that the signed document has not been altered after it was signed. Therefore, if any of these conditions are violated, the signature is no longer valid.

Digital signature schemes use classes of mathematical algorithms that enable the signature system described above. In other words, they allow a sender to sign a message, which can be verified by any receiver. These schemes are public key cryptography schemes, that is, each user must have a key pair, consisting of a public key and a private key. These keys are used as parameters in encryption and decryption, or, in the case of digital signatures, in signing and signature verification. The private key is secret, and should not be shared with anyone because this key is used when creating a signature. The public key is used in signature verification, and it should be available for anyone who would like to verify a person's signature. It must not be computationally feasible to calculate the private key from the public key, since the public key schemes rely on the fact that only the owner of the private key has access to it.

For example, if Alice wants to send a signed message to Bob, she applies the signature function to her message, using her private key. She then appends the signature to her message, and sends it to Bob. When Bob receives the message, he applies the signature verification function to the signature part, using Alice's public key. This function verifies the signature if and only if the signature was created using Alice's private key on the given message.

The output of most signature functions is approximately the same length as the input, therefore, adding a signature to a message would roughly double the size of the message. To avoid this, hash functions are commonly used with digital signatures. A hash function takes a message of arbitrary length as its input, and outputs a fixed-length digest of the message. This digest is normally no more than 200 bits long, so the digest is usually shorter than the original message. The idea is to apply the signature function to the digest instead of the whole message, to create a shorter signature. A digital signature system that uses a hash function is illustrated in Figure 1.

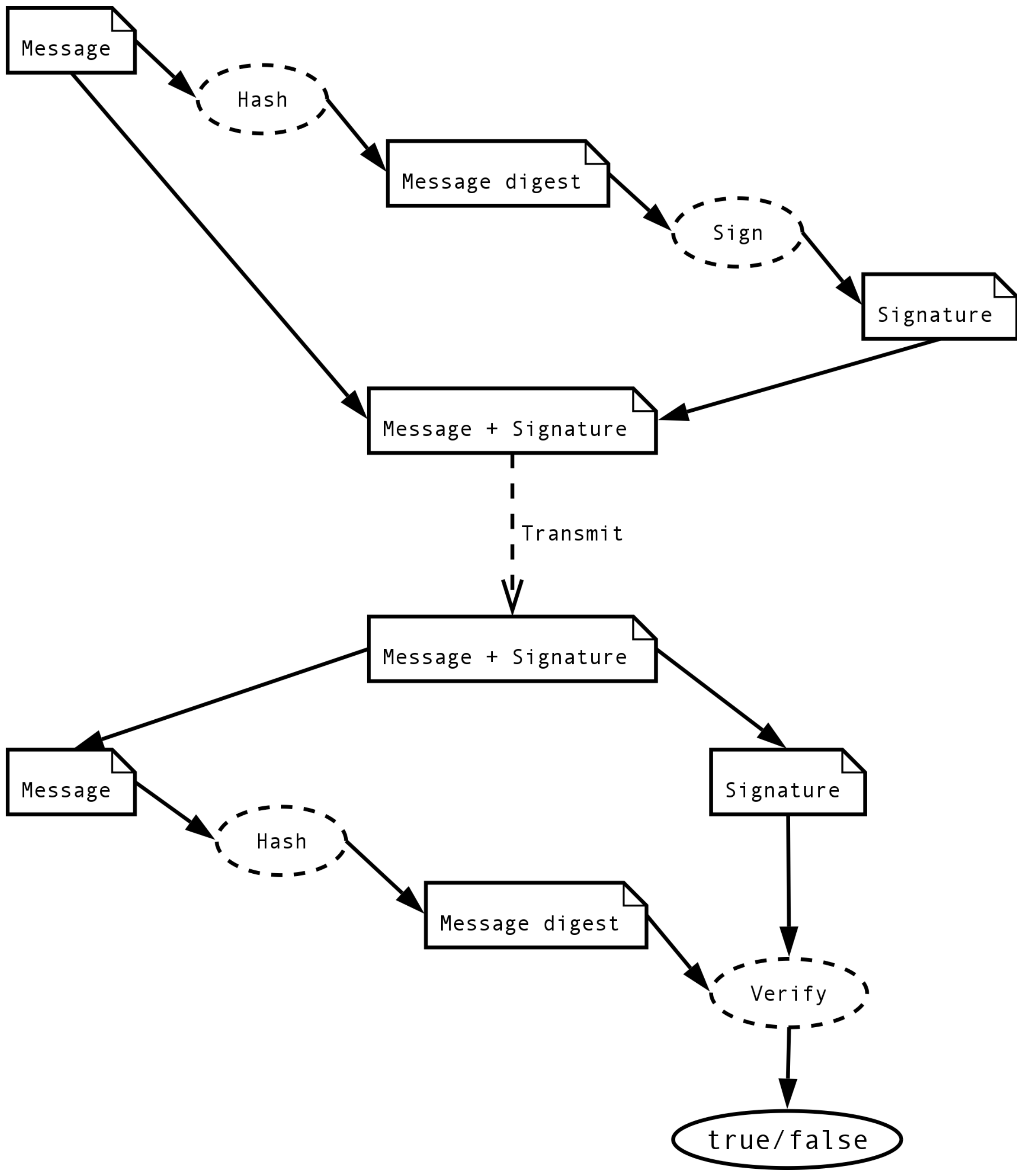

Figure 1: A digital signature system

2 Description & Approach

This project implements digital signatures for plain text messages. This scheme can be easily extended to a signature system for email communication, as plain text email messages are just a specific type of plain text messages. Three different signature schemes that base their security on two different mathematical problems were implemented. These schemes were RSA (Rivest-Shamir-Adleman), DSA (Digital Signature Algorithm) and ECDSA (Elliptic Curves Digital Signature Algorithm). RSA bases its security on the difficulty of factoring, DSA bases its security on the difficulty of finding discrete logarithms in finite fields, and ECDSA bases its security on the analogue of the discrete logarithm problem for elliptic curves over finite fields. All three algorithms need good hash functions to provide acceptable security.

2.1 Hash functions

A hash function h is a function that takes as its input a message m of arbitrary length, and produces as its output $h(m)$, a hash value or a message digest of fixed length. A good cryptographic hash function satisfies three basic properties:

1. Given a message m, it is easy to compute $h(m)$.
2. Given a hash value y, it is infeasible to find a message m so that $y = h(m)$.
3. It is infeasible to find two messages m_1 and m_2 such that $h(m_1) = h(m_2)$.

Property 1. is needed so that the calculation will not take too much time in any implementation. Properties 2. and 3. are for security reasons: If Bob signed a message m using a digital signature algorithm on $y = h(m)$, and Eve could find a different m' such that $h(m') = y$, then Eve could claim that Bob had signed the message m'. Also, if it were easy to find two different messages m_1 and m_2 so that $h(m_1) = h(m_2)$, then Eve could prepare two documents m_1 and m_2, where m_1 was a document that Bob would want to sign, and m_2 was a document that Bob would not sign, and $h(m_1) = h(m_2)$. If Eve got Bob to sign m_1, she could also claim that he had signed m_2.

Since good hash functions are very difficult to design, this project does not attempt to do so. For a description of existing hash functions, see chapter 18 of [Schneier96]. For this project, the Secure Hash Algorithm (SHA) was used with all three signature algorithms.

2.2 RSA

RSA was invented by Ron Rivest, Adi Shamir and Leonard Adleman in 1978 ([RSA]). It is one of the oldest and most popular public key cryptosystems, and it is also one of the easiest systems to understand and implement. RSA is discussed in [Koblitz94] pp. 92-95 and in [Schneier96] pp. 466-474.

To generate the RSA keys, two random prime numbers p and q are generated, each of length approximately 512 bits or more. These two numbers are then multiplied together to get $n = pq$.

Definition 1. Let n be a positive integer. Then the *Euler phi-function* $\phi(n)$ is defined to be the number of nonnegative integers b less than n which are relatively prime to n.

Now since $n = pq$ where p and q are prime numbers, it follows that $\phi(n) = (p-1)(q-1)$.

The public key is a randomly chosen integer d such that $d < \phi(n)$ and d and $\phi(n)$ are relatively prime. Then the private key e is calculated by taking $e = \text{rem}(1/d, \phi(n))$. After e has been calculated, p and q (and $\phi(n)$) should be discarded. n is usually also included with both the private key and the public key, because n is needed both to sign a message and to verify a signature.

To sign a message m, a hash function h is first applied, to get the hash value $h(m)$. Then the signature s is calculated using $s = \text{rem}(h(m)^e, n)$.

To verify a signature, first calculate the hash value $h(m)$ of the message m, and then check if $h(m) \equiv s^d \mod n$, where s is the signature. The signature should be verified if the congruence holds.

To see why this congruence is equivalent to a valid signature, suppose that the signature is valid. Then $s \equiv h(m)^e \mod n$, and hence $s^d \equiv (h(m)^e)^d \equiv h(m)^{1+k\phi(n)} \mod n$ for any positive integer k. But $a^{1+k\phi(n)} \equiv a \mod n$ for any positive integer a (see appendix A for proof), and therefore $s^d \equiv h(m) \mod n$.

Conversely, if $s^d \equiv h(m) \mod n$, then

$$s^d \equiv h(m) \equiv h(m)^{1+k\phi(n)} \equiv h(m)^{ed} \equiv (h(m)^e))^d \mod n$$

and hence $s \equiv h(m)^e \mod n$, so the signature is valid.

Suppose Bob has private key e and public key d. Then Eve would have d and n. If she could factor n to find p and q, she could calculate $\phi(n) = (p-1)(q-1)$ and hence calculate Bob's private key $e \equiv d^{-1} \mod \phi(n)$. In general, knowledge of $\phi(n)$ is equivalent to knowledge of the factorization of n. (See [Koblitz94], p. 22 for proof). It would also be possible for Eve to try every possible e until she finds an e such that $a^e d = a \mod n$ for all integers a, but this is even less effective than factoring n.

2.3 DSA

DSA (Digital Signature Algorithm) was proposed by the US government's National Institute of Standards and Technology (NIST) as the new Digital Signature Standard (DSS) in 1991. The official description of DSA can be found in NIST's latest version of the DSS document, [FIPS186-2]. DSA is also discussed in [Schneier96], pp. 483-495, and briefly in [Koblitz94], pp. 101-102.

For DSA key generation, a random prime q of about 160 bits is first generated. Then another prime p of about 1024 bits is also chosen at random, but with the property that $p \equiv 1 \mod q$. Next, choose a random integer g_0, and calculate $g = \text{rem}(g_0^{(p-1)/q}, p)$. If $g \neq 1$, then g generates a unique subgroup of $\mathbb{F}_p^*$ (where $\mathbb{F}_p^* = \mathbb{F}_p \setminus \{0\}$) of order q (see appendix B for proof). Now, choose a random integer x such that $0 < x < q$, and let x be the private key, and $y \equiv g^x \mod p$ the public key. Usually, g, p and q are also distributed with the keys, since they are needed in the sign and verify algorithms.

To sign a message m, a hash function must first be applied to the message to get the hash value $h(m)$. The Secure Hash Algorithm is specified in the DSA protocol, and this gives a hash value of 160 bits. Then choose a random integer k such that $0 < k < q$, and calculate $g' = \mathrm{rem}(g^k, p)$ and $r = \mathrm{rem}(g', q)$. Finally, find an integer s such that $s \equiv k^{-1}(h(m) + xr) \mod q$, and let the pair (r, s) be the signature. Now if $r = 0$ or $s = 0$, then one must choose a new value for k and calculate r and s again. This is because if $r = 0$, then $s \equiv k^{-1}h(m) + 0 \mod q$, so the signature generation would not involve the private key x, and hence the signature would be trivial to forge. And if $s = 0$, then $s^{-1} \mod q$, which is needed in the signature verification algorithm, would not exist. (Note that since k is chosen at random, the probability that either $r = 0$ or $s = 0$ is very small, so the first k that is chosen is usually suitable).

For signature verification, begin by calculating the hash value $h(m)$ of the message m. Then compute $u_1 \equiv s^{-1}h(m) \mod q$ and $u_2 \equiv s^{-1}r \mod q$. Now verify the signature if and only if $r \equiv (g^{u_1}y^{u_2} \mod p) \mod q$.

Now $g^{u_1}y^{u_2} = g^{s^{-1}h(m)}g^{xs^{-1}r} = g^{s^{-1}(h(m)+xr)} = g^k$. But $r \equiv (g^k \mod p) \mod q$, so the signature should indeed be verified if and only if $r \equiv (g^{u_1}(g^x)^{u_2} \mod p) \mod q$.

The security of DSA depends on the difficulty of finding x when $y = g^x \mod p$ is given. This is known as the discrete logarithm problem:

Definition 2. If G is a finite group, and $a \in G$, and $b \in G$ such that $b = a^x$ (in the group law of G) for some integer x, then x is the *discrete logarithm* of b to the base a. The *discrete logarithm problem* is the problem of finding such an x when a, b and G are given.

In DSA, the key pair consists of the private key x and the public key $y = g^x$, where g^x is calculated using the group law of the finite group $\mathbb{F}_p^*$. So, calculating Bob's private key from his public key is equivalent to solving the discrete logarithm problem in $\mathbb{F}_p^*$.

However, there is another way that Eve could find Bob's private key: If Bob has signed a message m, then Eve has r, and $r \equiv (g^k \mod p) \mod q$. So by solving the discrete logarithm problem in the subgroup generated by g (which has order q), Eve could find k, and from k she could calculate x. But, according to [Koblitz94], p. 102, finding discrete logarithms in this subgroup seems to be no easier in practice than finding discrete logarithms in $\mathbb{F}_p^*$.

2.4 Elliptic curves and ECDSA

2.4.1 Elliptic curves over finite fields

Most of the calculations in ECDSA are done in the group of points on an elliptic curve over a finite field. An elliptic curve is a curve with an equation of the form

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

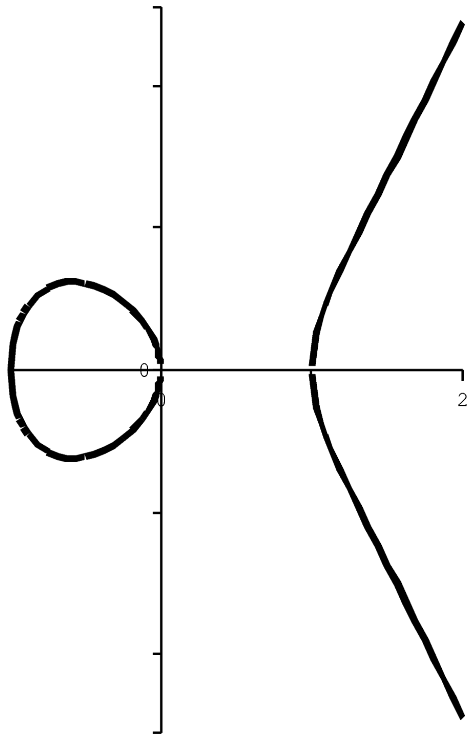

Figure 2: The elliptic curve $E:\ y^2 = x^3 - x$

and with the discriminant of E

$$\Delta_E = -2^4(4a^3 + 27b^2) \neq 0$$

If the curve is defined over a field of characteristic 2 or 3, more general equations are needed both for the curve and the discriminant (see [Koblitz94] chapter VI for details). An example of this type of elliptic curve is shown in Figure 2.

ECDSA uses curves over finite fields, that is the underlying field for the equations is finite, typically $\mathbb{F}_p$ for some prime p. The set of points on an elliptic curve over $\mathbb{F}_p$ forms a finite group $E(\mathbb{F}_p)$ together with $\mathcal{O}$, the point at infinity, with the following group law:

Let P and Q be two points in $E(\mathbb{F}_p)$ with $E : y^2 = x^3 + ax + b$. Then:

1. If P and Q have different x-coordinates, then the line going through P and Q intersects the curve at a third point. Reflect this point in the x axis, and the resulting point is the point $P + Q$.

2. If P and Q have the same x-coordinates, but different y-coordinates, then the line through P and Q will not intersect the curve at any third point. Then $P + Q = \mathcal{O}$.

3. If $Q = \mathcal{O}$, then $Q + P = P$ for all points P on E.
4. If $P = (x, y)$ and $Q = (x, -y)$ then $Q = -P$ and $P + Q = P - P = \mathcal{O}$.
5. If $P = Q$, take the tangent line to the curve at the point P. If this line intersects the curve at a second point, then reflect this point in the x axis, and define the resulting point to be $P + Q = 2P$. If the tangent line does not intersect the curve at a second point, then $P + Q = 2P = \mathcal{O}$.

$\mathcal{O}$ is the identity element of the addition law. It is difficult to prove that this addition law is really a group law, and the proof is out of the scope of this report. A proof can be found in [Silverman].

The graphical description of the addition law given above, can easily be used to derive this formula for adding two points P_1 and P_2 in $E(F)$ with $E : y^2 = x^3 + ax + b$ and all operations done in the field F (see [Koblitz94], p. 170 for derivation):

$$\text{Let } P_1 = (x_1, y_1),\ P_2 = (x_2, y_2),\ P_3 = (x_3, y_3) \quad \text{and} \quad P_3 = P_1 + P_2$$

$$\text{Then} \quad x_3 = \lambda^2 - x_1 - x_2 \quad \text{and} \quad y_3 = \lambda(x_1 - x_3) - y_1$$

$$\text{where} \quad \lambda = \begin{cases} \frac{y_2 - y_1}{x_2 - x_1} & \text{if } x_1 \neq x_2 \\ \frac{3x^2 + a}{2y_1} & \text{if } x_1 = x_2 \text{ and } y_1 \neq -y_2 \end{cases}$$

$$\text{If } x_1 = x_2 \text{ and } y_1 = -y_2, \text{ then } P_3 = \mathcal{O}$$

2.4.2 ECDSA

ECDSA is the elliptic curve analogue of NIST's Digital Signature Algorithm. It is described in [Koblitz98], pp. 134-136.

To generate the keys in ECDSA, start with an elliptic curve $E : y^2 = x^3 + ax + b$ over a prime field $\mathbb{F}_p$, where p has at least 192 bits. Now take a point P on the curve, such that P has prime order q (that is $qP = \underbrace{P + P + \cdots + P}_{q} = \mathcal{O}$ for some prime number q). Then, choose a random integer x such that $1 < x < q-1$, and let x be the private key and $Q = xP$ the public key. The curve E, the field $\mathbb{F}_p$, the point P and its order q are also needed to sign and verify signatures, so these parameters could also be distributed with the keys.

To generate a signature for a message m, first apply a hash function to m, to get the hash value $h(m)$. Then select a random integer k such that $1 < k < q - 1$, and compute $kP = (x_k, y_k)$. Let $r = \text{rem}(x_k, q)$. If $r = 0$, choose a new value for k, to get $r \neq 0$. Now compute $s \equiv k^{-1}(h(m) + xr) \mod q$, and, as before, if $s = 0$ choose a new value for k and try again. The signature is the pair (r, s), where both $r \neq 0$ and $s \neq 0$.

To verify a signature, first verify that r and s are indeed integers in the interval $[1, q-1]$. Then, calculate the hash value $h(m)$ of the message m, and compute $u_1 \equiv s^{-1}h(m) \mod q$ and $u_2 \equiv s^{-1}r \mod q$. Next calculate $u_1P + u_2Q = (x_0, y_0)$, and let $v = \text{rem}(x_0, q)$. Verify the signature if and only if $v = r$.

$u_1P + u_2Q \equiv s^{-1}h(m)P + s^{-1}rxP \equiv s^{-1}(h(m) + rx)P \equiv kP \mod q$ if the signature is valid. Hence $x_k \equiv x_0 \mod q$, and since $r = \text{rem}(x_k, q)$ and $v = \text{rem}(x_0, q)$, the signature is valid whenever $r = v$.

The security of ECDSA relies on a special case of the discrete logarithm problem defined earlier. The definition is repeated here, in the context of elliptic curves, for clarity:

Definition 3. If A is a point in $E(\mathbb{F}_p)$, then the *discrete logarithm problem* in $E(\mathbb{F}_p)$ (to the base A) is, given a point B in $E(\mathbb{F}_p)$, to find an integer x such that $B = xA$ if such an integer exists.

In ECDSA, the private key is an integer x, and the public key is the point $Q = xP$ for a given point P. Therefore, calculating the private key from the public key is the same as solving the discrete logarithm problem to the base P in $E(\mathbb{F}_p)$. One could also try and calculate x from a given signature (r, s) by calculating k from r, and then use k to get x from the expression for s. But r is just the x-coordinate (mod q) of kP, so again, one would have to solve the discrete logarithm problem to get k, and hence to get the private key x.

3 Design

3.1 Programming language and environment

Since cryptographic applications typically deal with very large numbers it was very important to choose a programming language and environment that could handle large precision arithmetic. It was also desirable to use a language that had some predefined functions for standard number theoretic operations like primality testing and modular arithmetic.

One option was to write this project in Maple. Since Maple is primarily a mathematical tool, it provides all the desired mathematical functions, making large precision arithmetic not a problem. However, Maple is a heavy program in itself, and even small programs tend to run significantly slower in Maple than in other multi-purpose programming languages.

Another option was to use the C programming language. C is more low-level than Maple and most other programming languages, and therefore produces significantly faster code than most other options. However, C lacks many of the predefined functions that more high-level languages include. Fortunately, most common functions can be found in libraries that can be obtained and easily utilized from within the C code. One such library is GMP: Gnu Multiple Precision Arithmetic Library ([GMP]). This library handles numbers of arbitrary precision, and cryptography is among its main target applications. Also, GMP is designed to be fast, even for arbitrarily large numbers.

After having considered the above options, it was decided to use C with GMP for the development of this project, because programming in C yields a shorter development cycle than the alternatives. Also, the final program is easier to use, since a stand-alone command-line program doesn't require the user to start Maple and import the project before execution can begin. Moreover, the command-line interface makes it fairly easy to integrate the program with standard Unix utilities for sending and receiving mail.

3.2 Hash algorithms

NIST specifies its own Secure Hash Algorithm (SHA) as the hash algorithm that should be used with DSA in [FIPS186-2]. However, DSA could also work with other hash algorithms, so some research was done to see if there were other suitable choices.

MD5 seemed to be the only real competitor to SHA, based on the descriptions of different hash algorithms given in [Schneier96]. However, SHA was chosen for this project, because it has a longer hash value than MD5 and therefore provides more security for digital signatures. MD5 is faster than SHA, but since the input to the hash function in this case will be relatively short text messages, the difference in speed is not likely to be noticed.

3.3 Cryptographic algorithms

The goal of the project was to develop digital signature programs that use several different cryptographic schemes. RSA, which is a simple, but fairly secure algorithm, was the first scheme to be implemented. RSA was chosen to be the first algorithm because is was important to get the other parts of the programs – the framework – up and running, before time could be spent on developing more sophisticated cryptographic algorithms. Next, the DSA algorithm was implemented, because it provided a convenient stepping stone for the third algorithm – ECDSA – which is considered one of the most secure digital signature algorithms available today. RSA, DSA and ECDSA are the three algorithms for digital signatures that NIST recommend in [FIPS186-2].

3.4 Program design

While three different signature systems are used, all three are implemented using the same overall structure, consisting of one program to generate keys, one program to sign messages, and one program to verify signatures. In addition to this, some functions that are used by several of these main programs, are collected in separate files. The hash function and methods used to read text messages and signatures are examples of such functions. Finally, a Makefile accompanies the source files to facilitate easy building and installing of the programs. The system depends on the GMP ([GMP]) and Mhash ([Mhash]) libraries.

The signature generation programs can sign messages given on the standard input (usually the keyboard) or in a text file. If the message is supplied at the standard input, the signature generation program will output the message with the appended signature at the standard output (usually the terminal). If the message is supplied in a text file, then the signature generation program can either append the signature to the same text file, or write the message and signature to a new text file specified by the user. The choice of taking the input from standard input and providing the output at standard output, will again make it fairly easy to integrate these programs with other standard Unix utilities for sending and receiving mail.

4 Results

4.1 Software

Nine executable programs were successfully designed and implemented. The programs implement key generation, signature generation and signature verification for each of the three algorithms RSA, DSA and ECDSA. The software is written in the C programming language on GNU/Linux, and is therefore available on all Unix/POSIX platforms. The programs have been successfully tested on Linux/PPC and Mac OS X. The software was distributed on CD with some copies of this report.

For each signature scheme implemented, there is one key generation program that generates the public and private keys needed for the algorithm and stores them in a public key file and a private key file respectively. The signature generation program can then sign a message using the key in the private key file, and, depending on the user input, a signature will either be appended to the message file, or the message and the signature will be written to a new file. The signature verification program verifies the signature for the given message using the key in the public key file, and prints out '`Signature verified`' or '`Signature NOT verified!`', depending on if the signature is valid or not.

4.2 An example

Here, an example of running the DSA programs is presented. The example assumes that the software has already been installed on the system as specified in appendix C. All numbers in the example are represented in hexadecimal (base 16).

To run the key generation program, type `dsa_generate` at the command line. In this example, this command produced these parameters:

```
p = e731282cd99d9c241fd54a8c23ab5e24ee4de6334aa65851702d5dfc6fb76e6742da465e6bcd
    1a1ebf058fe4b576fbae7ebff4050771660015363a1c503918fd1bd5803837f8f57afb9ed81d
    415624583f6da5815e7f01ad10aea0eb2bc7e729de86e7f9046c59dadde9f42211dfbd488221
    bcc37f6e2205489934e498ca3ded

q = af5bdfce89d06419815ed9fa15cc2c69151b883f

g = b03dc96d2e89cd654c831f17ac7f48dc3698ee412c907f535404d5641353126a7fea98c4df06
    b3ce7ab33afa8aed93fec3f8d71420adb0bd0930bd2f2d0ff4b23b60d69e1cf0e0922d75e10f
    6ea424a76748fdbf99a8f4fddc41f5f14bef8bed830ae89e16dacd6f129e0fe66669d996059d
    92976a5d460c0c11eec6e3c342a4

a = 6ec68efd1eeb5cb127f8202cfaf4ed86ad55288c

g^a = a6b0257ccb844c3c67c7b57d6608168479e1f6dda47cc820f8e7c94e008116c7a732951318
      e0b6578531e9cd8fa8d0dec91cc66a7879e31b1c1254cb2f5bdd6112d511a0088cad981cff
```

```
ec737fcfa1ff75b4eb6eacfb7cfc22d3858b83ffab874fee15044f8c7f051a5d5500f73157
dc3399a9e5b6beee136b321c204ad54d62
```

This gives a group $\mathbb{F}_p$ and a unique subgroup of order q, generated by g, to do calculations in. a is the private key, and g^a is the public key. Note that the program does not display any of these parameters on the screen, but rather it writes them to two different files, one containing the public key and one containing the private key. The public key file contains g^a, g, p, and q. The private key file contains a, g, p and q.

Now suppose that it was desired to sign the file `msg.txt`, with the following contents:

```
In our society, an individual’s signature on a document proves that this
individual acknowledges the contents of that document, whether it a personal
letter, a credit card transaction, a contract, or any other type of document.
But how does one sign a digital document? Digital signatures should provide the
same functionality and security for digital documents as handwritten signatures
do for paper documents.

Since any digital information can be easily copied and altered, a digital
version of a handwritten signature will not do. To make digital signatures
difficult to forge, one person’s signature must be different on different
documents. So, the signature must be tied both to the signer and to the message
or document that is to be signed. Also, it is important that anyone can verify
that the signature belongs to the person it claims to belong to, and that the
signed document has not been altered after it was signed. Therefore, if any of
these conditions are violated, the signature is no longer valid.
```

Then type `dsa_sign msg.txt sig.txt` at the command line. The program creates a new file `sig.txt`, which contains the message from `msg.txt` and the signature:

```
In our society, an individual’s signature on a document proves that this
individual acknowledges the contents of that document, whether it a personal
letter, a credit card transaction, a contract, or any other type of document.
But how does one sign a digital document? Digital signatures should provide the
same functionality and security for digital documents as handwritten signatures
do for paper documents.

Since any digital information can be easily copied and altered, a digital
version of a handwritten signature will not do. To make digital signatures
difficult to forge, one person’s signature must be different on different
documents. So, the signature must be tied both to the signer and to the message
or document that is to be signed. Also, it is important that anyone can verify
that the signature belongs to the person it claims to belong to, and that the
signed document has not been altered after it was signed. Therefore, if any of
these conditions are violated, the signature is no longer valid.
```

```
-----BEGIN DSA SIGNATURE-----
50849f51f5bba5cefe60f633a05487deb2b757f1
a2c85f054575406a31f4959f15dda2b0635e1d5e
```

The first number in the signature is r, and the second number is s. To verify this signature, simply type `dsa_verify sig.txt` at the command line. The program then outputs '`Signature verified`' to the screen.

Now suppose that the file `sig.txt` was changed to look like this:

```
In our society, an individual's signature on a document proves that this
individual acknowledges the contents of that document, whether it a personal
letter, a credit card transaction, a contract, or any other type of document.
But how does one sign a digital document? Digital signatures should provide the
same functionality and security for digital documents as handwritten signatures
do for paper documents.


-----BEGIN DSA SIGNATURE-----
50849f51f5bba5cefe60f633a05487deb2b757f1
a2c85f054575406a31f4959f15dda2b0635e1d5e
```

Note that the signature has not changed, only the message. If `dsa_verify` is run again, the output will now be '`Signature NOT verified!`'

4.3 Execution times

Table 1 shows the average running times in seconds for the key generation, signature generation and signature verification programs for each of the three algorithms. The key sizes used are given in the table, and approximately the same level of security is obtained for each algorithm with these key sizes. The averages are calculated from 10 executions of each programs, and the execution times were recorded using the Unix `time` utility. These execution times were obtained on a PowerPC G3, 900MHz, with 384 MB RAM, running Linux 2.4.22. The data reflects the actual time that the given program was active on the processor, not the time from when the execution started until completion. This is because the actual elapsed time from execution start until completion depends heavily on other tasks scheduled by the operating system, and would therefore be a poor measurement for the performance of the programs.

Table 1: Execution times in seconds

Algorithm (key size)	Key generation	Signature generation	Signature verification
RSA (1024)	1.0	0.04	0.03
DSA (1024)	4.8	0.04	0.01
ECDSA (192)	0.01	0.01	0.02

5 Discussion

5.1 Implementation issues

The goal of this project was to implement the mathematical algorithms for key generation, signature generation and signature verification for several digital signature schemes. The project was not concerned with key management or local security issues. The system is therefore not ready for use in a general setting for signing text messages, but with some modifications regarding the handling of keys (both public and private), the system should be complete.

The most serious security flaw the software contains is that there is no security around who can access the private key. The file containing the private key is not password protected, and it does not have any special restrictions on which users that can access it. So if Eve could get access to the files on Bob's computer, either by physical access or via a network connection, she could get Bob's private key. The easiest way to improve this is to add password protection on the private key file. One could also change the permissions for the private key file in the operating system (of course depending on which operating system one uses), so that only the user that owns the file can read it. It will also generally increase the security to run a firewall with strict restrictions on incoming traffic.

Another deficiency is that the system does not have functionality for using more than one public key. That is, whenever a key pair is generated, the private key is stored in the private key file, and the public key is stored in the public key file,and the signature verification program will always use the public key that is stored in this specific public key file. This means that Bob can really only verify his own signatures, which is not very useful. Bob could, of course import Alice's public key file, and store it under the name that the signature verification program looks for, and thus be able to verify Alice's signatures. However, he would then have to manually change the name of his own public key file. So for this system to be useful in a general setting, it needs some functionality for storing more than one public key, and some way of deciding which public key should be used in each signature verification.

Finally, the problem of global key management has not been addressed either. For Bob to be sure that the public key he received from Alice is really Alice's public key, Bob and Alice would need to have a secure method of distributing keys.

5.2 Security issues

5.2.1 Primality tests and random number generators

To generate secure keys for RSA, one needs to be able to generate random prime numbers. For this, both a good random number generator and a reliable primality test is needed. DSA and ECDSA needs random number generators both for key generation and signature generation, and a primality test is also needed to check that the numbers p and q are primes. All three algorithms rely on being able to generate random numbers and test for primality.

GMP provides a random number generator, and in this project it uses random data from `/dev/random`, which is provided by the operating system. This provides enough randomness so that the random numbers generated will be unpredictable.

The primality test that comes with GMP implements the Miller-Rabin primality test, which is described in [Koblitz94]. This is a probabilistic primality test, so there is a possibility that it will fail to detect that a number is composite. However, whenever the primality test is used in this project, it is repeated 30 times, which implies that the probability that a composite number will not be detected is $\frac{1}{4^{50}}$. This is the level of security that NIST reccomends for primality testing in [FIPS186-2].

5.2.2 RSA

It has been conjectured, although never proven, that the security of RSA depends wholly on the problem of factoring large numbers. Therefore, the most obvious way to attack RSA is to try and factor n. The fastest factoring algorithm currently known is the Number Field Sieve (described in [NFS]), which has a subexponential running time. That is, if the number to be factored has n bits, the running time depends on $L_n(\gamma, c) = \mathrm{e}^{c(\log n)^{\gamma}(\log logn)^{1-\gamma}}$. Several factoring algorithms actually run in subexponential time, but the Number Field Sieve has the smallest γ in the expression for $L(\gamma, c)$. This method was used to factor an RSA modulus ($n = pq$) of bitlength 512 in 1999. The factorization took about 7 months using several current fast workstations and supercomputers. See [RSA–512] for more details on this factorization.

Since a 512-bit RSA modulus was factored in 1999, it is clear that a larger key size is needed today. In this project the current popular key size of 1024 (n has 1024 bits) has been used. However, [Lenstra] suggests that one should use a key size of at least 1108 bits to be guaranteed sufficient security in 2004. Nevertheless, 1024 continues to be a very popular key size, and it should still provide enough security for most applications. The programs in this project can easily be changed to use a larger key size by changing the value of one constant.

Some factoring methods may not be very efficient in general, but can still work very well for special cases. One such method is Fermat factorization (see [Koblitz94] §V.3), which factors n relatively fast if n is the product of two primes p and q such that $|p - q|$ is fairly small. To avoid easy Fermat factorization of the RSA modulus n in this project, the primes p and q are taken to have a difference in length of about 10 bits.

One should also make sure that $p - 1$ and $q - 1$ have a fairly small g.c.d., and that both have at least one large prime factor. In this project the implementation of the generation of p and q takes care of these conditions: the primes p and q are generated by first choosing a random number, multiplying this number by 2, and then adding 1 to the result. If the result of this is a prime, it is taken to be p or q. Hence $p = 2m + 1$ for some prime m, and $q = 2l + 1$ for some other prime l, so $gcd(p - 1, q - 1) = 2$, and both $p - 1$ and $q - 1$ have large prime factors (m and l).

5.2.3 DSA

The discrete logarithm problem has much in common with factoring. It appears that the same order of magnitute of time is required for solving the discrete logarithm problem in $\mathbb{F}_p$ as for factoring a number n that has the same number of bits as p. There actually exists a discrete logarithm version of the Number Field Sieve, and it runs in subexponential time. So the remarks made about key sizes for RSA also apply to the size of the prime field $\mathbb{F}_p$ for DSA. The size of the smaller subgroup in DSA also contributes towards the security of the signatures, and according to [Lenstra], the popular group size of 160 bits will still be secure until 2026. So the overall system using a field size of 1024 bits and a group size of 160 bits should still be fairly secure for most applications.

One algorithm for finding discrete logarithms is the Silver-Polling-Hellman algorithm. This method works very well in a field $\mathbb{F}_p$ if $p - 1$ is smooth, that is if all the prime factors of $p - 1$ are relatively small. However, this is not really an issue for DSA since q is a large prime factor of $p - 1$.

The primes p and q are generated according to the method specified by NIST in appendix 2 of [FIPS186-2]. This method guarantees that the primes are generated in a random fashion.

5.2.4 ECDSA

One advantage that cryptosystems over elliptic curves have over the other algorithms used in this project is that there is no known subexponential attack against elliptic curve cryptosystems. That is, the methods that exist for calculating discrete logarithms in groups of points on elliptic curves all have exponential complexity.

Unfortunately, there is one problem with ECDSA that DSA does not have, namely that calculating the order of the group generated by the point P is difficult and time-consuming. In DSA, the subgroup generated by the element g has order q by construction. However, there is no analogous way to construct a point P that has a given order q on a given elliptic curve $E(\mathbb{F}_p)$. In fact, there is no easy (and fast) way to calculate the order of a point.

One algorithm that can be used to calculate the order of a point P on an elliptic curve $E(\mathbb{F}_p)$ is the baby-step-giant-step algorithm. The idea of this algorithm is to calculate a number m in the interval $(p + 1 - 2\sqrt{p}, p + 1 + 2\sqrt{p})$ (this interval comes from Hasse's Theorem, which is stated in [Koblitz94], p. 174 and proved in §V.1 of [Silverman]) such that $mP = \mathcal{O}$. But to calculate m, a list of the first $s \approx \sqrt[4]{p}$ multiples $P, 2P, 3P, \ldots, sP$ of the point P. But when p has 192 bits, $\sqrt[4]{p}$ has approximately 48 bits. So the list will contain approximately 2^{48} points. Each point requires at least 24 bytes of memory, so this list will use at least 24×2^{48} bytes, which is approximately 2^{52} bytes (or 4 exabytes). Clearly, this is not feasible with current computer hardware.

Now, in [Schoof95], another algorithm for calculating the number of points on an elliptic curve was also presented, and this algorithm actually runs in polynomial time. However, the runtime is polynomial of degree 8, and Schoof admits that the algorithm is impractical. Several improvements have been made to the algorithm, which has improved the running time to polynomial time of degree 6. However, it would still take about 20 minutes to calculate the number of points on an elliptic curve over a field of bit size 192.

There is no easy way of calculating the order of the group generated by the point P that is needed in ECDSA. Therefore, NIST has published a list of recommended curves and field sizes for elliptic curve cryptography in appendix 6 of [FIPS186-2]. For prime fields, one curve is given for each of the field sizes of 192, 224, 256, 384 and 521 (number of bits in the field size). For each of these curves, the number of points on the curve is given, and it is always a prime number q. That means that each point on the curve will have prime order q, and each point can be used as the base point in ECDSA. NIST has also specified a sample point on each curve, so that it is not necessary to calculate a base point. For this project, the key generation program for ECDSA simply chooses the appropriate curve and point based on the given field size (which can be changed by changing a constant), generates the private key x using a random number generator, and calculates the public key $Q = xP$. In contrast, the key generation programs for RSA and DSA generate both the global parameters and the private and public keys.

One question that emerges from this method of using precomputed parameters for ECDSA is how the security compares to that of a system where these parameters are randomly generated. Now this should not be a big problem, since all the precomputed parameters are public knowledge in any ECDSA system. The only slight disadvantage one gets from the precomputed parameters is that since these curves are recommended by the US government, they are probably widely used, and one can therefore expect that more effort would go into trying and solve the discrete logarithm problem for these exact curves and points than for other curves. One way to improve security could be to actually generate a random curve and spend the time it takes to find a point of prime order, and then use this curve and point in the key generation instead of the parameters that NIST recommends.

5.3 Execution times

As mentioned above, the average execution times that are given in section 4.3 are the average times that the programs were actually active on the processor, not the overall elapsed time from the execution started until it completed. These elapsed times will be somewhat higher, and they will depend on which other jobs that are scheduled by the operating system to use the processor. A general upper bound for the elapsed times for all the programs seemed to be 30 seconds on the test computer. This is very acceptable for key sizes of the magnitude used in this project.

For RSA and DSA the key generation programs run significantly slower than the signature generation and verification programs. This is because the key generation programs actually generate large random prime numbers, and both the random number generation and the primality tests take time. However, this is acceptable because it should not be necessary to generate new keys more than once a year. The ECDSA key generation program runs very much faster than the two other key generation programs. This is simply because ECDSA uses precomputed parameters instead of generating all the parameters itself.

5.4 Future work

In the future it would be advisable to add password protection for the private key file so that the system would actually have some practical security. Also, to take the system closer to practical use, it should be expanded to be able to handle more than one public key file. And finally, the system should be integrated into a chain of Unix email utilities so that email messages could be signed and verified automatically. The programs could also be expanded to sign types of documents other than plain text messages, such as email attachments.

6 Conclusion

The three systems for digital signatures that were implemented in this project are all usable in a single user environment where only self-verification of the signature is needed. All three systems have good mathematical security, and the key sizes are as good as or better than the standards currently used in the industry.

Implementing RSA and DSA was fairly straightforward compared to ECDSA. RSA and DSA had no big implementation issues to be resolved, since they have both been widely implemented in many applications. ECDSA however, is problematic because of the difficulty of calculating the order of the group generated by the base point P on the elliptic curve. There is currently no good solution for this problem, and this will continue to be an issue for ECDSA until a solution is found. In this project, this problem was overcome by adopting elliptic curves recommended by NIST.

References

[FIPS186-2] US Department of Commerce/National Institute of Standars and Technology. "Digital Signature Standard (DSS)" (FIPS 186-2). http://csrc.nist.gov/publications/fips/fips186-2/fips186-2-change1.pdf, 2000.

[GMP] Swox AB. GNU Multiple Precision Arithmetic Library. http://www.swox.com/gmp/.

[Koblitz94] N. Koblitz. *A Course in Number Theory and Cryptography*. 2nd ed. Springer-Verlag, 1994.

[Koblitz98] N. Koblitz. *Algebraic Aspects of Cryptography*. 1st ed. Springer-Verlag, 1998.

[Lenstra] A.K. Lenstra, E.R. Verheul. "Selecting Cryptographic Key Sizes". Preprint, September 1999.

[Mhash] N. Mavroyanopoulos, S. Schumann. Mhash. http://mhash.sourceforge.net/.

[NFS] A.K. Lenstra, H.W. Lenstra, Jr., M.S. Manasse, J.M. Pollard. "The Number Field Sieve" ACM Symposium on Theory of Computing, 1990. Available at http://citeseer.ist.psu.edu/lenstra90number.html.

[RSA] R. Rivest, A. Shamir, L. Adleman. "A Method for Obtaining Digital Signatures and Public-Key Cryptosystems." *Communications of the ACM*, v.21, n. 2, Feb 1978, pp. 120-126.

[RSA–512] S. Cavallar, W.M. Lioen, H.J.J. te Riele, B. Dodson, A.K. Lenstra, P.L. Montgomery, B. Murphy et al. "Factorization of a 512-bit RSA Modulus". February 2000. Available at http://citeseer.ist.psu.edu/cavallar00factorization.html.

[Schneier96] B. Schneier. *Applied Cryptography: Protocols, Algorithms, and source code in C*. 2nd ed. John Wiley & Sons Inc., 1996.

[Schoof95] R. Schoof. "Counting Points on Elliptic Curves over Finite Fields". *Journal de Théorie des Nombres de Bordeaux*, 7:pp. 219-254, 1995. Available from http://citeseer.ist.psu.edu/schoof95counting.html.

[Silverman] J. Silverman. *The Arithmetic of Elliptic Curves*. Springer-Verlag, 1986.

A Proof of $a^{1+k\phi(n)} \equiv a \mod n$

Claim. *$a^{1+k\phi(n)} \equiv a \mod n$ for all integers a.*

Proof. To prove this, use Fermat's Little Theorem:

Theorem 1 (Fermat's Little Theorem). *Let p be a prime, and a be any integer, then $a^p \equiv a \mod p$. Now let b be any integer with $gcd(b, n) = 1$, then $b^{p-1} \equiv 1 \mod p$.*

Proof. A proof of this theorem is given in [Koblitz94], p. 20. □

Let $x \equiv 1 \mod \phi(n)$.

First consider the case where $gcd(a, n) = 1$. Then, by Fermat's Theorem,

$$a^x = a^{1+k(p-1)(q-1)} = a \times (a^{p-1})^{k(q-1)} \equiv a \times 1 \mod p \equiv a \mod p$$

$$a^x = a^{1+k(p-1)(q-1)} = a \times (a^{q-1})^{k(p-1)} \equiv a \times 1 \mod q \equiv a \mod q$$

for some positive integer k. But since $gcd(p, q) = 1$, it follows that $a^x \equiv a \mod pq \equiv a \mod n$.

Now consider the case where $p|a$, then $a = p^r b$ for some positive integer b and $r \geq 0$. Then

$$a^x = (p^r b)^x \equiv 0 \mod p \equiv p^r b \mod p \equiv a \mod p$$

And if $q \nmid b$, $a^x \equiv a \mod q$ by the argument given above, since $gcd(a, q) = 1$. But then, for $p|a$ and $q \nmid a/p$, $a^x \equiv a \mod pq \equiv a \mod n$

And if $q|a$, then $a^x \equiv 0 \mod q \equiv a \mod q$, and if $p \nmid a/q$ then $a^x \equiv a \mod q$, since $gcd(a, p) = 1$. So for $q|a$ and $p \nmid a/q$, $a^x \equiv a \mod pq \equiv a \mod n$.

Finally, if $p|a$ and $q|a$, then trivially, $a^x \equiv 0 \mod n \equiv a \mod n$.

And hence, $a^x \equiv a \mod n$ for all $a \in \mathbb{Z}$. □

B Proof that g generates a unique cyclic subgroup of order q (DSA)

Claim. *If $g = g_0^{(p-1)/q} \not\equiv 1 \mod p$ then g generates a unique subgroup of $\mathbb{F}_p^*$ of order q.*

Proof. This proof makes use of the following two theorems:

Theorem 2. *$\mathbb{F}_p^*$ is cyclic.*

Theorem 3. *A cyclic group of order n has a unique subgroup of order d for all $d|n$.*

Proof. See [Koblitz94], pp. 34-35 for a proof of both propositions. □

Now $g = g_0^{(p-1)/q}$ has order $d|q$ since $g^q = g_0^{q(p-1)/q} = g_0^{p-1} = 1$. But since q is a prime, that implies that $d = 1$ or $d = q$. However, $g \neq 1$ by construction, and hence $d \neq 1$, and so it must be that $d = q$. So then g generates a cyclic subgroup of order q. And since $\mathbb{F}_p^*$ is cyclic, this subgroup must be unique. □

C Installation instructions

Instructions on how to install on a Unix system:

1. Make sure that GMP (http://www.swox.com/gmp/) and Mhash (http://mhash.sourceforge.net/) are already installed

2. Commands:
   ```
   gunzip program.tar.gz
   tar xvf program.tar
   cd ds
   make
   cp rsa_generate rsa_sign rsa_verify /usr/local/bin
   cp dsa_generate dsa_sign dsa_verify /usr/local/bin
   cp ecc_generate ecc_sign ecc_verify /usr/local/bin
   ```

3. To run (for RSA):

   - `rsa_generate`
     - generates `rsa_private.key` and `rsa_public.key`
   - `rsa_sign <inputfile> [<outputfile>]`
     - Signs the input text file `<inputfile>` using the private key in `rsa_private.key`
     - Stores the signed message in `<inputfile>` or `<outputfile>` if given.
   - `rsa_verify <file>`
     - Verifies the signed message in `<file>` using public key in `rsa_public.key`
     - Prints out "`Signature verified`" or "`Signature NOT verified!`"

   To run the DSA programs, simply use `dsa` for `rsa` above, and to run the ECDSA programs use `ecc` in place of `rsa`.