An Elliptic Curve Digital Signature Algorithm and Diffie-Hellman Key Agreement Algorithm Implementation in PHP-OOP

Matej Danter

December 4, 2010

Abstract

Elliptic curves (EC) are smooth algebraic curves of abelian variety, which form a commutative group using the multiplication operation. Elliptic curves may be defined over a variety of fields, for example $\mathbb{R}$, $\mathbb{C}$, $\mathbb{Q}$, finite fields, and others. Elliptic curve cryptography (ECC) is a flavor of public-key cryptography based on elliptic curves over finite fields. This paper provides an introduction to elliptic curves, the elliptic curve digital signature algorithm (ECDSA), and the elliptic curve Diffie-Hellman (ECDH) key agreement scheme. Descriptions of possible attack methods are briefly discussed for EC- based cryptographic schemes. This paper provides an extensive description of a PHP-OOP implementation of ECDSA, and ECDH, based on an implementation of EC using two major multi-precision mathematics extensions (gmp and bcmath) available to PHP.

1 Introduction

This paper describes an implementation of elliptic curves and two crypto-schemes based on elliptic curves defined over finite fields of prime modulus in object oriented PHP (PHP-OOP). The implementation leverages the object- oriented syntax of PHP to clearly delineate collections of properties and operations on them. The two crypto schemes discussed in this paper and implemented are the Digital Signature Algorithm (DSA), and the Diffie-Hellman key agreement protocol(DH).

Both protocols rely on public-key cryptography which makes use of asymmetric key algorithms. Asymmetric key algorithms rely on nontrivially related key pairs, a secret private key and a public key. The DSA is a United States Federal Government Standard (FIPS) for digital signatures. The DSA was proposed by the NIST in 1991 and adopted two years later. In order to implement DSA, hashing and public-key cryptography are used. The

Diffie-Hellman Key Agreement or Key Exchange protocol is a specific method of exchanging keys and establishing a shared secret over an insecure communication infrastructure. This scheme is one of the earliest practical example of cryptographic key exchange. This protocol was invented and published in 1976 by Whitfield Diffie and Martin Hellman. This key agreement protocol makes use of public key cryptography which relies on asymmetric key algorithms. [Wade Trappe(2006)]

PHP is a language that is a popular choice for creating web applications, though it may be used for command line script development. PHP was created in 1995, and is a loosely and dynamically typed imperative language that borrows most of its syntax from C, or C++ in the case of object oriented programming. As of the writing of this paper PHP offers two mainstream multi precision mathematics extensions: GMP and bcmath. GMP stands for GNU Multiple Precision arithmetic library. This library is freely available under the GNU general public license, and is optimized for performance. Bcmath received its name from Binary Calculator which supports numbers of any size and precision, represented as strings, this implementation is PHP- specific and does not rely on an underlying system library. [Zandstra(2008)]

Section 2 of this paper provides an introduction to elliptic curves, by describing the operations and related definitions along with a brief description of hashing functions. The elliptic curve variation of public- key cryptography as it applies to DSA is described in Section 3. Section 4 describes the Diffie-Hellman key agreement protocol using elliptic curve asymmetric key algorithms. Implementation details of elliptic curves and the two crypto schemes in PHP-OOP can be found in Section 5.

Due to their cryptographic strength and compactness of the key used for them in comparison to RSA, elliptic curves are naturally applicable to domains of technology where quick computation and limited resources are available. Methods for establishing a secure shared key and verifying authenticity using ECDSA and ECDH, as applied to the web application domain are discussed in Section 6.

2 Elliptic Curves Over Finite Fields

Elliptic curves are smooth (continuously differentiable) curves of the form

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

In the real number system, the equation above may take two shapes depending on how many real roots it has. Please refer to figures 1 and 2 for the graphs with one and three roots. By the nature of elliptic curves we also include a point at ∞ (sometimes called O for order of the curve). This point is usually placed at $\pm y$ because the bottom of the y

axis is identified with the top.

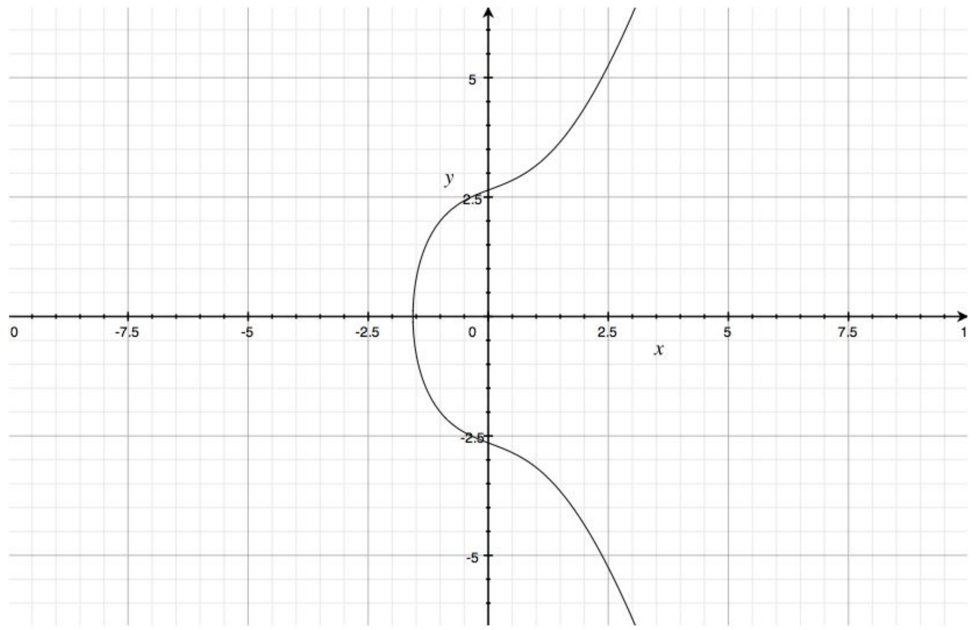

Figure 1: $y^2 = x^3 + 2x + 7$ has one root

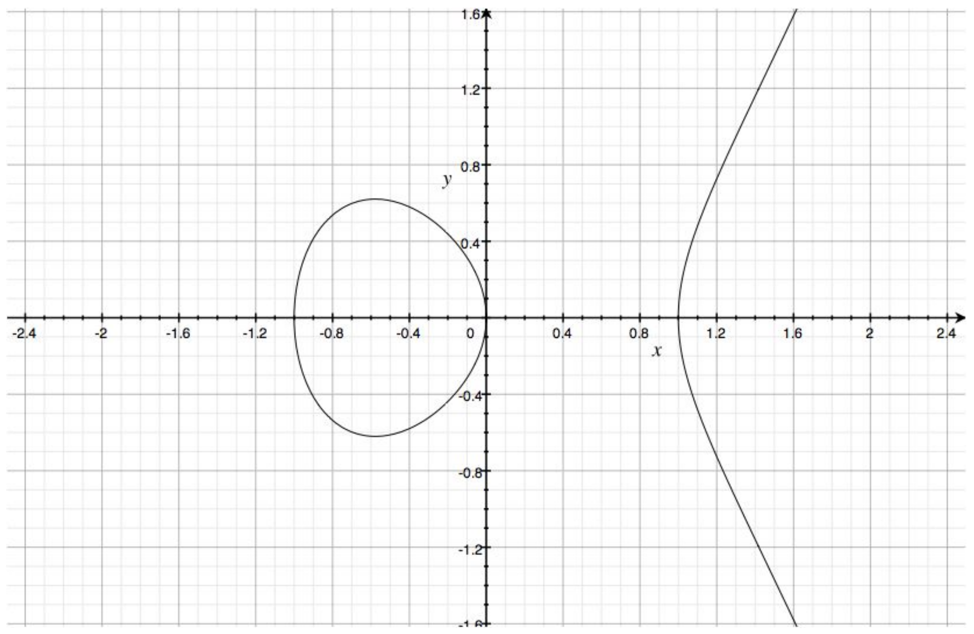

Figure 2: $y^2 = x(x+1)(x-1)$ with three roots

There is one operation defined on elliptic curves: addition. Addition on elliptic curves differs from adding points in the plane.
The law of addition on an elliptic curve is represented in Figure 3. [Washington(2003)]

Elliptic Curve Addition Law

Let an elliptic curve E be given by $y^2 = x^3 + bx + c$ and $P_1 = (x1, y1)$ and $P_2 = (x2, y2)$.

First, compute the slope of the tangent line m between P_1 and P_2 by:

$$m = \begin{cases} (y_2 - y_1)/(x_2 - x_1) & \text{if} \quad P_1 \neq P_2 \\ (3x_1^2 + b)/(2y_1) & \text{if} \quad P_1 = P_2 \end{cases}$$

We find $P_1 + P_2 = P_3 = (x_3, y_3)$

$$x_3 = m^2 - x_1 - x_2$$

$$y_3 = m(x_1 - x_3) - y_1$$

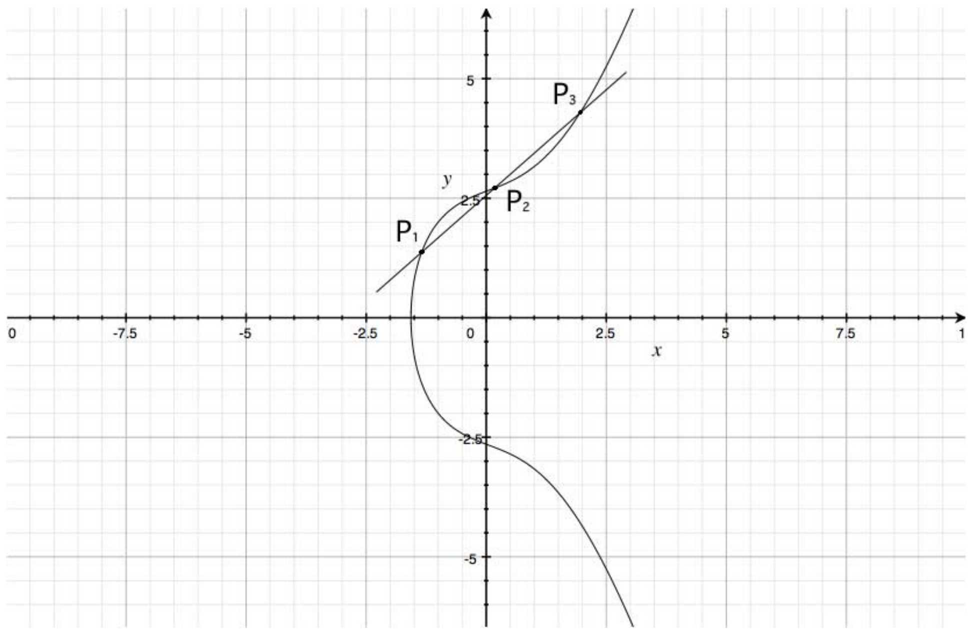

Figure 3: $P_1 + P_2 = P_3$

This addition law is associative and commutative, i.e.:

$$(P + Q) + R = P + (Q + R)$$

and

$$P + Q = Q + P.$$

Integer multiplication is defined as:

$$kP = \underbrace{(P + P + \ldots + P)}_{k\ times}$$

Additionally, elliptic curves hold identities:

$$P_1 + \infty = P_1$$

$$P_1 + -P_1 = \infty$$

This means that ∞ is the additive identity. Thus, $P + \infty = P$.

Elliptic curve cryptography is defined on fields with finitely many elements (finite fields) $(mod\ p)$ where p is a prime number, instead of being defined on $\mathbb{R}$.

The computations above are still used, but division is replaced by finding the multiplicative inverse of the divisor $(mod\, p)$, that is $a/b = a*b^{-1}$ where b^{-1} is given by $b*b^{-1} = 1(mod\, p)$. This requires that p be a prime number so that $gcd(b, p) = 1$.

An elliptic curve $(mod\ n)$, where n is a prime number, contains approximately $(n + 1)$ points (that is n points + the additive identity ∞). [Darrel Hankerson(2004)]

A more precise estimate was given by H. Hasse in the 1930s.

Hasse's theorem
Suppose E $(mod\ p)$ has N points. Then

$$|N - p - 1| < 2\sqrt(p)$$

The advantage Elliptic Curves lend to cryptography is that there is no good way to attack the elliptic curve analogous discrete logarithm problem. [Wade Trappe(2006)]

Discrete logarithm problem for elliptic curves
Suppose there exist two points P_1 and P_2 on a curve C and

$$P_2 = kP_1 = \underbrace{(P_1 + P_1 + \ldots + P_1)}_{k\ times}.$$

The objective is to find a suitable k.

There is no good attack method for the discrete logarithm problem for elliptic curves and other frequently used attack methods, such as the Pohlig-Hellman and Baby step Giant Step attacks work in special situations only, the index calculus attack becomes meaningless for the elliptic curve discrete logarithm problem.

Based on the cryptographically strong property of the discrete logarithm problem above, elliptic curves are good candidates for applications in cryptography. Elliptic curves lend themselves to well known and widely used algorithms in encryption schemes that use public-key cryptography. Two examples are the Digital Signature Algorithm (DSA) and various key agreement protocols such as the Diffie-Hellman key agreement protocol(DH). [Wade Trappe(2006)]

3 Digital Signature Algorithm and Elliptic Curves

As with any DSA algorithm, the **Elliptic Curve Digital Signature Algorithm** (ECDSA), as any DSA algorithm consists of **key generation**, **signing**, and **verification**. The key generation phase consists of two main phases. Phase 1 deals with choosing a hashing function and establishing the parameters that are publicly shared, while the second phase generates a public- private key pair. [I. Blake(1999)]

A cryptographic hash function is a function that takes any data and returns a fixed- size piece of information. Usually, well described methods are used to digest an arbitrary bit string. Hashing functions are commonly chosen to decrease the possibility of arriving at the same fixed size hash from processing two pieces of data that are not initially identical. A common class of deterministic functions for generating hashes are **one-way trapdoor functions**. One- way trapdoor functions are desirable because, while it is computationally trivial to generate a hash, once it is computed the message may only be verified by re-applying the same function to the original piece of data. This is in most cases achieved by optimizing the loss of information the data incurs while decreasing possible collisions. The situation where the different pieces of input generate the same hash is called a collision. A one-way function must have well documented properties in order to be called a cryptographic hash function. One of these is collision resistance. [Wade Trappe(2006)] Currently the most popular cryptographic hash functions belong in the SHA -1/256/224/512 family. With the exception of SHA-1, the number postfix designates the output size the cryptographic hashing function produces in bits, while SHA-1 produces 160 bit output. In the DSA key generation phase, after choosing a good candidate for a cryptographic hash function, establishing the publicly shared pieces of information will make it possible to verify the signature. When phase 1 is complete a public private key pair must be generated. This is classically carried out by using public key cryptography. Public key cryptography is also referred to as asymmetric key cryptography. Asymmetric key cryptography uses two distinct keys, one for encrypting the data, and another for decrypting it. As opposed to symmetric key cryptography the same key is used for both encryption and decryption. The most notorious asymmetric key cryptographic algorithm is RSA. This section discusses signature generation, signing and signature verification where elliptic curves are used to generate a public-private key pair in a way that is analogous to DSA.

3.1 Signature Generation Algorithm

For Alice (A) to sign and send a message to Bob (B), they must agree upon the field size, the parameters a and b that are the coefficients in $y^2 = x^3 + ax + b$, and base (generator) point G of prime order n. She also must choose d_A such that $d_A \epsilon [1, n-1]$, and a public key Q_A where $Q_A = d_A G$. [Wade Trappe(2006)]

Signature Generation Algorithm
In order to sign a message m, Alice follows the algorithm below:
1. Calculate $e = HASH(m)$, where $HASH(m)$ is an arbitrary cryptographically secure hash function.
2. Let l_0 be the leftmost bit of e.
3. Select a random integer k so that $k \epsilon [1, n-1]$.
4. Calculate $r = x_1(mod\ n)$ where $(x_1, y_1) = kG$. If $r = 0$ go back to step 3.
5. Calculate $s = k^{-1}(l_0 + rd_A)(mod\ n)$. If $s = 0$ go back to step 3.
6. The signature is (r, s).

Signature Verification Algorithm
In order for Bob (B) to successfully verify Alice's signature he must have a copy of Q_A, which is Alice's public key. To verify the authenticity of Q_A before verifying the signature, he may check three conditions:
1. $Q_A \neq O$ where O is the identity element.
2. the curve contains Q_A.
3. $nQ_A = O$.

Now to verify the signature:
1. Verify that $r, s \epsilon \mathbb{Z}$ and $r, s \epsilon [1, n-1]$.
2. Calculate $e = HASH(m)$, where the hash function is identical to the one Alice used.
3. Let l_0 be the leftmost bit of e.
4. Calculate $w = s^{-1}(mod\ n)$.
5. Calculate $u_1 = l_0 w(mod\ n)$ and $u_2 = rw(mod\ n)$.
6. Calculate $(x_1, y_1) = u_1 G + u_2 Q_A$.
7. The signature is valid if $r = x_1(mod\ n)$, invalid otherwise.

3.2 Correctness for ECDSA

Suppose a signature (r, s) on the message m is generated by Alice. Then it follows that $s = k^{-1}(HASH(m) + d_A r)\ (mod\ n)$. Thus

$$\begin{aligned} kG &= s^{-1}(HASH(m) + d_A r)G\ (mod\ n) \\ &= s^{-1}HASH(m)G + s^{-1}rd_A G\ (mod\ n) \\ &= HASH(m)wG + rwQ\ (mod\ n) \\ &= u_1 G + u_2 Q\ (mod\ n) \end{aligned}$$

Since $u_1G + u_2Q = u_1G + u_2d_AG = (u_1 + u_2d_A)G = kG$, this verifies $x_1 = r$ as required by DSA.

The main difference of ECDSA from classical DSA is the use of a different asymmetric key algorithm. Classically, the asymmetric key algorithm for signing and verification uses the usual version of the discrete logarithm problem instead of EC version.

4 Diffie Hellman Key Agreement

The Diffie-Hellman key agreement protocol is a protocol that allows two parties, each having a public-private key pair, to establish a shared secret over an untrusted or insecure communication channel. For the Elliptic Curve Diffie-Hellman (ECDH) protocol, the public-private key pairs consist of elliptic curves instead of the classical public key generation scheme using discrete logarithms. [Darrel Hankerson(2004)]

Key Establishment Protocol
In order for Alice and Bob to establish a shared key, the two parties must agree on the domain parameters, p that defines the prime field, the coefficients a, b in $y^2 = x^3 + ax + b$, the generator point G, $n \epsilon \mathbb{Z}$ such that $nG = O$, and the $cofactor\ h$.

■ Alice picks d_A randomly such that $d_A \epsilon \mathbb{Z}$ and $d_A \epsilon [1, n-1]$
■ Alice computes $Q_A = d_AG$, publishes Q_A and keeps d_A safe

Similarly

■ Bob picks d_B randomly such that $d_B \epsilon \mathbb{Z}$ and $d_B \epsilon [1, n-1]$
■ Alice computes $Q_B = d_BG$, publishes Q_B and keeps d_B safe

Alice computes
$$(x_k, y_k) = d_AQ_B$$
Bob computes
$$(x_k, y_k) = d_BQ_A$$
Alice and Bob's shared key is x_k.

$$d_BQ_A = d_AQ_B \text{ because } d_AQ_B = d_Ad_BG = d_Bd_AG = d_BQ_A$$

The Diffie-Hellman key agreement protocol is secure, because only the public keys $(Q_{A,B})$ are disclosed. Alice's ability to derive Bob's private key is analogous to solving the Elliptic Curve Discreet Logarithm Problem.

5 Elliptic Curve Cryptography PHP-OOP API

The PHP-OOP ECC library implementation consists of:
■ interfaces
■ implementing classes
■ a test suite designed to test conformance to ECDSAVS 186, and to demonstrate usage [Brassham(2004)]
■ necessary utilities for computations
■ index file taking care off class autoloading and initial environment setup in terms of setting script timeout and allowed memory limits.

The PHP-OOP ECC API was implemented to use one of the two main PHP multi precision math libraray extensions: **gmp** and **bcmath**. The implementation is set to evaluate which multi precision mathematics library is present and use the one present for ECC calculations for ECDSA and ECDH. If both **gmp** and **bcmath** are present, the implementation favors **gmp** for performance reasons. For performance statistics and hardware descriptions, please refer to Appendices A, B, C. For the source code please refer to Appendix D.

5.1 Elliptic Curve PHP Classes

Let us begin by enumerating the interfaces that the implemented source code adheres to. The interfaces are:
• CurveFpInterface represents an elliptic curve and all operations on it in a prime field.
• PointInterface represents a point and all possible operations on it.
For the interface source code please refer to Appendix D.
The implementation of the interfaces listed above is in the **classes** directory at the same level as the interface directory.
The files responsible for basic elliptic curve arithmetic and supporting operations are CurveFp.php, Point.php, NumberTheory.php, gmp_Utils.php, and bcmath_Utils.php. The methods in CurveFp.php, Point.php, NumberTheory.php contain source code implements operations using both bcmath and gmp, or reports an exception if neither multi precision mathematics library is present.

5.1.1 NumberTheory.php

This class encapsulates static methods that are concerned with computing number theoretic algorithms. The most used static methods in this class are is_prime(p), next_prime(p), and inverse_mod(a, m). the method inverse_mod is the most frequently used method and it is also the most costly in the case of bcmath, as that library has no native implementation for computing multiplicative inverses in a prime field. In the case of gmp, this operation is supported in the extension's API.

5.1.2 gmp_Utils.php

This class encapsulates static methods that are needed for elliptic curve arithmetic for the implementation using the gmp library extension, namely modular arithmetic and pseudo-random number generation. **Note:** The method gmp_mod2 had to be constructed in order to repair a bug with modular arithmetic in PHP's gmp extension. This bug may be architecture dependent.

5.1.3 bcmath_Utils.php

This class encapsulates static methods that are needed for elliptic curve arithmetic for the implementation using the bcmath library extension, namely random number generation, bitwise operations, and number base conversions.

5.1.4 CurveFp.php

The CurveFp class represents an elliptic curve over a prime field. In addition to storing the coefficients a,b and the base prime, this class verifies point containment on its instance using the **contains** public function, and contains a static method for comparing two CurveFp instances.

```
CurveFp->public function contains(x, y)
```

This method tests if the point (x,y) is contained on the curve of an instantiated CurveFp object. The method uses the line

```
gmp_cmp
(
    gmp_Utils::gmp_mod2
    (
        gmp_sub
        (
            gmp_pow(y, 2),
            gmp_add
            (
                gmp_add
                (
                    gmp_pow(x, 3),
                    gmp_mul(this->a, x)
                ),
            this->b
            )
        ),
    this->prime
    ),
0)
```

which translates to the logical comparison

$$y^2 - (x^3 + ax + b) == 0$$

which is derived from

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

If RHS = LHS, the current CurveFp object contains (x,y).

`CurveFp::public static function cmp(CurveFp cp1, CurveFp cp2)`

This static method compares cp1 and cp2 **CurveFp** instance variables to decide whether cp1 is identical to cp2.

5.1.5 Point.php

The Point class encapsulates all operations on a point on an elliptic curve. A point references a CurveFp object an (x,y) coordinate, and order associated with a point. Infinity is represented using a static variable named infinity.

`Point->public function __construct(CurveFp curve, x, y, order = null)`

The Point constructor's purpose is to check whether a point is contained on the curve that the Point object was instantiated with, and check that the order multiplied by the point results in the additive identity O if the order of a point was supplied.

`Point->public static function cmp(x1, x2)`

This static method's purpose is to compare two points. Point equality means that both x1 and x2 are of type **Point**; moreover, x1's values match x2's values along with their **CurveFp** values as demonstrated below.

```
if ((x1->x == x2->x) && (x1->y == x2->y)
                                      && CurveFp::cmp(x1->curve, x2->curve))
{
      return 0;
} else {
      return 1;
}
```

The three methods below are of key importance for elliptic curve arithmetic. By the nature of elliptic curve arithmetic, integer multiplication of a point means

$$kP_1 = \underbrace{(P_1 + P_1 + \ldots + P_1)}_{k\ times}.$$

where $k \epsilon \mathbb{Z}$.
Elliptic curve multiplication takes place in mul using the double, add, and leftmost_bit static methods.

```
Point->public static function add(p1, p2)
```

add expects two point instances p_1 and p_2 as inputs.
$p_1 = (x_1, y_1),\ p_2 = (x_2, y_x)$

point addition algorithm
The algorithm below implements elliptic curve point addition where $P_1 \neq P_2$

$$m = (y_2 - y_1)/(x_2 - x_1)$$

$$x_3 = m^2 - x_1 - x_2$$

$$y_3 = m(x_1 - x_3) - y_1$$

```
if (the curve for p1 matches the curve for p2) {
    if (x1 =  x2 mod p) {
     //at this point the y values my either be identical or opposite in sign
        if (y1=-y2 mod p) {
            return infinity
        } else {
            //call double instead of add
            return double
        }
    }
    //if x1 != x2
    p = p1->curve->getPrime();
    //compute the slope according to the formula where P1 != P2
    l = (p2->y - p1->y) * inverse_mod(p2->x - p1->x, p);

    x3 = (l^2 - p1->x - p2->x) mod p;

    y3 = (l * (p1->x - x3) - p1->y) mod p;
    p3 = new Point(x1->curve, x3, y3);

    return p3;
} else {
    //The Elliptic Curves do not match
}
```

```
Point->public static function double(Point x1)
```

double expects two point instances p_1 and p_2 as inputs.
$p_1 = (x_1, y_1),\ p_2 = (x_2, y_x)$

point doubling algorithm
The algorithm below implements elliptic curve point doubling i.e.($P_1 = P_2$)

$$m = (3x_1^2 + b)/(2y_1)$$

$$x_3 = m^2 - x_1 - x_2$$

$$y_3 = m(x_1 - x_3) - y_1$$

```
//p is the curve's prime base
//a is p1's a coefficient
//l is the slope
inverse = inverse_mod((2 * p1->y), p);
three_x2 = 3 * ($p1->x^ 2);
l = ((three_x2 + a) * inverse) mod p;
x3 = ((l^2) - (2 * p1->x)) mod p;
y3 = ((l * (p1->x - x3)) - p1->y) mod p;
if (0 > y3)
    y3 = p + y3;
p3 = new Point(p1->curve, x3, y3);
return p3;
```

```
Point->public static function mul(x2, Point x1)
```

The point multiplication algorithm
This is the classical double-and-add algorithm. The double-and-add method using $k * P$ where $k \epsilon \mathbb{Z}$ and P a point on an elliptic curve consists of decomposing k such that $k = k_0 + 2k_1 + 2^2k_2 + \ldots + 2^jk_j$. Classically, the double-and-add method may be described as

```
P a point
Q a point, the result of k*P
k element of Z
Q=P
for i from j to 0 do
    Q = 2Q    //double Q
    if k_i = 1 then Q := Q+P
return Q
```

This algorithm is modified below to account for positive and negative y values during scalar multiplication.

```
e = k;
if (cmp(p1, infinity) == 0) {
    return infinity
}
if (p1->order != null) {
    e = e mod p1->order
}
if (e == 0) {
    return infinity
}

if (e > 0) {
    e3 = 3 * e
    negative_self = new Point(p1->curve, p1->x, - p1->y, p1->order)
    i = leftmost_bit(e3) / 2
    result = p1
    while (i > 1) {
      result = double(result)
      if ( (e3 & i) != 0 && (e & i) == 0) {
        result = add(result, p1)
      }
      if ((e3 & i) == 0 && (e & i) != 0) {
        result = add(result, negative_self)
      }
      i = (i / 2)
    }
}
return result

Point->public static function leftmost_bit(x)
```

During each iteration in the scalar multiplication algorithm, this method is used to determine whether double, or double-and-add is performed.

```
if (x > 0) {
    result = 1;
    while (result <= x ) {
        result = 2 * result;
    }
    return result/2;
}
```

5.2 Elliptic Curve Digital Signature Algorithm

• PrivateKeyInterface represents signature generation for ECDSA.
• PublicKeyInterface represents signature verification for ECDSA.
• SignatureInterface represents signature pair encapsulation for ECDSA.

5.2.1 PrivateKey.php

The PrivateKey class implements all necessary operations for signature generation. For a reference on the ECDSA algorithm please refer to Section 2.

Constructor
A PrivateKey instance requires the PublicKey, which is a point on an elliptic curve, and a secret (a randomly selected integer mod n).

ECDSA Message Signing
The algorithm of sign adheres to the one in Section 2. Upon successfully signing a hash a new Signature object is returned.

```
public function sign(hash, random_k) {
    G = this->public_key->getGenerator()
    n = G->getOrder()
    k = random_k mod n
    p1 = Point::mul(k, G)
    r = p1->getX()

    if (r == 0) {
      report error random number r = 0
    }
    s = mod((inverse_mod(k, n) * mod((hash + (this->secret_multiplier * r)), n)), n)
    if (s == 0) {
      report error random number s = 0
    }
    return new Signature(r, s)
}
```

Hashing Function for the Message Digest Stage of ECDSA
ECDSA typically uses SHA1 for digesting a message, but other one-way functions may be used.

```
public static function digest_integer(m) {
    return string_to_int(hash('sha1', self::int_to_string(m)))
}
```

Checking point validity
Section 2.1 discusses checking point validity to determine the authenticity of the signature. The algorithm implements the 3 conditions for authenticity checking.

```
public static function point_is_valid(Point generator, x, y) {
n = generator->getOrder();
curve = generator->getCurve();

if (x < 0 || n <= x || y < 0 || n < y <= 0)
    return false

containment = curve->contains(x, y)
if (!containment)
    return false

point = new Point(curve, x, y)
op = Point::mul(n, point)

if (!(Point::cmp(op, Point::infinity) == 0))
    return false

return true
```

String to Integer Conversion
The helper functions below are necessary for message digestion. The purpose of the methods is to compute the integer representation of an ASCII string, and vice-versa. Please refer to Appendix D for implementation.

```
public static function int_to_string(x)
public static function string_to_int(x)
```

5.2.2 PublicKey.php

The PublicKey class implements all necessary operations for signature verification. For a reference on the ECDSA algorithm please refer to Section 2.

Constructor
A PublicKey instance requires a generator point to access the order of the generator and the public key which is a point on an elliptic curve.

ECDSA Message Verification
The algorithm of **verifies** adheres to the one in Section 2.1. Upon verification a boolean value is returned based on the validity of the signature.

```
public function verifies(hash, Signature signature) {
G = this->generator;
n = this->generator->getOrder();
point = this->point;
r = signature->getR();
s =signature->getS();

if (r < 1|| r > (n - 1)))
    return false

if (s < 1 || s > (n - 1))
    return false

c = inverse_mod(s, n)
u1 = hash *c mod n
u2 = r * c mod n
xy = Point::add(Point::mul(u1, G), Point::mul(u2, point))
v = xy->getX() mod n

if (v == r)
    return true
else
    return false
```

5.2.3 Signature.php

The signature class serves as the encapsulation of the signature tuple (r, s). It is a PHP object, containing r,s as private variables with publicly accessible setters and getters. This class is used to store the signature meaningfully while it is being passed between the PublicKey and PrivateKey classes.

5.3 Elliptic Curve Diffie-Hellman Key Agreement

• EcDHInterface represents all operations involved in EcDH for dual key encryption along with methods that utilize ECDH as a dual key encryption scheme, namely:

```
public function __construct(Point g);

public function calculateKey();

public function getPublicPoint();

public function setPublicPoint(Point q);

public function encrypt(string);

public function decrypt(string);

public function encryptFile(path);

public function decryptFile(path);
```

5.3.1 EcDH.php

The class has 5 instance variables:
■ for storing the generator point
■ for storing an object instance's own public Point
■ for storing a public point received from another object instance
■ the private key associated with the current object instance
■ the shared key as a product of successful key agreement using ECDH.
Interaction between Alice and Bob to establish a shared secret is of the form:

```
g = NISTcurve::generator_192();
Alice = new EcDH(g);
Bob = new EcDH(g);

//Alice and bob generate their private keys and public Point
pubPointA = Alice->getPublicPoint();
```

```
pubPointB = Bob->getPublicPoint();

//Alice sends Bob her public key and vice versa
Alice->setPublicPoint(pubPointB);
Bob->setPublicPoint(pubPointA);

//key_A == key_B
key_A = Alice->calculateKey();
key_B = Bob->calculateKey();
```

Description of Algorithms

`public function __construct(Point g);`

The constructor serves only to populate the generator point for ECDH.

`public function getPublicPoint();`

When called, this method generates a pseudo-random number ”secret” between 0 and the order of the generator point. Upon successful generation it returns the public point secret*Generator.

`public function setPublicPoint(Point q);`

This method is used to set a public point that was received from the person wishing to establish secure communication through an untrusted channel.

`public function calculateKey();`

The calculateKey method’s purpose is to finally establish the shared secret. When both parties call this method, they both arrive at the same key. Find the explanation of correctness behind ECDH in Section 3.

`public function encrypt(string);`

`public function decrypt(string);`

`public function encryptFile(path);`

`public function decryptFile(path);`

The methods above are demonstrations of how a dual-key encryption scheme may be used with the ECDH shared key agreement protocol. The first pair of methods may be used for arbitrary strings. For hashing the x value of the generated shared secret, the implementations use SHA256. This hash is then fed into the 256 bit EAS encryption function in CBC mode. The methods which operate on files are identical to the previous two, but are designed to read in a file as the message to be communicated between Alice and Bob.
Note: The dual key encryption/decryption algorithm uses AES 256 for two way encryption. This may be substituted by other two way encryption functions as long as the function used to encrypt the message is identical to the one used to decrypt the message.

5.4 Test Suite and Related Source Code

The PHP-OOP includes a Test Suite, along with a class encapsulating NIST published safe curves at various key-lengths.

5.4.1 TestSuite.php

The TestSuite class may be run in verbose or non-verbose mode. It was written to test the algorithms located in the NumberTheory class, to check NIST curve validity checking according to (X9.62) [Brassham(2004)], Point Validity testing accoding to Appendix B.2.2 of [Brassham(2004)], signature validity testing according to Appendix B.2.4 of [Brassham(2004)], and perform a Diffie-Hellman Key Agreement Protocol test.

5.4.2 NISTCurve.php

The NISTCurve class is an encapsulation of static methods that contains all NIST published elliptic curves that are approved as secure for elliptic curve cryptography. [NIST(2010a)] [NIST(2010b)]

6 Uses for Elliptic Curve Cryptography

The fact that the discreet logarithm for elliptic curves problem is difficult to solve means that cryptography based on elliptic curves is secure. As in any public key crypto system, the private key should be kept secret or communicated through a secure channel, in order to maintain the security asymmetric key cryptography provides.

The strength of crypto-schemes using elliptic curves are on par with RSA with the advantage that a smaller key may be used to achieve the same level of encryption strength as RSA does. In elliptic curve cryptography, the key size used does not scale linearly with cryptographic strength as in the case of RSA. For example, the EC key size equivalent of 1024 bit RSA encryption is 160 bits, while doubling the key size of RSA to 2048 bits equates to increasing the key size to 224 bits using ECC. [NSA(2009)] This is beneficial in terms of computation, data transmission, and in domains where space is a premium, such as mobile devices, RFID tags, etc.

Elliptic Curve DSA is useful in applications in which other DSA algorithms were used previously, with the added benefit of smaller key sizes. The uses for digital signatures in general are in the areas of authentication and identity establishment, in message integrity verification and non-repudiation. In the area of authentication, digital signatures are used to authenticate the source of the messages that initiate authentication requests. [NSA(2009)] Digital Signatures may be used along with two-way encryption. This is desirable because even though messages are encrypted between two parties, it is possible to alter the encrypted representation of the message. By augmenting two way encryption with digital signatures, it is possible to verify if the original message was altered. Another important use for digital signatures is non-repudiation of origin, where an entity initially signing a message is unable to later deny having signed the message. ECDSA lends smaller key lengths to the use of DSA with higher level of security compared to RSA if identical length keys are used.

The Diffie-Hellman key agreement protocol is used in everyday applied computer science in TLS (Transport Layer Security) to provide perfect forward secrecy. TLS is a ubiquitous method for perfect forward secrecy, appearing in the HTTPS protocol, as well as other software, such as the OpenSSH project. [OpenBSD(2010)] Also, because PHP is mostly a domain-specific language which focuses on the World Wide Web as its domain, having an application level protocol enables developers to employ ECDH as a means to achieve secure communication via an unsecured channel, without third party validation. The lack of third party validation would make the same assumption that OpenSSH makes, namely that the two parties identity is established at the first time of contact. In the case of secure web services communication, the crypto scheme similar to OpenSSH may be used. The advantage this provides to web service level communication is that the web services may not have to communicate through HTTPS, but are able to achieve encrypted communication among them that is on par or exceeds the current mainstream RSA key lengths used as HTTPS certificates.

References

[Wade Trappe(2006)] L. C. W. Wade Trappe, *Introduction to Cryptography with Coding Theory*, second edition ed., S. Yagan, Ed. Pearson, 2006.

[Zandstra(2008)] M. Zandstra, *PHP Objects, Patterns, and Practice*, second edition, Ed. Apress, 2008.

[Washington(2003)] L. Washington, *Elliptic Curves: Number Theory and Cryptography.* Chapman & Hall / CRC, 2003.

[Darrel Hankerson(2004)] S. V. Darrel Hankerson, Alfred Menezes, *Guide to Elliptic Curve Cryptography.* Springer, 2004.

[I. Blake(1999)] N. S. I. Blake, G. Seroussi, *Elliptic Curves in Cryptography.* Cambridge University Press, 1999.

[Brassham(2004)] L. E. Brassham, *The Elliptic Curve Digital Signature Algorithm Validation System (ECDSAVS).* NIST, 2004. [Online]. Available: http://csrc.nist.gov/groups/STM/cavp/documents/dss/ECDSAVS.pdf

[NIST(2010a)] NIST. (2010) Fips 186-2. digital signature standard (dss). [Online]. Available: http://csrc.nist.gov/publications/fips/archive/fips186-2/fips186-2.pdf

[NIST(2010b)] ——. (2010) Digital signature standard (dss). [Online]. Available: http://csrc.nist.gov/publications/fips/fips186-3/fips_186-3.pdf

[NSA(2009)] NSA. (2009, January) The case for elliptic curve cryptography. [Online]. Available: http://www.nsa.gov/business/programs/elliptic_curve.shtml

[OpenBSD(2010)] OpenBSD. (2010, 11) Openssh keeping your communities secret. [Online]. Available: http://www.openssh.com/

7 Appendix A: Hardware

The gmp and bcmath tests were run on the follwing hardware:

```
Processor Name:Intel Core 2 Duo
Processor Speed: 2.4 GHz
Number Of Processors: 1
Total Number Of Cores: 2
L2 Cache: 4 MB
Memory: 4 GB
Bus Speed: 800 MHz
```

8 Appendix B: Test Suite Performance using GMP

```
--------- START NEXT PRIME TEST ---------

Next prime took: 0.0036690235137939 seconds.

--------- END NEXT PRIME TEST ---------

--------- START SQUARE ROOT MOD P TEST ---------

Square roots mod 31 took: 0.12426710128784 seconds.

--------- END SQUARE ROOT MOD P TEST ---------

--------- START MULTIPLICATIVE INVERSE MOD P TEST ---------

Multiplicative inverse mod arbitrary primes took: 1.1855871677399 seconds.

--------- END MULTIPLICATIVE INVERSE TEST ---------

--------- START ELLIPTIC CURVE ARITHMETIC TEST ---------

Elementary EC arithmetic took: 0.0085620880126953 seconds.

--------- END ELLIPTIC CURVE ARITHMETIC TEST ---------

--------- START NIST PUBLISHED CURVES TEST ---------

NIST curve validity checking (X9.62) took: 0.2227098941803 seconds.

--------- END NIST PUBLISHED CURVES TEST ---------

--------- START POINT VALIDITY TEST ---------

Point validity testing (ECDSAVS.pdf B.2.2) took: 0.19945812225342 seconds.

--------- END POINT VALIDITY TEST ---------

--------- START SIGNATURE VALIDITY TEST ---------

Signing and verification tests from ECDSAVS.pdf B.2.4 took: 3.7693610191345 seconds.
```

```
--------- END SIGNATURE VALIDITY TEST ---------

--------- START DIFFIE HELLMAN KEY EXCHANGE TEST ---------

Diffie Hellman Dual Key Agreement encryption took: 0.31024312973022 seconds.

--------- END DIFFIE HELLMAN KEY EXHANGE TEST ---------

TEST SUITE TOTAL TIME : 5.8242769241333 seconds.
```

9 Appendix C: Test Suite Performance using bcmath

```
--------- START NEXT PRIME TEST ---------

Next prime took: 0.17068219184875 seconds.

--------- END NEXT PRIME TEST ---------

--------- START SQUARE ROOT MOD P TEST ---------

Square roots mod 31 took: 0.014307022094727 seconds.

--------- END SQUARE ROOT MOD P TEST ---------

--------- START MULTIPLICATIVE INVERSE MOD P TEST ---------

Multiplicative inverse mod arbitrary primes took: 5.2367839813232 seconds.

--------- END MULTIPLICATIVE INVERSE TEST ---------

--------- START ELLIPTIC CURVE ARITHMETIC TEST ---------

Elementary EC arithmetic took: 0.22237086296082 seconds.

--------- END ELLIPTIC CURVE ARITHMETIC TEST ---------

--------- START NIST PUBLISHED CURVES TEST ---------

NIST curve validity checking (X9.62) took: 95.086047887802 seconds.

--------- END NIST PUBLISHED CURVES TEST ---------

--------- START POINT VALIDITY TEST ---------

Point validity testing (ECDSAVS.pdf B.2.2) took: 94.691728115082 seconds.

--------- END POINT VALIDITY TEST ---------

--------- START SIGNATURE VALIDITY TEST ---------

Signing and verification tests from ECDSAVS.pdf B.2.4 took: 1514.5250930786 seconds.
```

```
--------- END SIGNATURE VALIDITY TEST ---------

--------- START DIFFIE HELLMAN KEY EXCHANGE TEST ---------

Diffie Hellman Dual Key Agreement encryption took: 80.673988103867 seconds.

--------- END DIFFIE HELLMAN KEY EXHANGE TEST ---------

TEST SUITE TOTAL TIME : 1790.6268291473 seconds.
```

10 Appendix D: Source Code