# Part 1

# Protocols

CHAPTER I

# Elliptic Curve Based Protocols

N.P. Smart

## I.1. Introduction

In this chapter we consider the various cryptographic protocols in which elliptic curves are primarily used. We present these in greater detail than in the book [**ECC**] and focus on their cryptographic properties. We shall only focus on three areas: signatures, encryption and key agreement. For each of these areas we present the most important protocols, as defined by various standard bodies.

The standardization of cryptographic protocols, and elliptic curve protocols in particular, has come a long way in the last few years. Standardization is important if one wishes to deploy systems on a large scale, since different users may have different hardware/software combinations. Working to a well-defined standard for any technology aids interoperability and so should aid the takeup of the technology.

In the context of elliptic curve cryptography, standards are defined so that one knows not only the precise workings of each algorithm, but also the the format of the transmitted data. For example, a standard answers such questions as

- In what format are finite field elements and elliptic curve points to be transmitted?
- How are public keys to be formatted before being signed in a certificate?
- How are conversions going to be performed between arbitrary bit strings to elements of finite fields, or from finite field elements to integers, and vice versa?
- How are options such as the use of point compression, (see [**ECC**, Chapter VI]) or the choice of curve to be signalled to the user?

A number of standardization efforts have taken place, and many of these reduce the choices available to an implementor by recommending or mandating certain parameters, such as specific curves and/or specific finite fields. This not only helps aid interoperability, it also means that there are well-defined sets of parameter choices that experts agree provide a given security level. In addition, by recommending curves it means that not every one who wishes to deploy elliptic curve based solutions needs to implement a point counting method like those in Chapter VI or [**ECC**, Chapter VII]. Indeed, since many curves occur in more than one standard, if one selects a curve from the intersection then, your system will more likely interoperate with people who follow a different standard from you.

Of particular relevance to elliptic curve cryptography are the following standards:

- **IEEE 1363**: This standard contains virtually all public-key algorithms. In particular, it covers ECDH, ECDSA, ECMQV and ECIES, all of which we discuss in this chapter. In addition, this standard contains a nice appendix covering all the basic number-theoretic algorithms required for public-key cryptography.
- **ANSI X9.62 and X9.63**: These two standards focus on elliptic curves and deal with ECDSA in X9.62 and ECDH, ECMQV and ECIES in X9.63. They specify both the message formats to be used and give a list of recommended curves.
- **FIPS 186.2**: This NIST standard for digital signatures is an update of the earlier FIPS 186 [**FIPS 186**], which details the DSA algorithm only. FIPS 186.2 specifies both DSA and ECDSA and gives a list of recommended curves, which are mandated for use in U.S. government installations.
- **SECG**: The SECG standard was written by an industrial group led by Certicom. It essentially mirrors the contents of the ANSI standards but is more readily available on the Web, from the site

  http://www.secg.org/
- **ISO**: There are two relevant ISO standards: ISO 15946-2, which covers ECDSA and a draft ISO standard covering a variant of ECIES called ECIES-KEM; see [**305**].

## I.2. ECDSA

ECDSA is the elliptic curve variant of the Digital Signature Algorithm (DSA) or, as it is sometimes called, the Digital Signature Standard (DSS). Before presenting ECDSA it may be illustrative to describe the original DSA so one can see that it is just a simple generalization.

In DSA one first chooses a hash function $H$ that outputs a bit-string of length $m$ bits. Then one defines a prime $q$, of over $m$ bits, and a prime $p$ of $n$ bits such that

- $q$ divides $p-1$.
- The discrete logarithm problem in the subgroup of $\mathbb{F}_p$ of order $q$ is infeasible.

With current techniques and computing technology, this second point means that $n$ should be at least 1024. Whilst to avoid birthday attacks on the hash function one chooses a value of $m$ greater than 160.

One then needs to find a generator $g$ for the subgroup of order $q$ in $\mathbb{F}_p^*$. This is done by generating random elements $h \in \mathbb{F}_p^*$ and computing

$$g = h^{(p-1)/q} \pmod{p}$$

until one obtains a value of $g$ that is not equal to 1. Actually, there is only a $1/q$ chance of this not working with the first $h$ one chooses; hence finding a generator $g$ is very simple.

Typically with DSA one uses SHA-1 [**FIPS 180.1**] as the hash function, although with the advent of SHA-256, SHA-384 and SHA-512 [**FIPS 180.2**] one now has a larger choice for larger values of $m$.

The quadruple $(H, p, q, g)$ is called a set of domain parameters for the system, since they are often shared across a large number of users, e.g. a user domain. Essentially the domain parameters define a hash function, a group of order $q$, and a generator of this group.

The DSA makes use of the function

$$f : \begin{cases} \mathbb{F}_p^* & \longrightarrow & \mathbb{F}_q \\ x & \longmapsto & x \pmod{q}, \end{cases}$$

where one interprets $x \in \mathbb{F}_p^*$ as an integer when performing the reduction modulo $q$. This function is used to map group elements to integers modulo $q$ and is often called the conversion function.

As a public/private-key pair in the DSA system one uses $(y, x)$ where

$$y = g^x \pmod{p}.$$

The DSA signature algorithm then proceeds as follows:

ALGORITHM I.1: **DSA Signing**

```
INPUT:  A message $m$ and private key $x$.
OUTPUT: A signature $(r, s)$ on the message $m$.
1.  Choose $k \in_R \{1, \ldots, q-1\}$.
2.  $t \leftarrow g^k \pmod{p}$.
3.  $r \leftarrow f(t)$.
4.  If $r = 0$ then goto Step 1.
5.  $e \leftarrow H(m)$
6.  $s \leftarrow (e + xr)/k \pmod{q}$
7.  If $s = 0$ then goto Step 1.
8.  Return $(r, s)$.
```

The verification algorithm is then given by

Algorithm I.2: **DSA Verification**

```
INPUT:  A message $m$, a public key $y$ and a signature $(r, s)$.
OUTPUT: Reject or Accept.
1.  Reject if $r, s \notin \{1, \ldots, q-1\}$.
2.  $e \leftarrow H(m)$.
3.  $u_1 \leftarrow e/s \pmod{q}$, $u_2 \leftarrow r/s \pmod{q}$.
4.  $t \leftarrow g^{u_1} y^{u_2} \pmod{p}$.
5.  Accept if and only if $r = f(t)$.
```

For ECDSA, the domain parameters are given by $(H, K, E, q, G)$, where $H$ is a hash function, $E$ is an elliptic curve over the finite field $K$, and $G$ is a point on the curve of prime order $q$. Hence, the domain parameters again define a hash function, a group of order $q$, and a generator of this group. We shall always denote elliptic curve points by capital letters to aid understanding. With the domain parameters one also often stores the integer $h$, called the cofactor, such that

$$\#E(K) = h \cdot q.$$

This is because the value $h$ will be important in other protocols and operations, which we shall discuss later. Usually one selects a curve such that $h \leq 4$.

The public/private-key pair is given by $(Y, x)$, where

$$Y = [x]G,$$

and the role of the function $f$ is taken by

$$f : \begin{cases} E & \longrightarrow & \mathbb{F}_q \\ P & \longmapsto & x(P) \pmod{q}, \end{cases}$$

where $x(P)$ denotes the $x$-coordinate of the point $P$ and we interpret this as an integer when performing the reduction modulo $q$. This interpretation is made even when the curve is defined over a field of characteristic two. In the case of even characteristic fields, one needs a convention as to how to convert an element in such a field, which is usually a binary polynomial $g(x)$, into an integer. Almost all standards adopt the convention that one simply evaluates $g(2)$ over the integers. Hence, the polynomial

$$x^5 + x^2 + 1$$

is interpreted as the integer 37, since

$$37 = 32 + 4 + 1 = 2^5 + 2^2 + 1.$$

The ECDSA algorithm then follows immediately from the DSA algorithm as:

**Algorithm I.3: ECDSA Signing**

```
INPUT:  A message $m$ and private key $x$.
OUTPUT: A signature $(r, s)$ on the message $m$.
1.  Choose $k \in_R \{1, \dots, q-1\}$.
2.  $T \leftarrow [k]G$.
3.  $r \leftarrow f(T)$.
4.  If $r = 0$ then goto Step 1.
5.  $e \leftarrow H(m)$
6.  $s \leftarrow (e + xr)/k \pmod{q}$.
7.  If $s = 0$ then goto Step 1.
8.  Return $(r, s)$.
```

The verification algorithm is then given by

**Algorithm I.4: ECDSA Verification**

```
INPUT:  A message $m$, a public key $Y$ and a signature $(r, s)$.
OUTPUT: Reject or Accept.
1.  Reject if $r, s \notin \{1, \dots, q-1\}$.
2.  $e \leftarrow H(m)$.
3.  $u_1 \leftarrow e/s \pmod{q}$, $u_2 \leftarrow r/s \pmod{q}$.
4.  $T \leftarrow [u_1]G + [u_2]Y$.
5.  Accept if and only if $r = f(T)$.
```

One can show that ECDSA is provably secure, assuming that the elliptic curve group is modelled in a generic manner and $H$ is a "good" hash function; see Chapter II for details.

An important aspect of both DSA and ECDSA is that the ephemeral secret $k$ needs to be truly random. As a simple example of why this is so, consider the case where someone signs two different messages, $m$ and $m'$, with the same value of $k$. The signatures are then $(r, s)$ and $(r', s')$, where

$$\begin{aligned} r &= r' = f([k]G); \\ s &= (e + xr)/k \pmod{q}, \text{ where } e = H(m); \\ s' &= (e' + xr)/k \pmod{q}, \text{ where } e' = H(m'). \end{aligned}$$

We then have that

$$(e + xr)/s = k = (e' + xr)/s' \pmod{q}.$$

In which case we can deduce

$$xr(s' - s) = se' - s'e,$$

and hence

$$x = \frac{se' - s'e}{r(s' - s)} \pmod q.$$

So from now on we shall assume that each value of $k$ is chosen at random.

In addition, due to a heuristic lattice attack of Howgrave-Graham and Smart [**174**], if a certain subset of the bits in $k$ can be obtained by the attacker, then, over a number of signed messages, one can recover the long term secret $x$. This leakage of bits, often called *partial key exposure*, could occur for a number of reasons in practical systems, for example, by using a poor random number generator or by side-channel analysis (see Chapter IV for further details on side-channel analysis). The methods of Howgrave-Graham and Smart have been analysed further and extended by Nguyen and Shparlinski (see [**261**] and [**262**]). Another result along these lines is the attack of Bleichenbacher [**31**], who shows how a small bias in the random number generator, used to produce $k$, can lead to the recovery of the long-term secret $x$.

## I.3. ECDH/ECMQV

Perhaps the easiest elliptic curve protocol to understand is the elliptic curve variant of the Diffie–Hellman protocol, ECDH. In this protocol two parties, usually called Alice and Bob, wish to agree on a shared secret over an insecure channel. They first need to agree on a set of domain parameters $(K, E, q, h, G)$ as in our discussion on ECDSA. The protocol proceeds as follows:

$$\begin{array}{ccc} \texttt{Alice} & & \texttt{Bob} \\ a & \xrightarrow{[a]G} & [a]G \\ [b]G & \xleftarrow{[b]G} & b \end{array}$$

Alice can now compute

$$K_A = [a]([b]G) = [ab]G$$

and Bob can now compute

$$K_B = [b]([a]G) = [ab]G.$$

Hence $K_A = K_B$ and both parties have agreed on the same secret key. The messages transferred are often referred to as *ephemeral public keys*, since they are of the form of discrete logarithm based public keys, but they exist for only a short period of time.

Given $[a]G$ and $[b]G$, the problem of recovering $[ab]G$ is called the Elliptic Curve Diffie–Hellman Problem, ECDHP. Clearly, if we can solve ECDLP then we can solve ECDHP; it is unknown if the other implication holds. A proof of equivalence of the DHP and DLP for many black box groups follows from the work of Boneh, Maurer and Wolf. This proof uses elliptic curves in a crucial way; see [**ECC**, Chapter IX] for more details.

The ECDH protocol has particularly small bandwidth if point compression is used and is very efficient compared to the standard, finite field based, Diffie–Hellman protocol.

The Diffie–Hellman protocol is a two-pass protocol, since there are two message flows in the protocol. The fact that both Alice and Bob need to be "online" to execute the protocol can be a problem in some situations. Hence, a one-pass variant exists in which only Alice sends a message to Bob. Bob's ephemeral public key $[b]G$ now becomes a long-term static public key, and the protocol is simply a mechanism for Alice to transport a new session key over to Bob.

Problems can occur when one party does not send an element in the subgroup of order $q$. This can either happen by mistake or by design. To avoid this problem a variant called cofactor Diffie–Hellman is sometimes used. In cofactor Diffie–Hellman the shared secret is multiplied by the cofactor $h$ before use, i.e., Alice and Bob compute

$$K_A = [h]([a]([b]G)) \text{ and } K_B = [h]([b]([a]G)).$$

The simplicity of the Diffie–Hellman protocol can however be a disguise, since in practice life is not so simple. For example, ECDH suffers from the man-in-the-middle attack:

$$\begin{tikzcd}
\texttt{Alice} & \texttt{Eve} & \texttt{Bob} \\
a \arrow[r, "{[a]G}"] & {[a]G} & \\
{[x]G} & x \arrow[l, "{[x]G}"'] & \\
 & y \arrow[r, "{[y]G}"] & {[y]G} \\
 & {[b]G} & b \arrow[l, "{[b]G}"']
\end{tikzcd}$$

In this attack, Alice agrees a key $K_A = [a]([x]G)$ with Eve, thinking it is agreed with Bob, and Bob agrees a key $K_B = [b]([y]G)$ with Eve, thinking it is agreed with Alice. Eve can now examine communications as they pass through her by essentially acting as a router.

The problem is that when performing ECDH we obtain no data-origin authentication. In other words, Alice does not know who the ephemeral public key she receives is from. One way to obtain data-origin authentication is to sign the messages in the Diffie–Hellman key exchange. Hence, for example, Alice must send to Bob the value

$$([a]G, (r, s)),$$

where $(r, s)$ is her ECDSA signature on the message $[a]G$.

One should compare this model of authenticated key exchange with the traditional form of RSA-based key transport, as used in SSL. In RSA-based key transport, the RSA public key is used to encrypt a session key from one user to the other. The use of a signed Diffie–Hellman key exchange has a number of advantages over an RSA-based key transport:

- In key transport only one party generates the session key, while in key agreement both can parties contribute randomness to the resulting session key.
- Signed ECDH has the property of forward secrecy, whereas an RSA-based key transport does not. An authenticated key agreement/transport protocol is called forward secure if the compromise of the long-term static key does not result in past session keys being compromized. RSA key transport is not forward secure since once you have the long-term RSA decryption key of the recipient you can determine the past session keys; however, in signed ECDH the long-term private keys are only used to produce signatures.

However, note that the one-pass variant of ECDH discussed above, being a key transport mechanism, also suffers from the above two problems of RSA key transport.

The problem with signed ECDH is that it is wasteful of bandwidth. To determine the session key we need to append a signature to the message flows. An alternative system is to return to the message flows in the original ECDH protocol but change the way that the session key is derived. If the session key is derived using static public keys, as well as the transmitted ephemeral keys, we can obtain *implicit* authentication of the resulting session key. This is the approach taken in the MQV protocol of Law, Menezes, Qu, Solinas and Vanstone [**216**].

In the MQV protocol both parties are assumed to have long-term static public/private key pairs. For example, we shall assume that Alice has the static key pair $([a]G, a)$ and Bob has the static key pair $([c]G, c)$. To agree on a shared secret, Alice and Bob generate two ephemeral key pairs; for example, Alice generates the ephemeral key pair $([b]G, b)$ and Bob generates the ephemeral key pair $([d]G, d)$. They exchange the public parts of these ephemeral keys as in the standard ECDH protocol:

$$\begin{array}{ccc} \texttt{Alice} & & \texttt{Bob} \\ b & \xrightarrow{[b]G} & [b]G \\ [d]G & \xleftarrow{[d]G} & d. \end{array}$$

Hence, the message flows are precisely the same as in the ECDH protocol. After the exchange of messages Alice knows

$$a, b, [a]G, [b]G, [c]G \text{ and } [d]G,$$

and Bob knows

$$c, d, [c]G, [d]G, [a]G \text{ and } [b]G.$$

The shared secret is then determined by Alice via the following algorithm:

ALGORITHM I.5: **ECMQV Key Derivation**

```
INPUT:  A set of domain parameters $(K, E, q, h, G)$
        and $a, b, [a]G, [b]G, [c]G$ and $[d]G$.
OUTPUT: A shared secret $G$,
        shared with the entity with public key $[c]G$.
1.  $n \leftarrow \lceil log_2(\#K) \rceil / 2$.
2.  $u \leftarrow (x([b]G) \pmod{2^n}) + 2^n$.
3.  $s \leftarrow b + ua \pmod{q}$.
4.  $v \leftarrow (x([d]G) \pmod{2^n}) + 2^n$.
5.  $Q \leftarrow [s]([d]G + [v]([c]G))$.
6.  If $Q$ is at infinity goto Step 1.
7.  Output $Q$.
```

Bob can also compute the same value of $Q$ by swapping the occurance of $(a, b, c, d)$ in the above algorithm with $(c, d, a, b)$. If we let $u_A$, $v_A$ and $s_A$ denote the values of $u$, $v$ and $s$ computed by Alice and $u_B$, $v_B$ and $s_B$ denote the corresponding values computed by Bob, then we see

$$\begin{aligned} u_A &= v_B, \\ v_A &= u_B. \end{aligned}$$

We then see that

$$\begin{aligned} Q &= [s_A]\left([d]G + [v_A]([c]G)\right) \\ &= [s_A][d + v_A c]G \\ &= [s_A][d + u_B c]G \\ &= [s_A][s_B]G. \end{aligned}$$

In addition, a cofactor variant can be used by setting $Q \leftarrow [h]Q$ before the test for whether $Q$ is the point at infinity in Step 6.

In summary, the ECMQV protocol allows authentic key agreement to occur over an insecure channel, whilst only requiring the same bandwidth as an unauthenticated Diffie–Hellman.

One can also have a one-pass variant of the ECMQV protocol, which enables one party to be offline when the key is agreed. Suppose Bob is the party who is offline; he will still have a long-term static public/private key pair given by $[c]G$. Alice then uses this public key both as the long-term key and the emphemeral key in the above protocol. Hence, Alice determines the shared secret via

$$Q = [s_A]\left(([c]G) + [v_A]([c]G)\right) = [s_A][v_A + 1]([c]G),$$