Lecture 14: Elliptic Curve Cryptography and Digital Rights Management

Lecture Notes on "Computer and Network Security"

by Avi Kak (kak@purdue.edu)

February 28, 2016
11:40pm

Goals:

- Introduction to elliptic curves
- A group structure imposed on the points on an elliptic curve
- Geometric and algebraic interpretations of the group operator
- Elliptic curves on prime finite fields
- **Perl and Python implementations for elliptic curves on prime finite fields**
- Elliptic curves on Galois fields
- Elliptic curve cryptography (EC Diffie-Hellman, EC Digital Signature Algorithm)
- Security of Elliptic Curve Cryptography
- ECC for Digital Rights Management (DRM)

CONTENTS

14.1: WHY ELLIPTIC CURVE CRYPTOGRAPHY?

- As you saw in Section 12.8 of Lecture 12, the computational overhead of the RSA-based approach to public-key cryptography increases with the size of the keys. As algorithms for integer factorization have become more and more efficient, the RSA based methods have had to resort to longer and longer keys.

- Elliptic curve cryptography (ECC) can provide the same level and type of security as RSA (or Diffie-Hellman as used in the manner described in Section 13.5 of Lecture 13) **but with much shorter keys**.

- Table 1 compares the key sizes for three different approaches to encryption for comparable levels of security against brute-force attacks. What makes this table all the more significant is that for comparable key lengths the *computational burdens* of RSA and ECC are comparable. *What that implies is that, with ECC, it takes one-sixth the computational effort to provide the same level of cryptographic security that you get with 1024-bit RSA.* [The table shown here is basically the same table as presented earlier in Section 12.9 of Lecture 12, except that now we also include ECC in our comparison.] [In case the reader is wondering why we placed the word *key* between quotation marks in the header of the "RSA and Diffie-Hellman" column in Table 1, read the beginning of Section 12.8 of Lecture 12. The reason for quoting the same word in the header for the ECC column will turn out to be similar, as you will see from this lecture.]

Symmetric Encryption Key Size in bits	*RSA and Diffie-Hellman "Key" size in bits*	*ECC "Key" Size in bits*
80	1024	160
112	2048	224
128	3072	256
192	7680	384
256	15360	512

Table 1: *A comparison of key sizes needed to achieve equivalent level of security with three different methods.*

- The computational overhead of both RSA and ECC grows as $O(N^3)$ where N is the key length in bits. [Source: Hank van Tilborg, NAW, 2001] Nonetheless, despite this parity in the dependence of the computational effort on key size, it takes far less computational overhead to use ECC on account of the fact that you can get away with much shorter keys.

- Because of the much smaller key sizes involved, ECC algorithms can be implemented on **smartcards** without mathematical coprocessors. **Contactless smart cards** work only with ECC because other systems require too much induction energy. Since shorter key lengths translate into faster handshaking protocols, ECC is also becoming increasingly important for **wireless communications**. [Source: Hank van Tilborg, NAW, 2001]

- For the same reasons as listed above, we can also expect ECC to become important for **wireless sensor networks**.

- If you want to combine forward secrecy, in the sense defined in Section 12.6 of Lecture 12, with authentication, a commonly used algorithm today is ECDHE-RSA. [The acronym "ECDHE" stands for "Elliptic Curve Diffie-Hellman Ephemeral". You will also see in common use a variant acronym: ECDH-RSA. The difference between ECDHE and ECDH is that the "ephemeral" implied by the last letter in the former implies just a one-time use of the session key.] In ECDHE-RSA, RSA is used for certificate based authentication using the TLS/SSL protocol and ECDHE used for creating a one-time session key using the method described in Section 14.12. [You could also use DHE-RSA, which uses the regular Diffie-Hellman Exchange protocol of Section 13.5 of Lecture 13 for creating session keys, for the same purpose. However, you are likely to get greater security with ECDHE-RSA.] [The main reason RSA is widely used for authentication is because a majority of the certificates in use today are based on RSA public keys. However, that is changing. You now see more and more organizations using ECC based certificates. ECC based certificates use the ECDSA algorithm for authentication. This algorithm is presented briefly in Section 14.13. When authentication is carried out with ECDSA and the session key generated with ECDH or ECDHE, the combined algorithm is denoted ECDHE-ECDSA or ECDH-ECDSA. As you will see in Section 14.13, ECDSA stands for "Elliptic Curve Digital Signature

Algorithm."]

- ECC is also used in the algorithms for Digital Rights Management (DRM), as we will discuss in Section 14.14.

- As you will see in Section 20.5 of Lecture 20, ECC is also used in the more recent versions of the Tor protocol.

- Although the algorithmic details of how ECC is used in DRM will be described later in Section 14.14, we will review in the rest of this section how ECC, along with AES, is used in game consoles to keep others from gaining direct access to the binaries and for ensuring that the hardware only executes authenticated code. In particular, we will focus on the PlayStation3 game console.

- PlayStation3 (PS3) stores the executables as **SELF** files. SELF stands for "Signed Executable and Linkable Format." [Think of these as encrypted and signed version of the ".exe" files in a Windows platform.] These files are stored encrypted in different sections in such a way that each section yields the encryption parameters, such as the key and the IV (initialization vector), needed for decrypting the next section. [According to the information at the web links at the end of this section, the first section of the file, 64 bytes long, contains the key and the IV (Initializing Vector) for decoding the metadata section that follows. The first section is encrypted with 256-bit AES in the CBC mode (See Section 9.5.2 of Lecture 9 for this mode). And the metadata section is encrypted with the 128-bit AES in the CTR mode that was described in Section 9.5.5 of Lecture 9. The metadata section of each file contains the key and the IV for decrypting the data section of a file. The data section is also encrypted with 128-bit AES in the CTR mode. As you would expect, the loader program that pulls these files into RAM must decrypt them on the fly, using the parameters extracted from each section to decrypt the next section.]

- In PS3, the SELF files are signed with ECDSA algorithm so that the hardware only executes authenticated code. ECDSA stands for Elliptic Curve Digital Signature Algorithm. We will talk about how exactly ECC can be used for digital signatures in Section 14.13. [Enforcing the condition that only the authenticated code be executed by the hardware is supposed to make it more difficul to run pirated games on a game console. However, this also makes it more difficult for folks to create their own games for PS3. Such folks tend to be mostly Linux users and they would obviously want to be able to replace the game OS with some variant of Linux on their game consoles.]

- See Section 14.13 on how the code authentication part of the security in PS3 was cracked.

- The information presented above concerning PlayStation3 can be found in much greater detail at the links shown below:

  `http://www.youtube.com/watch?v=5E0DkoQjCmI`

  `http://www.ps3devwiki.com/wiki/SELF_File_Format_and_Decryption`

The YouTube video is a recording of a panel session at the Console Hacking 2010 forum of the 27^{th} Chaos Communication Congress. You can see additional such video clips at YouTube if you search for strings like "Console Hacking 2010". The slides that were presented at CCC can be downloaded from

`http://events.ccc.de/congress/2010/Fahrplan/attachments/1780_27c3_console_hacking_2010.pdf`

These slides contain a lot of useful comparative information regarding the different game consoles.

14.2: THE MAIN IDEA OF ECC — IN A NUTSHELL

- Imagine we have a set of points (x_i, y_i) in a plane. The set is very, very large but finite. We will denote this set by E.

- Next imagine we can define a group operator on this set. As you know from Lecture 4, a group operator is typically denoted by the symbol '+' even when the operation itself has nothing whatsoever to do with ordinary arithmetic addition. So given two points P and Q in the set E, the group operator will allow us to calculate a third point R, also in the set E, such that $P + Q = R$.

- Given a point $P \in E$, we will particularly be interested in using the group operator to find $P+P$, $P+P+P$, $P+P+P+\ldots+P$ for an arbitrary number of repeated invocations of the group operator. Given *an ordinary integer* k, we will use the notation $k \times P$ to represent the repeated addition $P+P+\ldots+P$ in which P makes k appearances, with the operator '+' being invoked $k-1$ times. [Note that $k \times P$ is NOT an attempt to define a multiplication operator on the set E. That is because k is an ordinary integer. In other words, k is not in the set E. The only meaning to be associated with $k \times P$ is that of repeated addition.]

- Now imagine that the set E is magical in the sense that, after we have calculated $k \times P$ for a given point $P \in E$, it is extremely difficult to recover k from $k \times P$. We will assume that the only way to recover k from $k \times P$ is to try every possible repeated summation like $P + P$, $P + P + P$, $P + P + P + \ldots + P$ until the result equals what we have for $k \times P$. [Trying to figure out how many times P participates in the repeated sum $P + P + P + \ldots + P$ in order for the result to equal $k \times P$ is referred to as solving the *discrete logarithm problem*. To see why that is so, consider the traditional notion of logarithm that allows us to write $a^k = b$ as $k = \log_a b$. Obviously, a^k is nothing but $a \times a \times \ldots \times a$ with a making k appearances in the repeated invocations of the binary operator '×'. So when we write $a^k = b$ as $k = \log_a b$, we calculate the number of times a participates in the repeated invocations of the binary operator involved. That is the same as what we want to do in order to determine the value of k from $k \times P$: we want to find out how many times P participates in the repeated invocations of the '+' operator. Just don't be fooled by the appearance of the operator '×' in $k \times P$. It is really not a multiplication. It is a shortcut for denoting the repeated addition $P + P + \ldots + P$ involving k appearances of P. The notion of discrete logarithms was discussed earlier in Section 11.8 of Lecture 11 and in Section 13.7 of Lecture 13.]

- If we could ensure the above condition, then "products" like $k \times P$ for $P \in E$ could be used by two parties in a Diffie-Hellman like protocol for sharing a secret session key. Section 14.11 will show you how that can be done. [To convey to you the core idea of what you'll see in Section 14.11, let's say that the point P is made public for all to use. Now party A will select an integer $X_A = k_1$ as his/her private key. The public key for A will be $Y_A = X_A \times P$, that is, a k_1-fold application of the group operator to the point P, implying that while the private key is an ordinary integer, the public key is a point like P. Party B does exactly the same thing: it selects an integer $X_B = k_2$ as his/her private key, with the public key for B being $Y_B = X_B \times P$. The two parties exchange their public keys. Subsequently, A computes the session key by $K_A = X_A \times Y_B = k_1 \times k_2 \times P$ and B computes the session key $K_B = X_B \times Y_A = k_2 \times k_1 \times P$.

Obviously, $K_A = K_B$.]

- **All of the assumptions we have made above are satisfied when the set E of points (x_i, y_i) is drawn from an elliptic curve.**

14.3: WHAT ARE ELLIPTIC CURVES?

- First and foremost, elliptic curves have nothing to do with ellipses. Ellipses are formed by quadratic curves. Elliptic curves are always cubic. [Note: *Elliptic curves* are called *elliptic* because of their relationship to *elliptic integrals* in mathematics. An elliptic integral can be used to determine the arc length of an ellipse.]

- The simplest possible "curves" are, of course, straight lines.

- The next simplest possible curves are conics, these being quadratic forms of the following sort

$$ax^2 \quad + \quad bxy \quad + \quad cy^2 \quad + \quad dx \quad + \quad ey \quad + \quad f \quad = \quad 0$$

  If $b^2 - 4ac$ is less than 0, then the curve is either an ellipse, or a circle, or a point, or the curve does not exist; if it is equal to 0, then we have either a parabola, or two parallel lines, or no curve at all; if it is greater than 0, then we either have a hyperbola or two intersecting lines. (Note that, by definition, a conic is the intersection of a plane with two cones that are joined at their tips.)

- The next simplest possible curves are elliptic curves. An elliptic curve in its "standard form" is described by

$$y^2 \quad = \quad x^3 \;+\; ax \;+\; b$$

for some fixed values for the parameters a and b. This equation is also referred to as **Weierstrass Equation** of **characteristic** 0. [The equation shown involves multiplications and additions over certain objects that are represented by x, y, a, and b. The values that these object acquire are meant to be drawn from a set that must at least be a **ring** with a multiplicative identity element. (See Lecture 4 for what a ring is.) The **characteristic** of such a ring is the number of times you must add the multiplicative identity element in order to get the additive identity element. If adding the multiplicative identity element to itself, no matter how many times, **never** gives us the additive identity element, we say the characteristic is 0. For illustration, the set of all *real* numbers is of characteristic 0 because no matter how many times you add 1 to itself, you will never get a 0. When a set is `not` of characteristic 0, there will exist an integer p such that $p \times n = 0$ for all n. The value of p is then the characteristic of the integral domain. For example, in the set of remainders Z_9 (which is a ring with a multiplicative identity element of 1, although it is not an integral domain since $3 \times 3 = 0 \; mod \; 9$) that you saw in Lecture 5, the numbers $9 \times n$ are 0 for every value of the integer n. So we can say that Z_9 is a ring of characteristic 9. When we say that the equation shown above is of characteristic 0, we mean that the set of numbers that satisfy the equation constitutes a ring of characteristic 0.]

- Elliptic curves have a rich algebraic structure that can be put to use for cryptography.

- Figure 1 shows some elliptic curves for a set of parameters $(a,\ b)$. The top four curves all look smooth (they do not have cusps, for

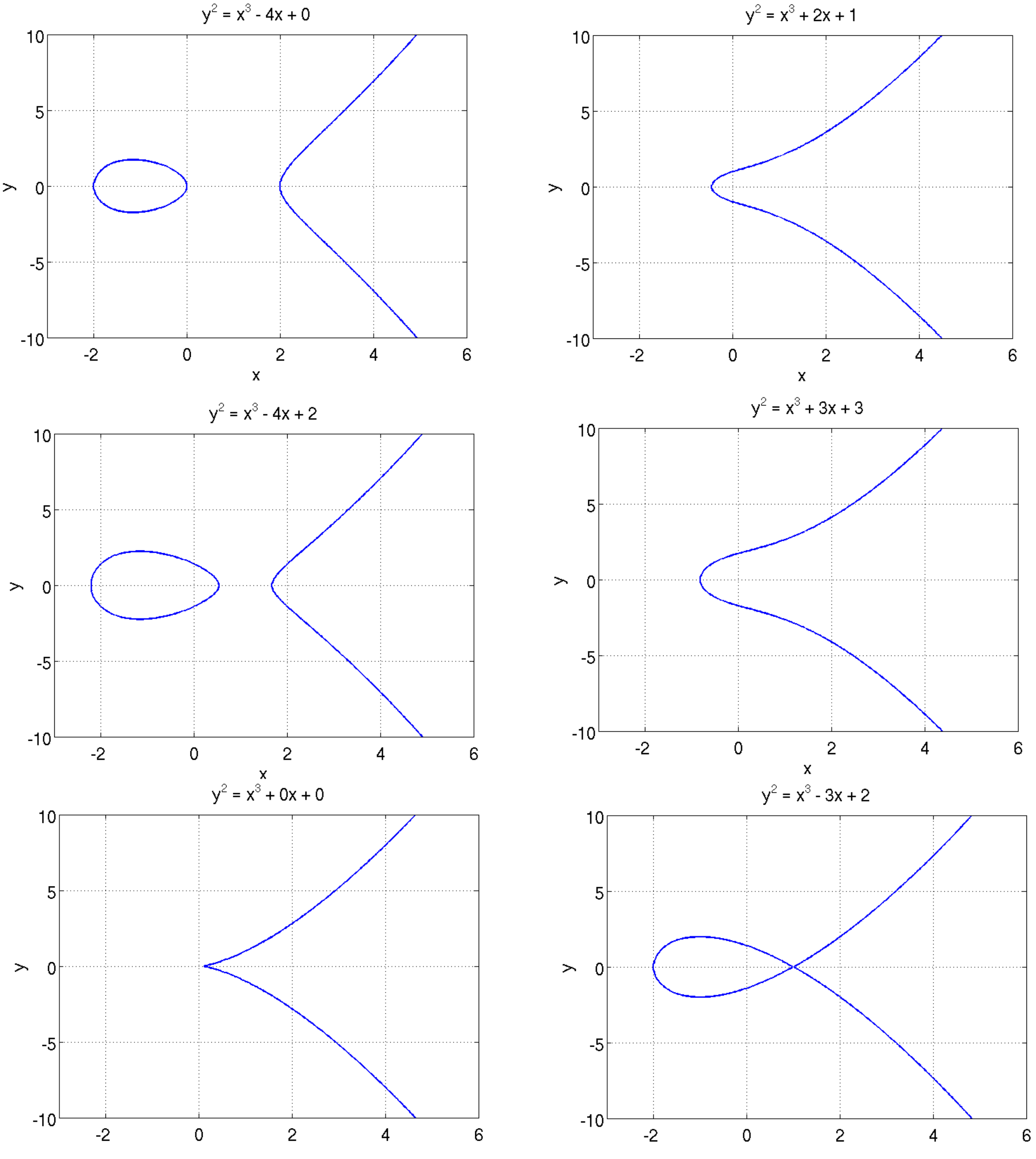


Figure 1: *Elliptic curves for different values of the parameters a and b.* (*This figure is from Lecture 14 of "Lecture Notes on Computer and Network Security" by Avi Kak.*)

example) because they all satisfy the following condition on the **discriminant** of the polynomial $f(x) = x^3 + ax + b$:

$$4a^3 + 27b^2 \neq 0 \tag{1}$$

[Note: The **discriminant of a polynomial** is the product of the squares of the differences of the polynomial roots. The roots of the polynomial $f(x) = x^3 + ax + b$ are obtained by solving the equation $x^3 + ax + b = 0$. Since this is a cubic polynomial, it will in general have three roots. Let's call them r_1, r_2, and r_3. Its discriminant will therefore be

$$D_3 = \prod_{i<j}^{3} (r_i - r_j)^2$$

which is the same as $(r_1 - r_2)^2(r_1 - r_3)^2(r_2 - r_3)^2$. It can be shown that when the polynomial is $x^3 + ax + b$, the discriminant reduces to

$$D_3 = -16(4a^3 + 27b^2)$$

This discriminant must not become zero for an elliptic curve polynomial $x^3 + ax + b$ to possess three distinct roots. If the discriminant is zero, that would imply that two or more roots have coalesced, giving the curve a cusp or some other form of non-smoothness. Non-smooth curves are called **singular**. This notion will be defined more precisely later. It is **not safe** to use singular curves for cryptography. As to why that is the case will become clear later in these lecture notes.]

- The **bottom two** examples in Figure 1 show two elliptic curves for which the condition on the discriminant is violated. For the one on the left that corresponds to $f(x) = x^3$, all three roots of the cubic polynomial have coalesced into a single point and we get a cusp at that point. For the one on the right that corresponds to $f(x) = x^3 - 3x + 2$, two of the roots have coalesced into the point where the curve crosses itself. These two curves are **singular**.

As mentioned earlier, it is **not safe** to use singular curves for cryptography.

- Note that since we can write

$$y \quad = \quad \pm\sqrt{x^3 \ + \ ax \ + \ b}$$

elliptic curves in their standard form will be symmetric about the x-axis.

- It is difficult to comprehend the structure of the curves that involve polynomials of degree greater than 3.

- To give the reader a taste of the parameters used in elliptic curves meant for real security, here is an example:

$$\begin{aligned} y^2 \quad = \quad x^3 \quad & + \quad 317689081251325503476317476413827693272746955927x \\ & + \quad 79052896607878758718120572025718535432100651934 \end{aligned}$$

This elliptic curve is used in the Microsoft Windows Media **Digital Rights Management** Version 2. We will have more to say about this curve in Section 14.14.

14.4: A GROUP OPERATOR DEFINED FOR POINTS ON AN ELLIPTIC CURVE

- The points on an elliptic curve can be shown to constitute a group.

- Recall from Lecture 4 that a group needs the following: (1) a group operator; (2) an identity element with respect to the operator; (3) closure and associativity with respect to the operator; and (4) the existence of inverses with respect to the operator.

- The group operator for the points on an elliptic curve is, by convention, called **addition**. Its definition has nothing to do with the conventional arithmetic addition.

- To add a point P on an elliptic curve to another point Q on the same curve, we use the following rule

  - We first join P with Q with a straight line. The third point of the intersection of this straight line with the curve, if such an intersection exists, is denoted R. The mirror image of this point with respect to the x-coordinate is the point $P + Q$. If the third point of intersection does **not** exist, we say it is at **infinity**.

– The upper two curves in Figure 2 illustrate the addition operation for two different elliptic curves. The values for a and b for the upper curve at the left are -4 and 0, respectively. The values for the same two constants for the upper curve on the right are 2 and 1, respectively.

- But what happens when the intersection of the line joining P and Q with the curve is at infinity?

- We denote the point at infinity by the special symbol **O** and, *through the stipulations that follow*, we then show that this can serve as the additive identity element for the group operator. [If you really think about it, the point represented by **O** is actually at infinity — along the y-axis. You see, the only time when the line joining P and Q does NOT intersect the curve is when that line is parallel to the y-axis. Stare at the right hand portion of the curves in Figure 2, the portion that is open toward the positive direction of the y-axis. As you follow this curve starting from the point on the x-axis, you see the concavity in the curve as it rises to eventually become parallel to the y-axis. This concavity implies that if you were to draw a line through any two points in the upper half of the curve, it is guaranteed to intersect the curve in its lower half portion. Additionally, if you draw a line between any point in the upper half of the curve and a point in lower half, it will intersect the curve either in the upper half or in the lower half.]

- We stipulate that $P + \mathbf{O} = P$ for any point on the curve. [To continue with the small-font note in the previous bullet, joining P with $\mathbf{O}$ according to our group law requires that we draw a line through P that is parallel to the y-axis, and that we then find the "other" point where this line intersects the curve. It follows from the next bullet that this "other" point will be the mirror reflection of P about the x-axis. That is, this "other" point will be at $-P$. When we reflect it with respect to the x-axis, we get back P.]

- We define the additive inverse of a point P as its mirror reflection with respect to the x coordinate. So if Q on the curve is the mirror reflection of P on the curve, then $Q = -P$. For any such two points, it would obviously be the case that the third point of intersection with the curve of a line passing through the first two points will be at infinity. That is, the point of intersection of a point and its additive inverse will be the distinguished point $\mathbf{O}$.

- We will further stipulate that that $\mathbf{O} + \mathbf{O} = \mathbf{O}$, implying that $-\mathbf{O} = \mathbf{O}$. [This is in keeping with the fundamental concept in mathematics that you get to the same point at infinity regardless of whether you head out in the positive direction or the negative direction along a coordinate axis.] Therefore, the mirror reflection of the point at infinity is the same point at infinity.

- Now we can go back to the issue of what happens to $P+Q$ when the intersection of the line passing through the two points P and Q with the elliptic curve is at infinity, as would be the case when P and Q are each other's mirror reflections with regard to the x-axis. Obviously, in this case, the intersection of P and Q is at

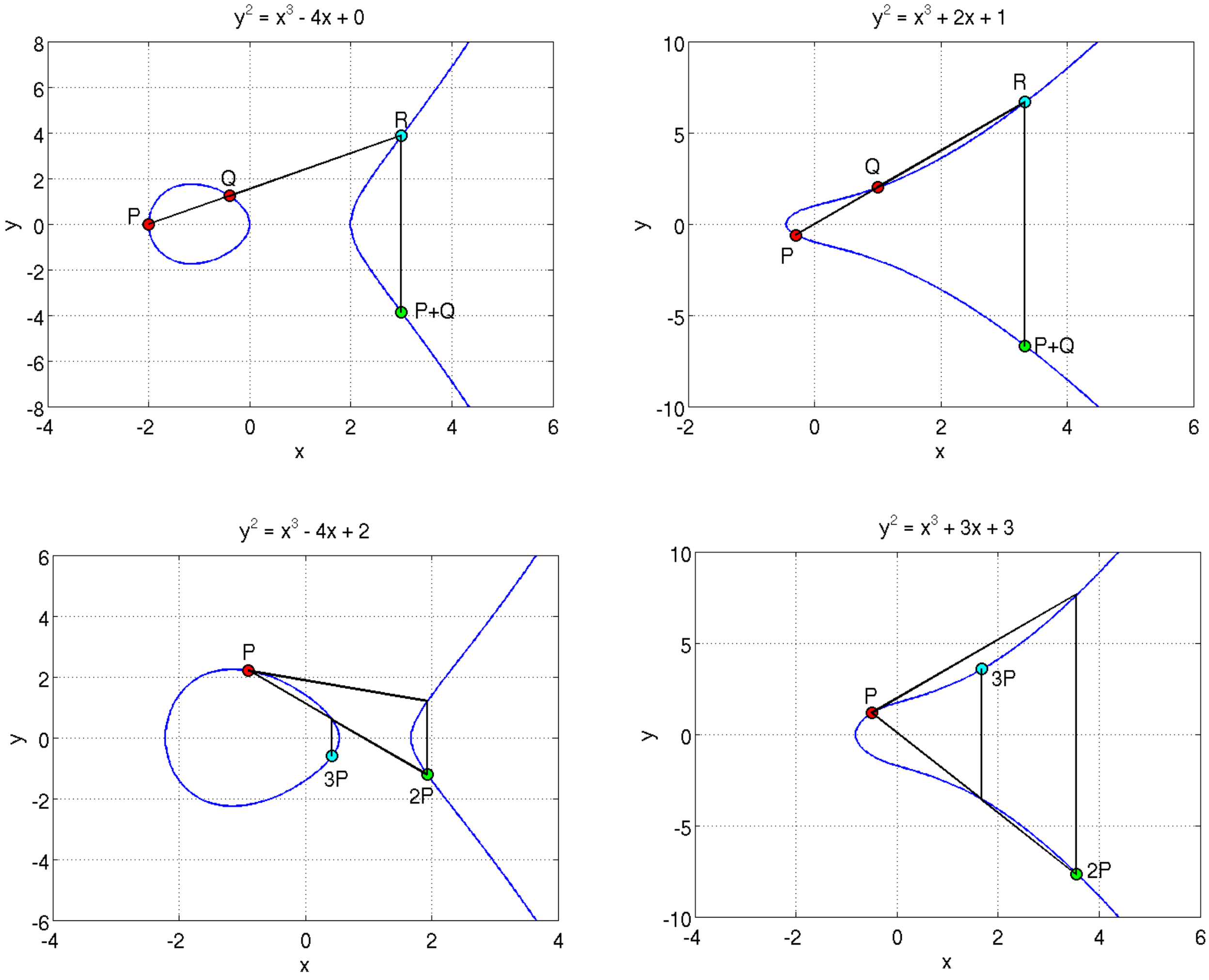


Figure 2: *A pictorial depiction of the group law for elliptic curves.* (*This figure is from Lecture 14 of "Lecture Notes on Computer and Network Security" by Avi Kak.*)

the distinguished point $\mathbf{O}$, whose mirror reflection is also at $\mathbf{O}$. Therefore, for such points, $P \ + \ Q \ = \ \mathbf{O}$ and $Q = -P$.

- We have already defined the additive inverse of a point P as its mirror reflection about the x-axis. What is the additive inverse of a point where the tangent is parallel to the y-axis? The additive inverse of such a point is the point itself. That is, if the tangent at P is parallel to the y-axis, then $P + P = \mathbf{O}$.

- In general, what does it mean to add P to itself? To see what it means, let's consider two distinct points P and Q and let Q approach P. The line joining P and Q will obviously become a tangent at P in the limit. Therefore, the operation $P + P$ means that we must draw a tangent at P, find the intersection of the tangent with the curve, and then take the mirror reflection of the intersection.

- For an elliptic curve

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

we define the set of all points on the curve **along with the distinguished point O** by $E(a, b)$.

- $E(a, b)$ is a group with the "addition" operator as we defined it previously in this section.

- $E(a, b)$ is closed with respect to the addition operation. We can also show geometrically that the property of associativity is satisfied. Every element in the set has its additive inverse in the set.

- Since the operation of "addition" is commutative, $E(a, b)$ is an **abelian group**. (Lecture 4 defines abelian groups.)

- Just for notational convenience, we now define "multiplication" on this group as repeated addition. Therefore,

$$k \times P \quad = \quad P \; + \; P \; + \; \ldots \; + \; P$$

  with P making k appearances on the right. [Note that we are NOT defining a multiplication operator over the set $E(a, b)$. This is merely a notational convenience to define a k-fold addition of an element of $E(a, b)$ to itself.]

- Therefore, we can express $P + P$ as $2P$, $P + P + P$ as $3P$, and so on.

- The two curves at the bottom in Figure 2 show us calculating $2P$ and $3P$ for a given P. The values of a and b for the lower curve on the left are -4 and 2, respectively. The values for the same two constants for the lower curve on the right are both 3.

14.5: THE CHARACTERISTIC OF THE UNDERLYING FIELD AND THE SINGULAR ELLIPTIC CURVES

- The examples of the elliptic curves shown so far were for **the field of real numbers**. (See Lecture 4 for what is meant by a field.) What that means is that the coefficients a and b and the values taken on by the variables x and y all belong to the field of real numbers. These fields are of **characteristic** zero because no matter how many times you add the multiplicative identity element to itself, you'll never get the additive identity element. (See the explanatory note at the fourth bullet in Section 14.3 for what is meant by the characteristic of a field.)

- The group law of Section 14.4 can also be defined when the underlying field is of characteristic 2 or 3. [As already mentioned in the explanatory note at the fourth bullet in Section 14.3, when we consider *real* numbers modulo 2, we have an underlying field of characteristic 2. By the same token, when we consider *real* numbers modulo 3, we have an underlying field of characteristic 3.] But now the elliptic curve $y^2 = x^3 + ax + b$ becomes **singular**, a notion that we will define more precisely shortly. While singular elliptic curves do admit group laws of the sort we showed in Section 14.4, such groups, although defined over the points on the elliptic curve, become **isomorphic** to either the multiplicative or the additive group over the underlying field itself, depending on the type of singularity. **That fact makes singular elliptic curves unsuitable for cryptography because they are easy to crack.**

- To show that the elliptic curve $y^2 = x^3 + ax + b$ becomes **singular** when the characteristic of the underlying field is 2, let's look at the partial derivatives of the two sides of the equation of this curve:

$$2ydy \quad = \quad 3x^2dx \ + \ adx$$

implying

$$\frac{dy}{dx} \quad = \quad \frac{3x^2 \ + \ a}{2y} \tag{2}$$

- A point on the curve is **singular** if $\frac{dy}{dx}$ is not properly defined there and a curve that contains a singular point is a **singular curve**. [If $\frac{dy}{dx}$ is not properly defined at a point, then we cannot construct a tangent at that point. Such a point would not lend itself to the group law presented in Section 14.4, since that law requires us to draw tangents.] This would be the point where both the numerator and the denominator are zero. [When only the denominator goes to zero, the slope is still defined even though it is ∞.] So the elliptic curve $y^2 = x^3 + ax + b$ will become singular if it contains a point (x, y) so that

$$3x^2 \ + \ a \qquad = \qquad 0$$

$$2y \quad = \quad 0$$

and the point (x, y) satisfying these two equations lies on the curve.

- Let's now consider the case when the underlying field is of characteristic 2. In this case, we go back to Equation (2) above and see that, since 2 is the same thing as 0 for such a field [this is based on the definition of **characteristic** in the explanatory note fourth bullet of Section 14.3], the derivative $\frac{dy}{dx}$ will **not** be defined at $x = \sqrt{\frac{-a}{3}}$ since that would make both the numerator and the denominator equal to zero. Therefore, the curve $y^2 = x^3 + ax + b$ will be singular for some values of a that can be obtained by substituting $x = \sqrt{\frac{-a}{3}}$ in the equation of the curve.

- Let's now consider the case of a field of characteristic 3. In this case, since 3 is the same thing as 0, we can write for the curve slope from Equation (2):

$$\frac{dy}{dx} \quad = \quad \frac{a}{2y}$$

This curve becomes singular if we should choose $a = 0$ since the denominator in the ratio shown above will also go to zero at the point where the curve intersects the x-axis.

- In general, when using the elliptic curve equation $y^2 = x^3 + ax + b$, we avoid underlying fields of characteristic 2 or 3 because of the nature of the constraints they place on the parameters a and b in order for the curve to not become singular.

14.6: AN ALGEBRAIC EXPRESSION FOR ADDING TWO POINTS ON AN ELLIPTIC CURVE

- Given two points P and Q on an elliptic curve $E(a,b)$, we have already pointed out that to compute the point $P + Q$, we first draw a straight line through P and Q. We next find the third intersection of this line with the elliptic curve. We denote this point of intersection by R. Then $P+Q$ is equal to the mirror reflection of R about the x-axis.

- In other words, if P, Q, and R are the three intersections of the straight line with the curve, then

$$P \quad + \quad Q \quad = \quad -R$$

- This implies that the three intersections of a straight line with the elliptic curve must satisfy

$$P \quad + \quad Q \quad + \quad R \quad = \quad \mathbf{O}$$

- We will next examine the algebraic implications of the above relationship between the three points of intersection.

- The equation of the straight line that runs through the points P and Q must be of the form:

$$y \quad = \quad \alpha x \ + \ \beta$$

where α is the slope of the line, which is given by

$$\alpha \quad = \quad \frac{y_Q \ - \ y_P}{x_Q \ - \ x_P}$$

- For a point $(x, \ y)$ to lie at the intersection of the straight line and the elliptic curve $E(a, b)$, the following equality must hold

$$(\alpha x \ + \ \beta)^2 \quad = \quad x^3 \ + \ ax \ + \ b \qquad (3)$$

since $y \ = \ \alpha x + \beta$ on the straight line through the points P and Q and since the equation of the elliptic curve is $y^2 \ = \ x^3 + ax + b$.

- For there to be three points of intersection between the straight line and the elliptic curve, the cubic form in Equation (3) must have three roots. **We already know two of these roots, since they must be x_P and x_Q, correspond to the points P and Q.**

- Being a cubic equation, since Equation (3) has at most three roots, the remaining root must be x_R, the x-coordinate of the third point R.

- Equation (3) represents a **monic polynomial**. What that means is that the coefficient of the highest power of x is 1.

- **A property of monic polynomials is that the sum of their roots is equal to the negative of the coefficient of the second highest power.** Expressing Equation (3) in the following form:

$$x^3 \;-\; \alpha^2 x^2 \;+\; (a \;-\; 2\alpha\beta)x \;+\; (b \;-\; \beta^2) \;=\; 0 \quad (4)$$

we notice that the coefficient of x^2 is $-\alpha^2$. Therefore, we have

$$x_P \;+\; x_Q \;+\; x_R \;=\; \alpha^2$$

We therefore have the following result for the x-coordinate of R:

$$x_R \;=\; \alpha^2 \;-\; x_P \;-\; x_Q \qquad (5)$$

- Since the point (x_R, y_R) must be on the straight line $y \;=\; \alpha x + \beta$, we can write for y_R:

$$y_R \;=\; \alpha x_R \;+\; \beta$$

$$\begin{aligned} &= \alpha x_R + (y_P - \alpha x_P) \\ &= \alpha(x_R - x_P) + y_P \qquad (6) \end{aligned}$$

- To summarize, ordinarily a straight line will intersect an elliptic curve at three points. If the coordinates of the first two points are (x_P, y_P) and (x_Q, y_Q), then the coordinates of the third point are

$$\begin{aligned} x_R &= \alpha^2 - x_P - x_Q \qquad (7) \\ y_R &= \alpha(x_R - x_P) + y_P \qquad (8) \end{aligned}$$

- We started out with the following relationship between P, Q, and R

$$P + Q = -R$$

we can therefore write the following expressions for the x and the y coordinates of the addition of two points P and Q:

$$\begin{aligned} x_{P+Q} &= \alpha^2 - x_P - x_Q \qquad (9) \\ y_{P+Q} &= \alpha(x_P - x_R) - y_P \qquad (10) \end{aligned}$$

since the y-coordinate of the reflection $-R$ is negative of the y-coordinate of the point R on the intersecting straight line.

14.7: AN ALGEBRAIC EXPRESSION FOR CALCULATING $2P$ FROM P

- Given a point P on the elliptic curve $E(a, b)$, computing $2P$ (which is the same thing as computing $P + P$), requires us to draw a tangent at P and to find the intersection of this tangent with the curve. The reflection of this intersection about the x-axis is then the value of $2P$.

- Given the equation of the elliptic curve $y^2 = x^3 + ax + b$, the slope of the tangent at a point (x, y) is obtained by differentiating both sides of the curve equation

$$2y\frac{dy}{dx} = 3x^2 + a$$

- We can therefore write the following expression for the slope of the tangent at point P:

$$\alpha = \frac{3x_P^2 + a}{2y_P} \tag{11}$$

- Since drawing the tangent at P is the limiting case of drawing a line through P and Q as Q approaches P, two of the three roots of the following equation (which is the same as Equation (3) you saw before):

$$(\alpha x \; + \; \beta)^2 \quad = \quad x^3 \; + \; ax \; + \; b \tag{12}$$

  must coalesce into the point x_P and the third root must be x_R. As before, R is the point of intersection of the tangent with the elliptic curve.

- As before, we can use the property that sum of the roots of the monic polynomial above must equal the negative of the coefficient of the second highest power. Noting two of the three roots have coalesced into x_P, we get

$$x_P \; + \; x_P \; + \; x_R \quad = \quad \alpha^2$$

- This gives us the following expression for the x coordinate of the point R:

$$x_R \quad = \quad \alpha^2 \; - \; 2x_P \tag{13}$$

- Since the point R must also lie on the straight line $y \; = \; \alpha x + \beta$, substituting the expression for x_R in this equation yields

$$
\begin{aligned}
y_R &= \alpha x_R + \beta \\
&= \alpha x_R + (y_P - \alpha x_P) \\
&= \alpha(x_R - x_P) + y_P \qquad (14)
\end{aligned}
$$

- To summarize, if we draw a tangent at point P to an elliptic curve, the tangent will intersect the curve at a point R whose coordinates are given by

$$
\begin{aligned}
x_R &= \alpha^2 - 2x_P \\
y_R &= \alpha(x_R - x_P) + y_P \qquad (15)
\end{aligned}
$$

- Since the value of $2P$ is the reflection of the point R about the x-axis, the value of $2P$ is obtained by taking the negative of the y-coordinate:

$$
\begin{aligned}
x_{2P} &= \alpha^2 - 2x_P \\
y_{2P} &= \alpha(x_P - x_R) - y_P \qquad (16)
\end{aligned}
$$

  Except for the fact that α is now different, these formulas look very much like those shown in Equations (9) and (10) for the case when the two points are the same.

14.8: ELLIPTIC CURVES OVER Z_p FOR PRIME p

- The elliptic curve arithmetic we described so far was over **real numbers**. These curves cannot be used as such for cryptography because calculations with real numbers are prone to round-off error. **Cryptography requires error-free arithmetic.** That is after all the main reason for why we introduced the notion of a finite field in Lectures 4 through 7.

- By restricting the values of the parameters a and b, the value of the independent variable x, and the value of the dependent variable y to some **prime finite field** Z_p, we obtain elliptic curves that are more appropriate for cryptography. Such curves would be descrbed by

$$y^2 \quad \equiv \quad (x^3 \quad + \quad ax \quad + \quad b) \quad (mod\, p) \tag{17}$$

The points on such curves would be subject to the modulo p version of the same smoothness constraint on the discriminant as we had for the case of real numbers [see Equation (1) in Section 14.3]:

$$(4a^3 \quad + \quad 27b^2) \quad \neq \quad 0 \quad\quad (mod\, p)$$

- We will use the notation $E_p(a, b)$ to represent all the points (x, y) that obey the conditions laid down above. $E_p(a, b)$ will also include the distinguished point **O**, the point at infinity.

- So the points in $E_p(a, b)$ are the set of coordinates (x, y), with $x, y \in Z_p$, such that the equation $y^2 = x^3 + ax + b$, with $a, b \in Z_p$ is satisfied modulo p and such that the condition $4a^3 + 27b^2 \neq 0 \ (mod\ p)$ is fulfilled.

- Obviously, then, the set of points in $E_p(a, b)$ is no longer a curve, but a collection of discrete points in the (x, y) plane (or, even more precisely speaking, in the plane corresponding to the Cartesian product $Z_p \times Z_p$).

- Since the points in $E_p(a, b)$ can no longer be connected to form a smooth curve, we cannot use the geometrical construction to illustrate the action of the group operator. That is, given a point P, now one cannot show geometrically how to compute $2P$, or given two points P and Q, one cannot show geometrically how to determine $P + Q$. **However, the algebraic expressions we derived for these operations continue to hold good provided the calculations are carried out modulo p.**

- Note that for a **prime finite field** Z_p, the value of p is its **characteristic**. (See Section 14.3 for what is meant by the characteristic of a ring.) Elliptic curves over **prime finite fields** with $p \leq 3$, while admitting the group law, are **not** suitable for cryptography. (See Section 14.5)

- The set $E_p(a, b)$ of points, with the elliptic curve defined over a prime finite field Z_p, constitutes a group, the group operator being as defined in Sections 14.6 and 14.7. [In the hierarchy of algebraic structures presented in Lecture 4, the set $E_p(a, b)$ is NOT even a ring since we have not defined multiplication over the set. Yes, we can compute things like $k \times G$ for an element $G \in E_p(a, b)$, since we can construe such a product as repeated addition of the element G. Nonetheless, we are NOT allowed to compute a product of arbitrary two elements in $E_p(a, b)$.]

14.8.1: Perl and Python Implementations for Elliptic Curves Defined Over Prime Finite Fields

- Shown next is Python code that implements the algebraic formulas derived previously in Sections 14.6 and 14.7 for the case of elliptic curves defined over a prime finite field Z_p. [Note that this code is NOT optimized for very large primes, that is, for primes of the size you are likely to encounter in production work.]

- The implementation of the `add()` in lines (B1) through (B21) is based on the algebraic formulas for the group law in Sections 14.6 and 14.7. This code takes care of all possibilities concerning the group operator: (i) when both the points are at infinity; (ii) when only one of the points is at infinity; (iii) when the two points are different but on the same vertical line; (iv) when the two points are the same; (v) when the two points are different but on the same vertical line; and, finally, (vi) and when the two points are different and NOT on the same vertical line. The code shown in lines (C1) through (C9) is for what we loosely refer to as multiplying a point on the curve with an integer. A naive implementation of this would be as shown below where we simply add the point to itself repeatedly.

```
def k_times_Point(curve, point, k, mod):
    if isinstance(point, basestring): return "point at infinity"
    elif k == 1: return point
    else:
        result = point
        for i in range(k-1):
            result = add(curve, result, point, mod)
        return result
```

What is shown in the code block in lines (C1) through (C9) is a more efficient version of this. With this implementation, if the number of times you need to add a point to itself is, say, 2^n, you would need to call **add()** only n times. When the number of times you need to add a point to itself is not a power of 2, you specialcase that as shown in line (C6).

```
#!/usr/bin/env python

##  ECC.py
##  Author: Avi Kak
##  February 26, 2012
##  Modified: February 28, 2016

import random, sys, functools
from PrimeGenerator import *          # From Homework Problem 15 of Lecture 12
from Factorize import factorize       # From Section 12.6 of Lecture 12


def MI(num, mod):            # This method is from Section 5.7 of Lecture 5    #(A1)
    '''
    The function returns the multiplicative inverse (MI)
    of num modulo mod
    '''
    NUM = num; MOD = mod                                                       #(A2)
    x, x_old = 0, 1                                                            #(A3)
    y, y_old = 1, 0                                                            #(A4)
    while mod:                                                                 #(A5)
        q = num // mod                                                         #(A6)
        num, mod = mod, num % mod                                              #(A7)
        x, x_old = x_old - q * x, x                                            #(A8)
        y, y_old = y_old - q * y, y                                            #(A9)
    if num != 1:                                                               #(A10)
        return "NO MI. However, the GCD of %d and %d is %u" % (NUM, MOD, num)  #(A11)
    else:                                                                      #(A12)
        MI = (x_old + MOD) % MOD                                               #(A13)
        return MI                                                              #(A14)
```

```
def add(curve, point1, point2, mod):                                           #(B1)
    ’’’
    If ‘point1 + point2 = result_point’, this method returns the
    result_point, where ‘+’ means the group law for the set of points
    E_p(a,b) on the elliptic curve y^2 = x^3 + ax + b defined over the
    prime finite field Z_p for some prime p.
    Parameters:
          curve    =  (a,b)       represents the curve y^2 = x^3 + ax + b
          point1   =  (x1,y1)     the first point on the curve
          point2   =  (x2,y2)     the second point on the curve
          mod      =  a prime p for Z_p elliptic curve
    The args for the parameters point1 and point2 may also be the string
    "point at infinity" when one or both of these points is meant to be the
    identity element of the group E_p(a,b).
    ’’’
    if isinstance(point1, str) and isinstance(point2, str):                    #(B2)
        return "point at infinity"                                             #(B3)
    elif isinstance(point1, str):                                              #(B4)
        return point2                                                          #(B5)
    elif isinstance(point2, str):                                              #(B6)
        return point1                                                          #(B7)
    elif (point1[0] == point2[0]) and (point1[1] == point2[1]):                #(B8)
        alpha_numerator = 3 * point1[0]**2 + curve[0]                          #(B9)
        alpha_denominator =  2 * point1[1]                                    #(B10)
    elif point1[0] == point2[0]:                                              #(B11)
        return "point at infinity"                                            #(B12)
    else:                                                                     #(B13)
        alpha_numerator = point2[1] - point1[1]                               #(B14)
        alpha_denominator = point2[0] - point1[0]                             #(B15)
    alpha_denominator_MI = MI( alpha_denominator, mod )                       #(B16)
    alpha = (alpha_numerator * alpha_denominator_MI) % mod                    #(B17)
    result = [None] * 2                                                       #(B18)
    result[0] = (alpha**2 - point1[0] - point2[0]) % mod                      #(B19)
    result[1] = (alpha * (point1[0] - result[0]) - point1[1]) % mod           #(B20)
    return result                                                             #(B21)

def k_times_point(curve, point, k, mod):                                       #(C1)
    ’’’
    This method returns a k-fold application of the group law to the same
    point.  That is, if ‘point + point + .... + point = result_point’,
    where we have k occurrences of ‘point’ on the left, then this method
    returns result of such ‘summation’.  For notational convenience, we may
    refer to such a sum as ‘k times the point’.
    Parameters:
          curve    =  (a,b)       represents the curve y^2 = x^3 + ax + b
          point    =  (x,y)       a point on the curve
          k        =  positive integer
          mod      =  a prime p for Z_p elliptic curve
    ’’’
    if k <= 0: sys.exit("k_times_point called with illegal value for k")       #(C2)
    if isinstance(point, str): return "point at infinity"                      #(C3)
    elif k == 1: return point                                                  #(C4)
    elif k == 2: return add(curve, point, point, mod)                          #(C5)
    elif k % 2 == 1:                                                           #(C6)
        return add(curve, point, k_times_point(curve, point, k-1, mod), mod)   #(C7)
    else:                                                                      #(C8)
        return k_times_point(curve, add(curve, point, point, mod), k/2, mod)   #(C9)

def on_curve(curve, point, mod):                                               #(C10)
    '''
    Checks if a point is on an elliptic curve.
    Parameters:
          curve    =  (a,b)       represents the curve y^2 = x^3 + ax + b
          point    =  (x,y)       a candidate point
          mod      =  a prime p for Z_p elliptic curve
    '''
    lhs = point[1]**2                                                          #(C11)
    rhs = point[0]**3 + curve[0]*point[0] + curve[1]                           #(C12)
    return lhs % mod == rhs % mod                                              #(C13)

def get_point_on_curve(curve, mod):                                            #(D1)
    '''
    WARNING: This is NOT an appropriate function to run for very large
             values of mod (as in the elliptic curves for production work.
             It would be much, much too slow.
    Returns a point (x,y) on a given elliptic curve.
    Parameters:
          curve    =  (a,b)       represents the curve y^2 = x^3 + ax + b
          mod      =  a prime p for Z_p elliptic curve
    '''
    ran = random.Random()                                                      #(D2)
    x = ran.randint(1, mod-1)                                                  #(D3)
    y = None                                                                   #(D4)
    trial = 0                                                                  #(D5)
    while 1:                                                                   #(D6)
        trial += 1                                                             #(D7)
        if trial >= (2*mod): break                                             #(D8)
        rhs = (x**3 + x*curve[0] + curve[1]) % mod                             #(D9)
        if rhs == 1:                                                           #(D10)
            y = 1                                                              #(D11)
            break                                                              #(D12)
        factors = factorize(rhs)                                               #(D13)
        if (len(factors) == 2) and (factors[0] == factors[1]):                 #(D14)
            y = factors[0]                                                     #(D15)
            break                                                              #(D16)
        x = ran.randint(1, mod-1)                                              #(D17)
    if not y:                                                                  #(D18)
        sys.exit("Point on curve not found.  Try again --- if you have time")  #(D19)
    else:                                                                      #(D20)
        return (x,y)                                                           #(D21)

def choose_curve_params(mod, num_of_bits):                                     #(E1)
    a,b = None,None                                                            #(E2)
    while 1:                                                                   #(E3)
        a = random.getrandbits(num_of_bits)                                    #(E4)
        b = random.getrandbits(num_of_bits)                                    #(E5)
        if (4*a**3 + 27*b**2)%mod == 0: continue                               #(E6)
        break                                                                  #(E7)
    return (a,b)                                                               #(E8)
```

```
def mycmp(p1, p2):                                                             #(F1)
    if p1[0] == p2[0]:                                                         #(F2)
        if p1[1] > p2[1]: return 1                                             #(F3)
        elif p1[1] < p2[1]: return -1                                          #(F4)
        else: return 0                                                         #(F5)
    elif p1[0] > p2[0]: return 1                                               #(F6)
    else: return -1                                                            #(F7)

def display( all_points ):                                                     #(G1)
    point_at_infy = ["point at infinity" for point in all_points \
                                             if isinstance(point,str)]         #(G2)
    all_points = [[int(str(point[0]).rstrip("L")), \
                   int(str(point[1]).rstrip("L"))] \
                     for point in all_points if not isinstance(point, str)]    #(G3)

    all_points.sort( key = functools.cmp_to_key(mycmp) )
    all_points += point_at_infy                                                #(G5)
    print(str(all_points))                                                     #(G6)


if __name__ == '__main__':

    # Example 1:
    p = 23                                                                     #(M1)
    a,b = 1,4          # y^2 = x^3 + x + 4                                     #(M2)
    point = get_point_on_curve( (a,b), p)                                      #(M3)
    print("Point: %s\n" % str(point))       # (7,3)                            #(M4)
    all_points = list(map( lambda k: k_times_point((a,b), \
                              (point[0],point[1]), k, p), range(1,30)))         #(M5)
    display(all_points)                                                        #(M6)
          # [[0, 2], [0, 21], [1, 11], [1, 12], [4, 7], [4, 16], [7, 3],
          #  [7, 20], [8, 8], [8, 15], [9, 11], [9, 12], [10, 5],
          #  [10, 18], [11, 9], [11, 14], [13, 11], [13, 12], [14, 5],
          #  [14, 18], [15, 6], [15, 17], [17, 9], [17, 14], [18, 9],
          #  [18, 14], [22, 5], [22, 18], 'point at infinity']

    # Example 2:
    generator = PrimeGenerator( bits = 16 )                                    #(M7)
    p = generator.findPrime()                           # 64951                #(M8)
    print("Prime returned: %d" % p)                                            #(M9)
    a,b = choose_curve_params(p, 16)                                           #(M10)
    print("a and b for the curve: %d %d" % (a, b))      # 62444, 47754         #(M11)
    point = get_point_on_curve( (a,b), p)                                      #(M12)
    print(str(point))                                   # (1697, 89)           #(M13)

    # Example 3:
    ##  Parameters of the DRM2 elliptic curve:
    p = 785963102379428822376694789446897396207498568951                       #(M14)
    a = 317689081251325503476317476413827693272746955927                       #(M15)
    b = 79052896607878758718120572025718535432100651934                        #(M16)
    # A point on the curve:
    Gx = 771507216262649826170648268565579889907769254176                      #(M17)
    Gy = 390157510246556628525279459266514995562533196655                      #(M18)

    print(str(list(map( lambda k: k_times_point((a,b), (Gx,Gy), k, p),
                                   range(1,5)))))                        #(M19)
#         [(77150721626264982617064826856557988990776925417 6L,
```

- All you have to do to execute the above script is to make the call:

      ECC.py

  A typical call will produce the output that is shown in the commented out sections of the code shown above. As you can see in `main`, the script presents three examples. Example 1, in lines (M1) through (M6), first specifies a small prime in line (M1) and the parameters of the curve in line (M2). It then calls on the function `get_point_on_curve()` to fetch a point on the curve. As shown in commented out part of line (M4), the point returned is at the coordinates $(7, 3)$. Starting at this point, the statements in lines (M4) and (M5) uses the notion of repeated additions to generated 30 points on the elliptic curve. These are displayed by the statement in line (M6) in the commented out section just below that line.

- Subsequently, in Example 2 in lines (M7) through (M13), we first call on the `PrimeGenerator` tool in lines (M7) and (M8) to give us a 16-bit prime number for a new modulus whose value is shown in the commented out portion of line (M8). In line (M10), we then call on the function `choose_curve_params()` to return values for the curve parameters a and b for a non-singular elliptic curve with respect to the modulus shown in line (M8). Using the values of a and b shown in the commented out portion of line (M11), we then call `get_point_on_curve()` in line (M12) to give us a point on the curve, whose coordinates are shown in the commented-out portion of line (M13).

- Finally, in Example 3 in lines (M14) through (M19), for the modulus and the curve parameters a and b, we use values that were actually used in a DRM application. These values are shown in lines (M14), (M15), and (M16). In lines (M17) and (M18), we then specifiy a point on the curve from the same DRM application. Subsequently, we call on the `k_times_point()` function in line (M19) to use the group law to generate a total of five points on the curve starting from the fist point shown in lines (M17) and (M18).

- I'll now present a Perl version of the Python script shown above:

```
#!/usr/bin/env perl

##  ECC.pl
##  Author: Avi Kak
##  February 28, 2016

use strict;
use warnings;
use Math::BigInt;

require "FactorizeWithBigInt.pl";          # From Lecture 12, Section 12.9
require "PrimeGenerator.pl";               # From Lecture 12, Section 12.13
```

```
###############################  class ECC  ######################################
package ECC;

sub new {                                                                    #(A1)
    my ($class, %args) = @_;                                                 #(A2)
    bless {                                                                  #(A3)
        mod  =>  $args{mod},                                                 #(A4)
        a    =>  $args{a},                                                   #(A5)
        b    =>  $args{b},                                                   #(A6)
    }, $class;                                                               #(A7)
}

# class method:
sub choose_curve_params {                                                    #(B1)
    my ($mod, $num_of_bits) = @_;                                            #(B2)
    my ($param1,$param2) = (undef, undef);                                   #(B3)
    while (1) {                                                              #(B4)
        my @arr = map {my $x = rand(1); $x > 0.5 ? 1 : 0 } 0 .. $num_of_bits-1;
                                                                             #(B5)
        my $bstr = join ’’, split /\s/, "@arr";                              #(B6)
        $param1 = oct("0b".$bstr);                                           #(B7)
        $param1 = Math::BigInt->new("$param1");                              #(B8)
        @arr = map {my $x = rand(1); $x > 0.5 ? 1 : 0 } 0 .. $num_of_bits-1; #(B9)
        $bstr = join ’’, split /\s/, "@arr";                                 #(B10)
        $param2 = oct("0b".$bstr);                                           #(B11)
        $param2 = Math::BigInt->new("$param2");                              #(B12)
        last unless $param1->copy()->bpow(Math::BigInt->new("3"))
         ->bmul(Math::BigInt->new("4"))->badd($param2->copy()
          ->bmul($param2)->bmul(Math::BigInt->new("27")))
            ->bmod($mod)->bzero();                                           #(B13)
    }
    return ($param1, $param2);                                               #(B14)
}

sub mycmp3 {                                                                 #(C1)
    my $self = shift;                                                        #(C2)
    my ($p1, $p2) = ($a, $b);                                                #(C3)
    if ($p1->[0]->bcmp($p2->[0]) == 0) {                                     #(C4)
        if ($p1->[1]->bcmp($p2->[1]) > 0) {                                  #(C5)
            return 1;                                                        #(C6)
        } elsif ( $p1->[1]->bcmp($p2->[1]) < 0) {                            #(C7)
            return -1;                                                       #(C8)
        } else {                                                             #(C9)
            return 0;                                                        #(C10)
        }
    } elsif ($p1->[0]->bcmp($p2->[0]) > 0) {                                 #(C11)
        return 1;                                                            #(C12)
    } else {                                                                 #(C13)
        return -1;                                                           #(C14)
    }
}

sub display {                                                                #(D1)
    my $self = shift;                                                        #(D2)
    my @all_points = @{$_[0]};                                                  #(D3)
    my @numeric_points = grep {$_ !~ /point_at_infinity/} @all_points;          #(D4)
    my @sorted  = sort   mycmp3   @numeric_points;                              #(D5)
    push @sorted, "point_at_infinity";                                          #(D6)
    my @output = map { $_ !~ /point_at_infinity/ ?
                   "($_->[0],$_->[1])" : "point_at_infinity" } @sorted;         #(D7)
    print "@output\n";                                                          #(D8)
}

##  This function returns the multiplicative inverse (MI) of $num modulo $mod
sub MI {                                                                        #(E1)
    my $self = shift;                                                           #(E2)
    my ($num, $mod) = @_;                                                       #(E3)
    my ($NUM, $MOD) = ($num, $mod);                                             #(E4)
    my ($x, $x_old) = (Math::BigInt->bzero(), Math::BigInt->bone());            #(E5)
    my ($y, $y_old) = (Math::BigInt->bone(), Math::BigInt->bzero());            #(E6)
    while ($mod->is_pos()) {                                                    #(E7)
        my $q = $num->copy()->bdiv($mod);                                       #(E8)
        ($num, $mod) = ($mod, $num->copy()->bmod($mod));                        #(E9)
        ($x, $x_old) = ($x_old->bsub( $q->bmul($x) ), $x);                      #(E10)
        ($y, $y_old) = ($y_old->bsub( $q->bmul($y)), $y);                       #(E11)
    }
    if ( ! $num->is_one() ) {                                                   #(E12)
        return undef;                                                           #(E13)
    } else {                                                                    #(E14)
        my $MI = $x_old->badd( $MOD )->bmod( $MOD );                            #(E15)
        return $MI;                                                             #(E16)
    }
}

##  The args for the parameters point1 and point2 may also be the string
##  "point at infinity" when one or both of these points is meant to be the
##  identity element of the group E_p(a,b).
sub add {                                                                       #(F1)
    my $self = shift;                                                           #(F2)
    my ($point1, $point2) = @_;                                                 #(F3)
    my ($alpha_numerator, $alpha_denominator);                                  #(F4)
    if (($point1 =~ /point_at_infinity/)
                          && ($point2 =~ /point_at_infinity/)) {                #(F5)
        return "point_at_infinity";                                             #(F6)
    } elsif ($point1 =~ /point_at_infinity/) {                                  #(F7)
        return $point2;                                                         #(F8)
    } elsif ($point2 =~ /point_at_infinity/) {                                  #(F9)
        return $point1;                                                         #(F10)
    } elsif (($point1->[0]->bcmp( $point2->[0] ) == 0)
                   && ($point1->[1]->bcmp( $point2->[1] ) == 0 )) {             #(F11)
        $alpha_numerator = $point1->[0]->copy()->bmul($point1->[0])
                   ->bmul(Math::BigInt->new("3"))->badd($self->{a});            #(F12)
        $alpha_denominator =  $point1->[1]->copy()->badd($point1->[1]);         #(F13)
    } elsif ($point1->[0]->bcmp( $point2->[0] ) == 0 ) {                        #(F14)
        return "point_at_infinity";                                             #(F15)
    } else {
        $alpha_numerator = $point2->[1]->copy()->bsub( $point1->[1] );          #(F16)
        $alpha_denominator = $point2->[0]->copy()->bsub( $point1->[0] );        #(F17)
    }
    my $alpha_denominator_MI =
                  $self->MI( $alpha_denominator->copy(), $self->{mod} );     #(F18)
    my $alpha =
      $alpha_numerator->bmul( $alpha_denominator_MI )->bmod( $self->{mod} ); #(F19)
    my @result = (undef, undef);                                             #(F20)
    $result[0] = $alpha->copy()->bmul($alpha)
        ->bsub( $point1->[0] )->bsub( $point2->[0] )->bmod( $self->{mod} );  #(F21)
    $result[1] = $alpha->copy()
            ->bmul( $point1->[0]->copy()->bsub($result[0]) )
                   ->bsub( $point1->[1] )->bmod( $self->{mod} );             #(F22)
    return \@result;                                                         #(F22)
}

##  Returns a point (x,y) on a given elliptic curve.
sub get_point_on_curve {                                                     #(G1)
    my $self = shift;                                                        #(G2)
    my $randgen = Math::BigInt::Random::OO->new( max => $self->{mod} - 1 );  #(G3)
    my $x = Math::BigInt->new();                                             #(G4)
    unless ($x->is_pos()) {                                                  #(G5)
        $x =  $randgen->generate(1);                                         #(G6)
    }
    my $y;                                                                   #(G7)
    my $trial = Math::BigInt->bzero();                                       #(G8)
    while (1) {                                                              #(G9)
        last if $trial->binc()->bcmp(
                      $self->{mod}->copy()->badd($self->{mod}) ) >= 0;       #(G10)
        my $rhs = $x->copy()->bpow(Math::BigInt->new("3"))
                     ->badd($x->copy()->bmul($self->{a}->copy()))
                           ->badd($self->{b}->copy())->bmod( $self->{mod} ); #(G11)
        if ($rhs->is_one()) {                                                #(G12)
            $y = Math::BigInt->bone();                                       #(G13)
            last;                                                            #(G14)
        }
        my @factors = @{FactorizeWithBigInt->new($rhs)->factorize()};        #(G15)
        if ((@factors == 2) && ($factors[0] == $factors[1])) {               #(G16)
            $y = $factors[0];                                                #(G17)
            last;                                                            #(G18)
        }
        $x = Math::BigInt->new();                                            #(G19)
        unless ($x->is_pos()) {                                              #(G20)
            $x =  $randgen->generate(1);                                     #(G21)
        }
    }
    if (! defined $y) {                                                      #(G22)
        die "Point on curve not found.  Try again --- if you have time";     #(G23)
    } else {                                                                 #(G24)
        my @point = ($x, $y);                                                #(G25)
        return \@point;                                                      #(G26)
    }
}

##  This method returns a k-fold application of the group law to the same
##  point.  That is, if 'point + point + .... + point = result_point',
##  where we have k occurrences of 'point' on the left, then this method
##  returns result of such 'summation'.  For notational convenience, we may
##  refer to such a sum as ‘k times the point’.
##  Parameters:
sub k_times_point {                                                          #(H1)
    my $self = shift;                                                        #(H2)
    my ($point, $k) = @_;                                                    #(H3)
    die "k_times_point called with illegal value for k" unless $k > 0;       #(H4)
    if ($point =~ /point_at_infinity/) {                                     #(H5)
        return "point_at_infinity";                                          #(H6)
    } elsif ($k == 1) {                                                      #(H7)
        return $point;                                                       #(H8)
    } elsif ($k == 2) {                                                      #(H9)
        return $self->add($point, $point);                                   #(H10)
    } elsif ($k %2 == 1) {                                                   #(H11)
        return $self->add($point, $self->k_times_point($point, $k-1));       #(H12)
    } else {                                                                 #(H13)
        return $self->k_times_point($self->add($point, $point), int($k/2));  #(H14)
    }
}

1;

################################  main    ########################################
package main;

#Example 1:
my $p = 23;                                                                  #(M1)
$p = Math::BigInt->new("$p");                                                #(M2)
my ($a, $b) = (1,4);          # y^2 = x^3 + x + 4                            #(M3)
$a = Math::BigInt->new("$a");                                                #(M4)
$b = Math::BigInt->new("$b");                                                #(M5)
my $ecc = ECC->new( mod => $p, a => $a, b => $b );                           #(M6)
my $point = $ecc->get_point_on_curve();                                      #(M7)
print "Point: @{$point}\n";        # Point: (7,3)                            #(M8)
my @all_points = map {my $k = $_; $ecc->k_times_point($point, $k)} 1 .. 31;  #(M9)
$ecc->display(\@all_points);                                                 #(M10)
# (0,2) (0,21) (1,11) (1,12) (4,7) (4,16) (7,3) (7,3) (7,20) (8,8) (8,15) (9,11)
# (9,12) (10,5) (10,18) (11,9) (11,14) (13,11) (13,12) (14,5) (14,18) (15,6) (15,17)
# (17,9) (17,14) (18,9) (18,14) (22,5) (22,18) (22,18) point_at_infinity

# Example 2:
my $generator = PrimeGenerator->new(bits => 16);                             #(M11)
$p = $generator->findPrime();           # 64951                              #(M12)
$p = Math::BigInt->new("$p");                                                #(M13)
print "Prime returned: $p\n";           # Prime returned: 56401              #(M14)
($a,$b) = ECC::choose_curve_params($p, 16);                                  #(M15)
print "Parameters a and b for the curve: $a, $b\n";                          #(M16)
                          # Parameters a and b for the curve: 52469, 51053
$ecc = ECC->new( mod => $p, a => $a, b => $b );                              #(M17)
$point = $ecc->get_point_on_curve();                                         #(M18)
print "Point: @{$point}\n";             # Point: 36700 97                    #(M19)

# Example 3:
##  Parameters of the DRM2 elliptic curve:
$p = Math::BigInt->new("785963102379428822376694789446897396207498568951");  #(M20)
$a = Math::BigInt->new("317689081251325503476317476413827693272746955927");  #(M21)
```

```
$b = Math::BigInt->new("79052896607878758718120572025718535432100651934");   #(M22)
# A point on the curve:
my $Gx =
     Math::BigInt->new("771507216262649826170648268565579889907769254176");  #(M23)
my $Gy =
     Math::BigInt->new("390157510246556628525279459266514995562533196655");  #(M24)
$ecc = ECC->new( mod => $p, a => $a, b => $b );                              #(M25)
@all_points = map {my $k = $_; $ecc->k_times_point([$Gx,$Gy], $k)} 1 .. 5;   #(M26)
$ecc->display(\@all_points);                                                 #(M27)
#  (131207041319172782403866856907760305385848377513,
#                             2139936453045853218229235170381891784525607843)
#  (404132732284922951107528145083106738835171813225,
#                             165281153861339913077400732834828025736032818781)
#  (69522588007620989965528835803979590326842783681O,
#                             87701351544010607198039768840869029919832813267)
#  (716210695201203540500406352786629938966496775642,
#                             251074363473168143346338802961433227920575579388)
#  (771507216262649826170648268565579889907769254176,
#                             390157510246556628525279459266514995562533196655)
```

- All you have to do to invoke the above script is to invoke it by the command line:

  ```
  ECC.pl
  ```

  As the reader can see in the output shown in the commented out portion sof the script, the Perl version behaves the same as the Python code shown earlier.

- The elliptic curve used in Example 1 in both the scripts shown in this section is an example of a *cyclic* curve. As shown in the commented-out section just after line (M6) of the Python script and just after line (M10) of the Perl version, the number of points on such a curve, including the point at infinity, is a prime number — in this case 29. [We say that the *order* of the curve used in Example 1 is 29.] For a cyclic curve, every point, except of course the point at infinity, can serve as the generator of the entire curve. That is, any point on such a curve can be used to generate all the other points, including the point at infinity, through the $k \times G$ calculation for different values of k. If we attempted to generate more than 29 points, the additional points would be repeated versions of the points already calculated. [For more information on cyclic curves, see the paper "The Elliptic Curve Digital Signature Algorithm (ECDSA)" by Don Johnson, Alfred Menezes, and Scott Vanstone.]

- We should also mention that you can also define an elliptic curve when the coordinates are drawn from the multiplicative group $(Z/NZ)^{\times}$ for any positive integer N. Recall from Section 11.8 of Lecture 11 and Section 13.5 of Lecture 13 that when $N = p$, that is, when N is a prime, we denote this multiplicative group by Z_p^*. The group Z_p^*, NEVER to be confused with the finite field Z_p, consists of the $p - 1$ integers $\{1, 2, 3, \cdots, p - 1\}$. In Section 14.14, we will show how an elliptic curve whose points are drawn from Z_p^* is used in Digital Rights Management. The set $E_p(a, b)$ of points, with the elliptic curve defined over the group Z_p^* also constitutes a group for the same reasons as stated above.

- As we will see in the next section, elliptic curves can also be defined over **Galois Fields** $GF(2^m)$ that we introduced in Lecture 7. Galois fields have characteristic 2. Because of that fact, elliptic curves over $GF(2^m)$ require a form that is different from the one you have seen so far.

14.9: ELLIPTIC CURVES OVER GALOIS FIELDS $GF(2^m)$

- For hardware implementations of ECC, it is common to define elliptic curves over a Galois Field $GF(2^n)$.

- What makes the binary finite fields more convenient for hardware implementations is that the elements of $GF(2^n)$ can be represented by n-bit binary code words. (See Lecture 7.)

- You will recall from Lecture 7 that the addition operation in $GF(2^n)$ is like the XOR operation on bit patterns. That is $x + x = 0$ for all $x \in GF(2^n)$. This implies that a finite field of the form $GF(2^n)$ is of **characteristic** 2. (See Section 14.3 for what is meant by the **characteristic** of a field.)

- As mentioned earlier, the elliptic curve we showed earlier ($y^2 = x^3 + ax + b$) is meant to be used only when the underlying finite field is of characteristic **greater** than 3. (See Section 14.5)

- The elliptic curve equation to use when the underlying field is described by $GF(2^n)$ is

$$y^2 \ + \ xy \ = \ x^3 \ + \ ax^2 \ + b, \qquad b \neq 0 \qquad (18)$$

  The constraint $b \neq 0$ serves the same purpose here that the constraint $4a^3 + 27b^2 \neq 0$ did for the case of the elliptic curve equation $y^2 = x^3 + ax + b$. The reason for the constraint $b \neq 0$ is that the discriminant becomes 0 when $b = 0$. As mentioned earlier, when the discriminant becomes zero, we have multiple roots at the same point, causing the derivative of the curve to become ill-defined at that point. In other words, the curve has a singularity at the point where discriminant is 0.

- Shown in Figure 3 are six elliptic curves described by the analytical form $y^2 + xy \ = \ x^3 + ax^2 + b$ for different values of the parameters a and b. The four upper curves are non-singular. The parameters a and b for the top-left curve are 2 and 1, respectively. The same parameters for the top-right curve are 2 and -1, respectively. For the two non-singular curves in the middle row, the one on the left has 0 and 2 for its a and b parameters, whereas the one on the right has -3 and 2. The two curves in the bottom row are both singular, but for different reasons. The one on the left is singular because b is set to 0. As the next section will show, this is a sufficient condition for the discriminant of an elliptic curve (of the kind being studied in this section) to be singular. However, as the next section explains, it is possible for the discriminant of such curves to be singular even when b is not zero. This is demonstrated by the curve on the right in the bottom row.

- The fact that the equation of the elliptic curve is different when the underlying field is $GF(2^n)$ introduces the following changes in the behavior of the group operator:

  – Given a point $P = (x, y)$, we now consider the negative of this point to be located at $-P = (x, -(x + y))$.

  – Given two distinct points $P = (x_P, y_P)$ and $Q = (x_Q, y_Q)$, the addition of the two points, represented by (x_{P+Q}, y_{P+Q}), is now given by

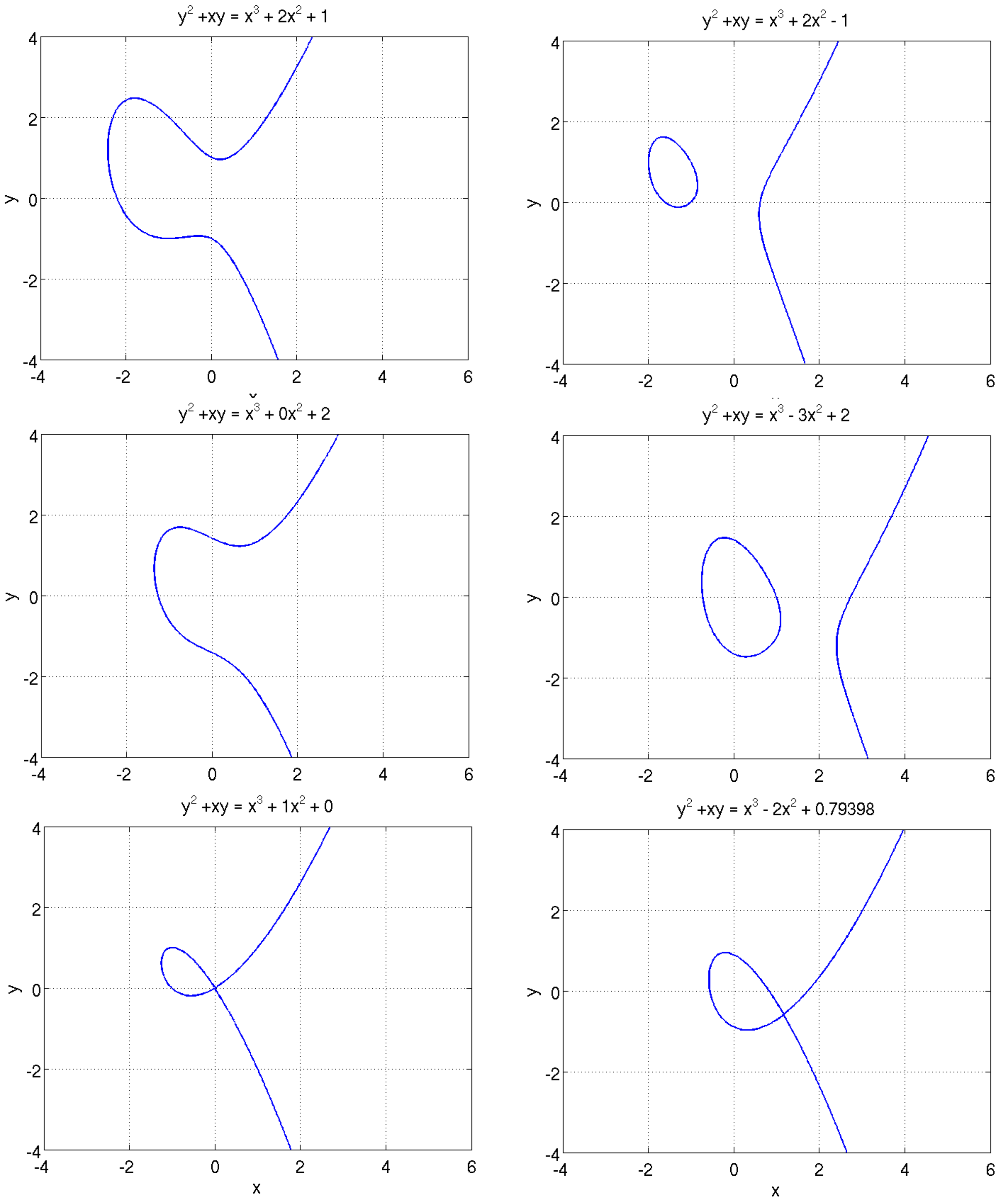
$$\begin{aligned} x_{P+Q} &= \alpha^2 + \alpha - x_P - x_Q - a \\ y_{P+Q} &= -\alpha(x_{P+Q} - x_P) - x_{P+Q} - y_P \end{aligned} \quad (19)$$

with

$$\alpha = \frac{y_Q - y_P}{x_Q - x_P} \quad (20)$$

  – To double a point, that is to calculate $2P$ from P, we now use the formulas

$$\begin{aligned} x_{2P} &= \alpha^2 + \alpha - a - 2x_P \\ y_{2P} &= -\alpha^2 - \alpha + a + (2 + \alpha)x_P - \alpha x_{2P} - y_P \end{aligned} \quad (21)$$



$y^2 + xy = x^3 + 2x^2 + 1$

$y^2 + xy = x^3 + 2x^2 - 1$

$y^2 + xy = x^3 + 0x^2 + 2$

$y^2 + xy = x^3 - 3x^2 + 2$

$y^2 + xy = x^3 + 1x^2 + 0$

$y^2 + xy = x^3 - 2x^2 + 0.79398$

Figure 3: *Elliptic curves meant to be used with Galois fields.*

(*This figure is from Lecture 14 of “Lecture Notes on Computer and Network Security” by Avi Kak.*

with

$$\alpha \quad = \quad \frac{3{x_P}^2 + 2ax_P - y_P}{2y_P + x_P} \tag{22}$$

This value of α is obtained by differentiating both sides of $y^2 + xy \ = \ x^3 + ax^2 + b$ with respect to x and writing down an expression for $\frac{dy}{dx}$ just as we derived the expression for α in Equation (11) in Section 14.7.

– Since the results for doubling shown in Equation (21) *can* be obtained (although the style of derivation shown in Section 14.7 is to be preferred) from those in Equation (19) by letting x_Q approach x_P, which in our case can be simply accomplished by setting $x_Q = x_P$, the reader may be puzzled by the very different appearances of the expressions shown for y_{P+Q} and y_{2P}. If you set $x_Q = x_P$ in the expression for y_{P+Q}, then both the y-coordinate expressions can be shown to reduce to $-\alpha^3 - 2\alpha^2 + \alpha(3x_P + a - 1) + 2x_P + a - y_P$.

[The expressions shown in Equations (19) through (22) are derived in a manner that is completely analogous to the derivation presented in Sections 14.6 and 14.7. As before, we recognize that the points on a straight line passing through two points (x_P, y_P) and (x_Q, y_Q) are given by $y = \alpha x + \beta$ with $\alpha = \frac{y_Q \ - \ y_P}{x_Q \ - \ x_P}$. To find the point of intersection of such a line with the elliptic curve $y^2 + xy \ = \ x^3 + ax^2 + b$, as before we form the equation

$$(\alpha x \ + \ \beta)^2 \ + \ x(\alpha x + \beta) \quad = \quad x^3 \ + \ ax^2 \ + \ b \tag{23}$$

which can be expressed in the following form as a monic polynomial:

$$x^3 \ + \ (a - \alpha^2 - \alpha)x^2 \ + \ (-2\alpha\beta - \beta)x \ + \ (b \ - \ \beta^2) \quad = \quad 0 \tag{24}$$

Reasoning as before, this cubic equation can have at most three roots, of which two are already known, those being the points P and Q. The remaining root, if its exists, must correspond to the point to the point R, which the point where the straight line passing through P and Q meets the curve again. Again using the property that the sum of the the roots is equal to the negative of the coefficient of the second highest power, we can write

$$x_P \ + \ x_Q \ + \ x_R \ = \ \alpha^2 + \alpha - a$$

We therefore have the following result for the x-coordinate of R:

$$x_R \ = \ \alpha^2 \ + \ \alpha \ - \ a \ - \ x_P \ - \ x_Q \qquad (25)$$

Since this point must be on the straight line $y = \alpha x + \beta$, we get for the y-coordinate at the point of intersection $y_R \ = \ \alpha x_R + \beta$. Substituting for β from the equation $y_P \ = \ \alpha x_P + \beta$, we get the following result for y_R:

$$y_R \ = \ \alpha(x_R - x_P) + y_P \qquad (26)$$

Earlier we stated that for the elliptic curves of interest to us in this section, the negative of a point $R \ = \ (x_R, \ y_R)$ is given by $-R \ = \ (x_R, \ -(x_R + y_R))$. Since the point (x_{P+Q}, y_{P+Q}) is located at the negative of the point R at (x_R, y_R), we can write the following result for the summation of the two points P and Q:

$$\begin{aligned} x_{P+Q} & \ = \ x_R \ = & \alpha^2 \ + \ \alpha \ - \ x_P \ - \ x_Q \ - \ a \\ y_{P+Q} & \ = \ -(x_R + y_R) \ = & -\alpha(x_{P+Q} \ - \ x_P) \ + \ x_{P+Q} \ - \ y_P \end{aligned} \qquad (27)$$

The result for doubling of a point can be derived in a similar manner.

Figure 4 shows these operations in action. The two figures in the topmost row show us calculating $P + Q$ for the two points P and Q as shown. The figure on the left in the middle row shows the doubling of a point and the figure on the right the tripling of a point. Shown in the bottom row are the operations of doubling and tripling a point.]

- We will use the notation $E_{2^n}(a, b)$ to denote the set of all points $(x, y) \in GF(2^n) \times GF(2^n)$, that satisfy the equation

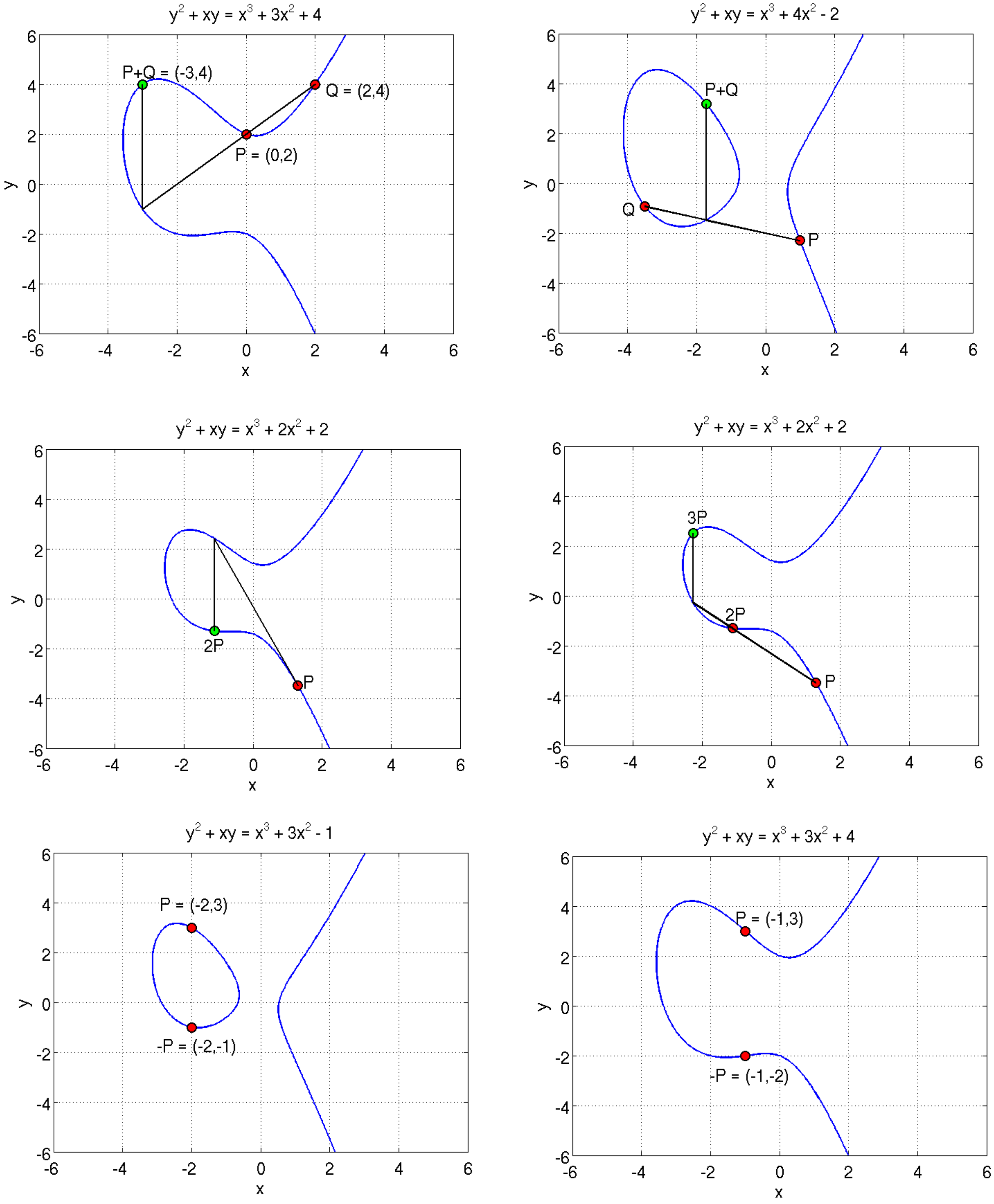


Figure 4: *Group law on the elliptic curves for Galois fields.*

(*This figure is from Lecture 14 of "Lecture Notes on Computer and Network Security" by Avi Kak.*)

$$y^2 \ + \ xy \ = \ x^3 \ + \ ax^2 \ + \ b,$$

with $a \in GF(2^n)$ and $b \in GF(2^n)$, along with the distinguished point **O** that serves as the additive identity element for the group structure formed by the points on the curve. Note that we do not allow b in the above equation to take on the value which is the additive identity element of the finite field $GF(2^n)$.

- If g is a generator for the field $GF(2^n)$ (see Section 7.12 of Lecture 7 for what is meant by the generator of a finite field), then all the element of $GF(2^n)$ can be expressed in the following form

  $$0, \ 1, \ g, \ g^2, \ g^3, ..., g^{2^n-2}$$

  This implies that the majority of the points on the elliptic curve $E_{2^n}(a, b)$ can be expressed in the form $(g^i, \ g^j)$, where $i, j \ = \ 0, \ 1, \ \ldots, \ n-2$. In addition, there may be points whose coordinates can be expressed $(0, g^i)$ or $(g^i, 0)$, with $i \ = \ 0, \ 1, \ \ldots, \ n-2$. And then there is, of course, the distinguished point **O**.

- The **order of an elliptic curve**, that is the number of points in the group $E_{2^n}(a, b)$ **is important from the standpoint of the cryptographic security of the curve.** [Note: When we talk about the order of $E_{2^n}(a, b)$, we must of course include the distinguished point **O**.]

- Hasse's Theorem addresses the question of how many points are on an elliptic curve that is defined over a **finite** field. This theorem says that if N is the number of points on $E_q(a,b)$ when the curve is defined on a finite field Z_q with q elements, then N is bounded by

$$|N \ - \ (q \ + \ 1)| \quad \leq \quad 2\sqrt{q}$$

What this says that the number of points, N, on an elliptic curve must be in the interval $[q+1-\sqrt{q},\ q+1+\sqrt{q}]$. As mentioned previously, N includes the additive identity element $\mathbf{O}$.

- Since the Galois field $GF(2^n)$ contains 2^n elements, we can say that the **order** of $E_{2^n}(a,b)$ is equal to $2^n \ + \ 1 \ - \ t$ where t is a number such that $|t| \leq \sqrt{2^n}$.

- An elliptic curve defined over a Galois Field $GF(2^n)$ is **supersingular** if $2|t$, that is if 2 is a divisor of t. [Supersingularity is **not** to be confused with singularity. As previously explained in Section 14.5, when an elliptic curve is defined over real numbers, singularity of the curve is related to its smoothness. More specifically, a curve is singular if its slope at a point is not defined. **Supersingularity**, on the other hand, is related to the order of E_{2^n} and how this order relates to the number of points in the underlying finite field.]

- Should it happen that $t = 0$, then the order of E_{2^n} is $2n + 1$. Since this number is always odd, such a curve can never be supersingular. Supersingular curves defined over fields of characteristic

2 (which includes the binary finite fields $GF(2^n)$) always have an odd number of points, including the distinguished point $\mathbf{O}$.

- Supersingular curves are to be avoided for cryptography because they are vulnerable to the MOV attack. More on that in Section 14.14.

- The set $E_{2^n}(a, b)$ of points constitutes a group, with the group operator as defined by Equations (19) through (22).

14.10: IS $b \neq 0$ A SUFFICIENT CONDITION FOR THE ELLIPTIC CURVE $y^2 + xy = x^3 + ax^2 + b$ TO NOT BE SINGULAR?

- In general, we want to avoid using **singular** elliptic curves for cryptography for reasons already indicated.

- In Section 14.9 we indicated that when using a curve of form $y^2 + xy = x^3 + ax^2 + b$, you want to make sure that $b \neq 0$ since otherwise the curve will be singular.

- We will now consider in greater detail when exactly the curve $y^2 + xy = x^3 + ax^2 + b$ becomes singular for the case when the underlying field consists of real numbers. Toward that end we will derive an expression for the discriminant of a polynomial that is singular if and only if the curve $y^2 + xy = x^3 + ax^2 + b$ is singular. The condition which will prevent the discriminant going to zero will be the condition under which the curve $y^2 + xy = x^3 + ax^2 + b$ will stay nonsingular.

- To meet the goal stated above, we will introduce the coordinate transformation

$$y \quad = \quad Y \; - \; \frac{x}{2}$$

in the equation

$$y^2 \quad + \quad xy \quad = \quad x^3 \quad + \quad ax^2 \quad + b$$

- The purpose of the coordinate transformation is to get rid of the troublesome term xy in the equation. Note that this coordinate transformation cannot make a singularity disappear, and neither can it introduce a new singularity. With this transformation, the equation of the curve becomes

$$Y^2 \; - \; \frac{x^2}{4} \quad = \quad x^3 \quad + \quad ax^2 \quad + \quad b$$

which can be rewritten as

$$Y^2 \quad = \quad x^3 \quad + \quad (a + \frac{1}{4})x^2 \quad + \quad b$$

The polynomial on the right hand side of the equation shown above has a singular point wherever its discriminant goes to zero.

- In general, the discriminant of the polynomial

$$a_3z^3 \quad + \quad a_2z^2 \quad + \quad a_1z \quad = \quad 0$$

is given by

$$D_3 \quad = \quad a_1^2 a_2^2 \ - \ 4a_0 a_2^3 \ - \ 4a_1^3 a_3 \ + \ 18a_0 a_1 a_2 a_3 \ - \ 27a_0^2 a_3^2$$

- Substituting the coefficient values for our case, $a_3 = 1$, $a_2 = (a + \frac{1}{4})$, $a_1 = 0$, and $a_0 = b$, in the general formula for the discriminant of a cubic polynomial, we get for the discriminant

  $$D_3 \quad = \quad -\,4b\left(a + \frac{1}{4}\right)^3 \ - \ 27b^2$$

  This simplifies to

  $$D_3 \quad = \quad \frac{1}{16}\left[-64a^3b \ - \ 48a^2b \ - \ 12ab \ - \ b \ - \ 432b^2\right]$$

  which can be expressed as

  $$D_3 \quad = \quad -\,\frac{1}{16}b\left[64a^3 \ + \ 48a^2 \ + \ 12a \ + \ 432b \ + \ 1\right]$$

- Therefore, if $b = 0$, the discriminant will become 0. However, it should be obvious that even when the $b = 0$ condition is not satisfied, certain values of a and b may cause the discriminant to go to 0.

- As with the supersingular curves, elliptic curves that are singular are to be avoided for cryptography because they are vulnerable to the MOV attack described in Section 14.14.

14.11: ELLIPTIC CURVE CRYPTOGRAPHY — THE BASIC IDEA

- That elliptic curves over finite fields could be used for cryptography was suggested independently by Neal Koblitz (University of Washington) and Victor Miller (IBM) in 1985.

- Just as RSA uses multiplication as its basic arithmetic operation (exponentiation is merely repeated multiplication), ECC uses the "addition" group operator as its basic arithmetic operation (multiplication is merely repeated addition).

- Suppose G is a user-chosen "base point" on the curve $E_q(a, b)$, where $q = p$ for some prime p when the underlying finite field is a prime finite field and $q = 2^n$ when the underlying finite field is a Galois field.

- In accordance with how the group operator works, $k \times G$ stands for $G + G + G + \ldots + G$ with G making k appearances in this expression.

- The core notion that ECC is based on is that, with a proper choice for G, whereas it is relatively easy to calculate $C = M \times G$, it can be extremely difficult to recover M from C even when an adversary knows the curve $E_q(a, b)$ and the G used. As explained earlier in Section 14.2, recovering M from C is referred to as having to solve the **discrete logarithm** problem. [On the basis of the comment made earlier in Section 14.2 regarding "discrete logarithms," determining the **number of times** G participates in $C = G \circ G \circ G \circ \ldots \circ G$, where '$\circ$' is the group operator, can be thought of as taking the "logarithm" of C to the base G.]

- An adversary could try to recover M from $C = M \times G$ by calculating $2G$, $3G$, $4G$, . . ., kG with k, in the worst case, spanning the size of the set $E_q(a, b)$, and then seeing whether or not the result matched C. But if q is sufficiently large and if the point G on the curve $E_q(a, b)$ is chosen carefully, that would take much too long.

- As you'll see in the next section, we do not directly use for encryption the repeated additions as expressed by $M \times G$. In the next section, we will use these forms in a Diffie-Hellman based approach to cryptography with elliptic curves.

14.12: ELLIPTIC CURVE DIFFIE-HELLMAN SECRET KEY EXCHANGE

- The reader may wish to first review Section 13.5 of Lecture 13 before proceeding further. The Diffie-Hellman idea was first introduced in that section. This section introduces the Elliptic-Curve Diffie-Hellman (ECDH) algorithm for establishing a secret session key between two parties. [You may see two acronyms used in connection with this algorithm — ECDH and ECDHE — to reflect how it used. The acronym ECDHE officially stands for "Elliptic Curve Diffie-Hellman Ephemeral." If the key exchange described in this section is used in conjunction with authentication provided by, say, RSA-based certificates, the combined algorithm may be shown as ECDHE-RSA, although it should really be designated as just ECDH-RSA. The word "ephemeral" is supposed to capture the situation when there is no authentication between to parties and they just want a session key on a one-time basis.]

- A community of users wishing to engage in secure communications with ECC chooses the parameters q, a, and b for an elliptic-curve based group $E_q(a, b)$, and a base point $G \in E_q(a, b)$.

- A selects an integer X_A to serve as his/her private key. A then generates $Y_A = X_A \times G$ to serve as his/her public key. A makes publicly available the public key Y_A.

- B designates an integer X_B to serve as his/her private key. As was done by A, B also calculates his/her public key by $Y_B = X_B \times G$.

- In order to create a shared secret key (that could subsequently be used for, say, a symmetric-key based communication link), both A and B now carry out the following operations:

  - A calculates the shared session key by

$$K = X_A \times Y_B \tag{28}$$

  - B calculates the shared session key by

$$K = X_B \times Y_A \tag{29}$$

  - The calculations in Eqs. (19) and (20) yield the same result because

$$\begin{aligned} K \text{ as calculated by } A &= X_A \times Y_B \\ &= X_A \times (X_B \times G) \\ &= (X_A \times X_B) \times G \\ &= (X_B \times X_A) \times G \\ &= X_B \times (X_A \times G) \end{aligned}$$

$$\begin{aligned} &= X_B \times Y_A \\ &= K \;\; \text{as calculated by } B \end{aligned}$$

- To discover the secret session key, an attacker could try to discover X_A from the publicly available base point G and the publicly available Y_A. Recall, $Y_A = X_A \times G$. But, as already explained in Section 14.11, this requires solving the discrete logarithm problem which, for a properly chosen set of curve parameters and G, can be extremely hard.

- To increase the level of difficulty in solving the discrete logarithm problem, we select for G a base point whose **order** is very large. The **order** of a point on the elliptic curve is the **least number of times** G must be added to itself so that we get the **identity element O** of the group $E_q(a, b)$. [We can also associate the notion of **order** with an elliptic curve over a finite field: The **order of an elliptic curve** is the total number of points in the set $E_q(a, b)$. This order is denoted $\#E_q(a, b)$.]

- Since the integers X_A, Y_A, X_B, and Y_B must all be less than the order of the base point G, the value of the order of the base point must also be made publicly available.

- The base point G is also known as the **generator** of a **subgroup** of $E_q(a, b)$ whose elements are all given by G, $2G$, $3G$, . . ., and, of course, the identity element **O**. For the size of the subgroup to equal the **degree** of the generator G, the value of n must be a prime when the underlying field is a Galois field $GF(2^n)$.

14.13: ELLIPTIC-CURVE DIGITAL SIGNATURE ALGORITHM (ECDSA)

- This is the ECC version of the digital signature algorithm presented in Section 13.6 of Lecture 13. This algorithm, known more commonly by its acronym ECDSA, has been much in the news lately because of its use for code authentication in PlayStation3 game consoles. Code authentication means that the digital signature of a binary file is checked and verified before it is allowed to be run on a processor.

- Paralleling our earlier description in Section 13.6 of Lecture 13, the various steps of ECDSA are:

  - For a digital signature based on an elliptic curve defined over a prime finite field Z_p, select a large prime p and choose the parameters a and b for the curve, and base point G of high order n, meaning that $n \times G = \mathbf{O}$ for a large n. [For high security work, you would want to choose the curve parameters as recommended in the NIST document FIPS 186-3 available from `http://csrc.nist.gov/publications/fips/fips186-3/fips_186-3.pdf`.] Now randomly select X, $1 \leq x \leq n-1$, to serve as your private key.

– Next you calculate your public key Y by

$$Y \quad = \quad X \times G$$

where the "multiplication" operation is according to the group law for the elliptic curve. [For its implementation in Python, see the function `k_times_point(curve, point, k, mod)` in the code shown in Section 14.8.] Note that the public key consists of a pair of numbers that are the coordinates of the point Y on the elliptic curve.

– You will make p, a, b, G, n and Y publicly available and you will treat X as your private key.

– Generate a one-time random number K such that $0 < K < n - 1$. By one-time we mean that you will discard K after each use. That is, each digital signature you create will be with a different K. [You must discard K after each use. Using the same K for two different signatures is a major security breach in the use of this algorithm, as will be explained later.]

– Now you are ready to construct a digital signature of a document. Let M be an integer that represents a hash of the document you want to sign. (See Lecture 15 on hashing functions.)

– The digital signature you construct for M will consist of two parts that we will denote sig_1 and sig_2. You construct sig_1 by first calculating the elliptic curve point $K \times G$ and retaining only its x-coordinate modulo n:

$$sig_1 \quad = \quad (K \times G)_x \;\; mod\; n$$

Should the modulo operation produce a zero value for sig_1, you try a different value for K. You next construct sig_2 by

$$sig_2 \;\; = \;\; K^{-1} \cdot (M \; + \; X \cdot sig_1) \qquad mod\; n$$

where K^{-1} is the multiplicative inverse of K modulo n that can be obtained with the Extended Euclid's Algorithm (See Sections 5.6 and 5.7 of Lecture 5).

- Let's say you have sent your document along with its signature (sig_1, sig_2) to some recipient and the recipient wishes to make sure that he/she is not receiving a modified message. The recipient can verify the authenticity of the document by (a) first calculating its hash M of the document (using the same algorithm that you did); (b) calculating the numbers $w = sig_2^{-1}\; mod\; n$, $u_1 = M \cdot w\; mod\; n$, and $u_2 = sig_1 \cdot w\; mod\; n$; (c) using these numbers to compute the point $(x, y) = u_1 \times G + u_2 \times Y$ on the curve, where the operator '×' is the "multiplication" operator corresponding to the repeated invocations of the group law; and, finally, authenticating the signature by checking whether the equivalence $sig_1 \equiv x\; (mod\; n)$ holds.

- We will now address the danger of using the same K for two different documents — danger in the sense that an adversary can figure out your private key and then proceed to counterfeit your signature. Let the hashes of two different documents you are signing with the same K value be M and M'. The two signatures for these two documents will look like:

$$\begin{aligned}
sig_1 &= (K \times G)_x \ \ mod\ n \\
sig_2 &= K^{-1} \cdot (M \ - \ X \cdot sig_1) \quad\ mod\ n \\
sig_1' &= (K \times G)_x \ \ mod\ n \\
sig_2' &= K^{-1} \cdot (M' \ - \ X \cdot sig_1') \quad\ mod\ n
\end{aligned}$$

where the primed signatures are for the second document. Note that sig_1 and sig_1' remain the same because they are independent of the document. Therefore, if an adversary were to calculate the difference $sig_2 - sig_2'$, he would obtain

$$sig_2 - sig_2' \quad = \quad K^{-1}(M - M')$$

From this, the adversary can immediately calculate the value of K you used for your digital signature. And, using the equation $sig_2 = K^{-1} \cdot (M - X \cdot sig_1)\ mod\ n$, the adversary can proceed to calculate your private key X. [This was the ploy used to break the ECDSA based code authentication in PlayStation3 a couple of years back.]

- For a proof of the ECDSA algorithm, see the paper "The Elliptic Curve Digital Signature Algorithm (ECDSA)" by Don Johnson, Alfred Menezes, and Scott Vanstone that appeared in International Journal of Information Security, pp. 36-63, 2001. ECDSA as a standard is described in the document ANSI X9.62.

14.14: SECURITY OF ECC

- Just as RSA depends on the difficulty of large-number factorization for its security, ECC depends on the difficulty of the large number discrete logarithm calculation. This is referred to as the **Elliptic Curve Discrete Logarithm Problem** (ECDLP).

- It was shown by Menezes, Okamoto, and Vanstone (MOV) in 1993 that (for supersingular elliptic curves) the problem of solving the ECDLP problem (where the domain is the group $E_q(a, b)$) can be reduced to the much easier problem of finding logarithms in a finite field. There has been much work recently on extending the MOV reduction to general elliptic curves.

- In order to not fall prey to the MOV attack, the underlying elliptic curve and the base point chosen must satisfy what is known as the **MOV Condition**.

- The MOV condition is stated in terms of the **order** of the base point G. The order m of the base point G is the value of m such that $m \times G = \mathbf{O}$ where $\mathbf{O}$ is the additive identity element of the group $E_q(a, b)$ as defined in Section 14.4.

- The MOV condition states that the **order** m of the base-point should not divide $q^B - 1$ for small B, say for $B < 20$. Note that q is the prime p when the underlying finite field is Z_p or it is 2^n when the underlying finite field is $GF(2^n)$.

- When using $GF(2^n)$ finite fields, another security consideration relates to what is known as the **Weil descent attack**. To not be vulnerable to this attack, n must be a prime.

- Elliptic curves for which the total number of points on the curve equals the number of elements in the underlying finite field are also considered cryptographically weak.

14.15: ECC FOR DIGITAL RIGHTS MANAGEMENT

- ECC has been and continues to be used for Digital Rights Management (DRM). DRM stands for technologies/algorithms that allow a content provider to impose limitations on the whos and hows of the usage of some media content made available by the provider.

- ECC is used in the DRM associated with the Windows Media framework that is made available by Microsoft to third-party vendors interested in revenue-generating content creation and distribution. In what follows, we will refer to this DRM as **WM-DRM**.

- The three main versions of WM-DRM are Version 1 (released in 1999), Version 2 (released in 2003, also referred to as Version 7.x and Version 9), and Version 3 (released in 2003, also known as Version 10). All three versions have been cracked. As you would expect in this day and age, someone figures out how to strip away the DRM protection associated with, say, a movie and makes both the unprotected movie and the protection stripping algorithm available anonymously on the web. In the meantime, the content provider (like Apple, Sony, Microsoft, etc.) comes out with a patch to fix the exploit. Thus continues the cat and mouse game between the big content providers and the anonymous "crackers."

- Again as you would expect, the actual implementation details of most DRM algorithms are proprietary to the content providers and distributors. But, on October 20, 2001, an individual, under the pseudonym Beale Screamer, posted a detailed description of the inner workings of the WM-DRM Version 2. This information is still available at the URLs `http://cryptome.org/ms-drm.htm` and `http://cryptome.org/beale-sci-crypt.htm` where you will find a command-line tool named **`FreeMe`** for stripping away the DRM protection of the older versions of Windows Media documents. Since Version 2 is now considered out of date, the main usefulness of the information posted at the web site lies in its educational value.

- WM-DRM Version 2 used elliptic curve cryptography for exchanging a secret session key between a user's computer and the license server at the content provider's location. As to how that can be done, you have already seen the algorithm in Section 14.12.

- The ECC used in WM-DRM V. 2 is based on the first elliptic curve $y^2 = x^3 + ax + b$ that was presented in Section 14.3. The ECC algorithm used is based on the points on the curve whose x and y coordinates are drawn from the multiplicative group

Z_p^*, defined earlier in Section 11.8 of Lecture 11, Section 13.5 of Lecture 13, and Section 14.8 of this lecture, with the number p set to the value shown below: [Recall from Section 11.8 of Lecture 11 and Section 13.5 of Lecture 13 that the multiplicative group Z_p^* consists of the $p-1$ integers $\{1, 2, 3, \cdots, p-1\}$]

$$p \quad = \quad 785963102379428822376694789446897396207498568951$$

In the WM-DRM ECC, all are represented using 20 bytes. Here is the hex representation of the modulus p shown above:

$$p \quad = \quad 0x89abcdef012345672718281831415926141424f7$$

- We also need to specify values for the parameters a and b of the elliptic curve $y^2 \ = \ x^3 \ + \ ax \ + \ b$. As you would expect, these parameters are also drawn from the multplicative group Z_p^* and their values are given by

$$\begin{aligned} a \quad &= \quad 317689081251325503476317476413827693272746955927 \\ b \quad &= \quad 790528966078787587181205720257185354321006519 34 \end{aligned}$$

Since all numbers in the ECC implementation under consideration are stored as blocks of 20 bytes, the hex representations of the byte blocks stored for a and b are

$$\begin{aligned} a \quad &= \quad 0x37a5abccd277bce87632ff3d4780c009ebe41497 \\ b \quad &= \quad 0x0dd8dabf725e2f3228e85f1ad78fdedf9328239e \end{aligned}$$

- Following the discussion in Sections 14.11 and 14.12, the ECC algorithm would also need to choose a base point G on the elliptic curve $y^2 \; = \; x^3 \; + \; ax \; + \; b$. The x and the y coordinates of this point in the ECC as implemented in WM-DRM are

$$\begin{aligned} G_x &= 771507216262649826170648268565579889907769254176 \\ G_y &= 390157510246556628525279459266514995562533196655 \end{aligned}$$

  The 20-byte hex representations for these two coordinates are

$$\begin{aligned} G_x &= 0x8723947fd6a3a1e53510c07dba38daf0109fa120 \\ G_y &= 0x445744911075522d8c3c5856d4ed7acda379936f \end{aligned}$$

- As mentioned in Section 14.12, an ECC protocol must also make publicly available the order of the base point. For the present case, this order is given by

$$\#E_p(a,b) \;=\; 785963102379428822376693024881714957612686157429$$

- With the elliptic curve and its parameters set as above, the next question is how exactly the ECC algorithm is used in WM-DRM.

- When you purchase media content from a Microsoft partner peddling their wares through the Window Media platform, you would need to download a “license” to be able play the content on your computer. Obtaining the license consists of your computer randomly generating a number $n \in Z_p$ for your computer's private key. Your computer then multiplies the base point G with the private key to obtain the public key. Subsequently your computer can interact with the content provider's license server in the manner described in Section 14.12 to establish a secret session key for the transfer of license related information into your computer.

- In order to ensure that only your computer can use the downloaded license, WM-DRM makes sure that you cannot access the private key that your computer generated for the ECC algorithm. Obviously, if you could get hold of that n, you could pass the encrypted content file and the private key to your friend and they would be able to pretend to be you vis-a-vis the license server. WM-DRM hides an RC4 encrypted version of the private key in the form of a linked list in which each nodes stores one half of the key.

- When DRM software is cracked, it is usually done by what is known as "hooking" the DRM libraries on a computer as they dump out either the keys or the encrypted content.

14.16: HOMEWORK PROBLEMS

1. ECC uses numbers that correspond to points on elliptic curves. What is an elliptic curve? Does it have anything to do with an ellipse?

2. What is the geometrical interpretation of the group law that is used for the numbers drawn from the elliptic curves in ECC?

3. What is the fundamental reason for why ECC can use shorter keys for providing the same level of security as what RSA does with much longer keys?

4. Section 14.13 described the ECDSA algorithm (which, as was mentioned in Section 14.1, is used for authentication in ECC based certificates). One significant disadvantage of ECDSA vis-a-vis an RSA based digital signature algorithm is that the security of ECDSA depends on the quality of the random number generator used for K. Why do you think the security of ECDSA would be compromised if K is generated from a low-entropy source?

5. **Programming Assignment:**

   Section 14.8 included Python code (unoptimized for large primes) that implemented the group law for the set of points on standard-form elliptic curves over prime finite fields. Extend this code with the implementations required for the different algorithmic steps of the ECDSA algorithm of Section 14.13.

Acknowledgments

I'd like to thank Helena Verrill and Subhash Kak for sharing their insights with me on the mathematics of elliptic curves and on the subject of elliptic curve cryptography. Helena Verrill is the source of much of the information provided regarding the singularity and supersingularity of elliptic curves.