Curves for mere mortals

Ron Garret

*** DRAFT 12/9/16 ***

NOTE: This is a work in progress

Overview

This is a survey of elliptic curve cryptography targeted towards a general technical audience. I'm going to assume you are comfortable with high-school algebra, know something about computer programming, and have some passing familiarity with ideas like coordinate transforms and modular arithmetic, but no higher math (i.e. I'm not going to expect you to know what a group or a field is).

The motivation for this survey is my own frustration in trying to come up to speed on elliptic curves. The motivations for features of various curves -- indeed the very fact that there *are* features to curves, and why they matter -- are often obscured in the literature. This makes it hard to bridge the gap between elementary tutorials and the research literature.

For example, if you read the Wikipedia article you will learn that an elliptic curve is an equation of the form:

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

If you consult the NIST standard you will see that this is confirmed, but that *a* is always set to be equal to 3. But if you read the curve25519 paper the formula you will find there is:

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

or the Ed25519 paper which uses an equation of the form:

$$-x^2 + y^2 = dx^2y^2 + c$$

If, like me, you find yourself wondering what the connection is between these various formulas (and others you will find when you start groveling through the literature), what the advantages and disadvantages are of using one over the other, and if there is a simple conceptual framework that ties them all together (there is), this paper is for you.

Road map

Cryptography is based fundamentally on *mathematically hard problems*, that is, mathematical problems for which no efficient method of finding solutions is known. The formulas for elliptic curves are the result of trying to design the simplest possible algebraic equations that produce

mathematically hard problems. The fully general form of an elliptic curve is a polynomial equation in x and y that contains every possible combination of exponents, i.e.:

$$Ax^3 + Bx^2y + Cxy^2 + Dy^3 + Ex^2 + Fxy + Gy^2 + Hx + Iy + J = 0$$

but it turns out that this fully general form is not needed. Setting most of these terms to zero still results in mathematically hard problems.

You may notice that even the fully general form does not cover the case of Ed25519. That formula contains an x^2y^2 term that does not appear in the general formula. In fact, you may have noticed that the Edwards curve formula appears to be *fundamentally* different from the general formula, as the general formula is cubic and the Edward formula is quartic. This is an illusion. The Edwards formula is derived from the general formula by a change of variables that introduces a new degree of freedom, so the "x" and "y" in the Edwards formula have different semantics than they do in the general formula. In fact, it would be more accurate to write the Edwards formula as:

$$Au^2 + v^2 = Bu^2v^2 + c$$

where $u = f(x, y)$ and $v = g(x, y)$ for some appropriately defined functions f and g. We'll get to all that later.

The specific mathematically hard problem that elliptic curves produce is called the elliptic curve discrete logarithm problem. This is actually somewhat of a misnomer. It would be more accurate to call the particular problem of concern here the "elliptic curve discrete division problem", but it is invariably referred to as the discrete log problem (DLP). This is because it is closely related to another problem which *is* appropriately called the discrete log problem.

Recall the the plain-vanilla logarithm problem is, given:

$$x = b^e$$

find e given x and b. The value b is called the *base* of the logarithm, i.e. if $x = b^e$ then $e = log_b(x)$.

Finding logarithms of real numbers is an easy (in the mathematical sense) problem. Logarithms can be computed to any desired level of precision in polynomial time by computing the sum of a converging infinite series like the Taylor series.

To convert this into a mathematically hard problem we make two conceptually simply changes. First, instead of computing logarithms over real numbers we instead compute logarithms over *modular integers*. It turns out that this alone is enough to produce a mathematically hard problem, and indeed this one change is the basis for practical cryptographic systems like Diffie-Hellman key exchange and RSA public-key cryptography. But there are some practical difficulties with crypto systems based on this one change alone, and so there are benefits to introducing a second change, which is to replace the standard numerical multiplication

operation with a different mathematical operation called *point addition*. It is in this second change where elliptic curves enter the picture.

Because the structure of the math is identical with and without elliptic curves, things are much easier to understand if you first understand the discrete logarithm problem *without* elliptic curves, and then add (no pun intended) curves in at at the end. Accordingly I will start by discussing the non-elliptic-curve version of the discrete logarithm problem. But before I get to that I will start with a brief review of modular arithmetic just to make sure everyone is on the same page.

Modular arithmetic

Modular arithmetic is regular arithmetic performed on values constrained to be integers between 0 and some specific value P called the *modulus*. For cryptography, P is almost always a prime number, which is why I am calling it P and not M. Sometimes the math will involved two different prime moduli, in which case I will call them P and Q. You can, of course, do modular arithmetic using non-prime moduli, but prime moduli work best for generating instances of hard mathematical problems, which is what we are trying to do.

Modular arithmetic works exactly like regular arithmetic when it comes to addition, subtraction, and multiplication, except that the result is taken modulo P, i.e. the result is divided by P and the remainder is taken as the result. For example, if P=13, then the result of adding 7 and 8 is 2, because 7+8=15, and the remainder of dividing 15 by 13 is 2. This is written as:

$$7 + 8 = 2 (mod 13)$$

Modular multiplication and subtraction work the same way: you simply do the operation in the usual way, divide the result by P, and take the remainder. But division and square roots are different. When you divide by an integer or take the square root of an integer in the usual way the result is generally not an integer. But the results of modular arithmetic operations *must* be integers between 0 and P. So what do we do if, for example we want to divide 3 by 7?

Division and modular inverses

The conceptual trick for defining how to divide modular integers is to observe that the results of multiplying certain pairs of modular integers is 1. For example:

$$7 \times 2 = 1 (mod 13)$$

Because 1 is the multiplicative identity (i.e. any number multiplied by 1 is itself), therefore, 2 and 7 must be multiplicative inverses of each other, i.e.:

$$1/7 = 2 (mod 13)$$

and

$$1/2 = 7 (mod 13)$$

Let's see if this makes sense:

$$4 \times 7 = 28 = 2 (mod 13)$$

$$6 \times 7 = 42 = 3 (mod 13)$$

$$8 \times 7 = 56 = 4 (mod 13)$$

Notice how the result is exactly what you would expect if you multiplied the left hand side by 1/2. (Exercise for the reader: what happens if you multiply an odd number by 1/2 (mod13)?)

So we can divide by a number N *if* we can find the multiplicative inverse of N, but it is far from clear how to do that. In non-modular arithmetic it is easy to find multiplicative inverses because we are allowed to use rational numbers, so the multiplicative inverse of a/b is always just b/a. But for modular arithmetic all values have to be integers. Intuitively it is not clear that all modular integers even *have* multiplicative inverses, let alone how to actually find them.

One possible way to find multiplicative inverses is to simply try multiplying all possible pairs of integers less than P and see which pairs have products equal to 1 (mod P). This works if P is small, but for real cryptographic operations we are going to be using very large values of P (dozens to hundreds of digits), so this approach won't work.

Happily it turns out that there is an efficient algorithm for computing modular multiplicative inverses when the modulus is a prime number (this is one of the reasons for making P prime). It is called the *extended Euclidean* algorithm, and I won't try to explain it here because the Wikipedia article does a fine job. All that matters for this discussion is that this algorithm *exists*, and hence finding modular multiplicative inverses is *not* a mathematically hard problem. It *is*, however, computationally expensive compared to adding, subtracting, and multiplying, so for the sake of efficiency we do want to avoid computing inverses whenever we can.

It also turns out that we can prove that every integer modulo P has a multiplicative inverse if P is prime.

Modular square roots

Another mathematical operation we want to be able to perform is taking a square root mod some (generally prime) number P. For example:

$$5 \times 5 = 25 = 12 (mod 13)$$

so the square root of 12 (mod 13) is 5. You might want to ponder the problem of how to find square roots mod P before looking at the answer, which is called the Tonelli-Shanks algorithm. Again, the details don't really matter. The only thing that matters is that there exists an algorithm for efficiently computing square roots modulo a prime number P, and so this too is not a

mathematically hard problem. But as with inverses, it is an expensive operation.

Note that unlike modular multiplicative inverses, it is *not* the case that every number has a square root mod P even if P is prime. There is an easy way to test whether or not a number has a square root mod P called Euler's criterion: if we computer $a^{(p-1)/2}(modp)$ the result will be -1 if and only iff a is a square mod p, otherwise the result will be 1. (Note that if you follow the previous link, the term "quadratic residue" is a synonym for "perfect square (mod p)".)

Modular exponentiation

There is one more mathematical operation on modular arithmetic that is important for cryptography, and that is modular exponentiation, i.e. computing b^e. We have already seen this operation in use in the previous section where it was used in the Euler-criterion algorithm to test if a number is a perfect square mod p.

The straightforward way of doing modular exponentiation is to simply multiply b by itself e times, but in cryptographic applications both b and e are typically very large numbers so this approach is not feasible. Instead there is an algorithm called exponentiation by squaring which computes modular exponents with surprising efficiency.

The discrete logarithm problem

The discrete logarithm problem, which is the mathematically hard problem on which we are going to base our cryptographic algorithms, is the inverse of modular exponentiation. The problem is completely analogous to the regular logarithm problem. The only difference is that the regular logarithm problem is tractable (by computing e.g. a Taylor-series approximation) while the discrete logarithm is not.

Exercise: the rules of modular arithmetic are exactly the same as the rules of normal arithmetic. All of the fundamental mathematical operations (addition, subtraction, multiplication, division, exponentiation and square roots) exist and obey most of the same familiar arithmetic laws: commutativity, associativity, distribution, etc. So why can't we compute a Taylor series approximation to the discrete log the same way that we did for the real-number log?

Cyclic groups

The answer to the exercise in the preceding section is that there is one important mathematical law that modular arithemtic does *not* obey, and that is *monotonicity*. In regular arithmetic, if a and b are both greater than 1, then a+b and a*b are always greater than a or b, and a/b and sqrt(a) are always less than a. These facts are what ultimately allow Taylor series to* converge *when computed on real numbers. But monotonicity does* not *hold for modular arithmetic, and so Taylor series do not converge. (Note that this fact alone does* not* prove that the discrete logarithm problem is mathematically hard. In fact, we don't actually know exactly how hard it is. It is possible that an efficient algorithm for computing discrete logs actually exists, just as for modular inverses and square roots, but we just have not yet been clever enough to find it.)

The underlying structure of modular arithmetic is something called a *cyclic group*. A cyclic group is to modular numbers what a *number line* is to regular real numbers. Just as *line* is an abstract concept which can applied to things other than numbers (like geometry), a *group* is also an abstract concept with an astonishingly broad range of applications, most of which are not relevant to us. In this section I'm going to cover just parts that are relevant to elliptic curves.

A *group* is a set of elements together with an associated operation called the *group law*, which takes two elements in the set and combines them to produce a third element in the set. In other words, the group law is a function from pairs of elements in the set onto elements in the set. The group law also has to meet a few other criteria, which I will get to in a moment.

The group law is usually denoted by an infix $\oplus$, i.e. a$\oplus$b denotes the group law being applied to two group elements a and b. But we could just as easily have used more conventional functional notation and written f(a,b) or g(a,b).

In our case, the group elements are going to start out being numbers, and (spoiler alert!) the group law will be modular exponentiation. We are going to build up cryptographic operations using familiar mathematics, and then swap in elliptic curves right at the end. When we do this (again, spoiler alert) the group elements are going to change from numbers to *points*, i.e. ordered pairs of numbers, and the group law is going to switch from modular exponentiation to something called scalar point multiplication. But the point (no pun intended) is that when we make this transition, all of the underlying mathematical *structure* is going to remain exactly the same. In particular, we're still going to have a group. And not just any group, but a particular kind of group: a *finite, cyclic* group, of which the "cyclic" part will turn out to be the most important.

To be a group, the group law must meet four criteria:

1. Closure: the result of applying the group law to any two elements in the group must produce an element in the group.
2. Associativity: a$\oplus$(b$\oplus$c) must be equal to (a$\oplus$b)$\oplus$c
3. There must exist an identity element 0 such that for every group element a, a$\oplus$0 = a.
4. Every group element a must have an inverse a^{-1} such that $a \oplus a^{-1} = 0$.

A *finite* group is, as you might guess, a group whose set of group elements is a finite set. In our case we ensure this by using modular arithmetic, so the elements of our groups will be integers from 0 to p-1 where p is some (usually prime) number.

A *cyclic* group is one where all of the elements of the group can be reached by repeatedly applying the group law to one element of the group. That element is called a *generator*.

The simplest example of a finite cyclic group is the integers modulo p with addition as the group law. The generator for such a group is the number 1, because (obviously) by repeatedly adding 1 to itself you eventually reach every element of the group. But this isn't particularly interesting or useful. A more interesting and useful example is integers modulo some prime p with *multiplication* as the group law instead of addition. Let us take p=13 as an example. It turns out

that 2 is a generator for this group because:

$$2x2 = 4$$

$$2x4 = 8$$

$$2x8 = 16 = 3(mod13)$$

$$2x3 = 6$$

$$2x6 = 12$$

$$2x12 = 24 = 11(mod13)$$

$$2x11 = 22 = 9(mod13)$$

and so on. By proceeding in this manner we generate the sequence:

$$2, 4, 8, 3, 6, 12, 11, 9, 5, 10, 7, 1, 2...$$

And that is really the crux of the matter. Every element in this sequence is a power of 2 (the generator), and so the *index* of each element is its discrete logarithm, i.e. the number of times you need to multiply 2 by itself to get to that element.

Note that a finite cyclic group does not have to be be dense. For example, we could have used 4 as a generator and gotten the sequence:

$$4, 3, 12, 9, 10, 1, 4...$$

This is still a perfectly legitimate cyclic group. (Exercise: prove this, i.e. show that the set {1, 3, 4, 9, 10, 12} satisfies all the criteria for a group with the group law being multiplication modulo 13.)

All public key cryptographic algorithms are based on finite cyclic groups. The trick to making them cryptographically secure is to generate a group with a large enough number of elements so that solving the discrete logarithm problem by brute force is intractable. But this raises an interesting question: if we can't actually carry out the computations to actually generate (i.e. to enumerate all of the elements of) a group, how do we know what to use as a generator, and how big the resulting group actually is?

Finding a generator for a cyclic group

There is no general method for finding a generator of an arbitrary group. However, there are methods for finding generators for particular kinds of groups. The one that is of the most practical interest is called a Schnorr group, which is a group of integers modulo a prime p which has the form nq+1, where q is also prime. For a group of this form, a number h^n is a generator of the group iff $h^n(modp)$ is not equal to 1.

An important special case of a Schnorr group is when n=2, in which case this group is called a Sophie-Germain group. The number $4^q (mod p)$ (i.e. h=2) is always a generator for a Sophie-Germain group.

Schnorr groups and Sophie-Germain groups are generally found by brute-force search. Finding suitable groups for cryptography is very computationally intensive, so it is generally done off-line and the results are often published as standards for re-use.

Diffie-Hellman key exchange

Now that we understand cyclic groups and generators we can easily build our first practical cryptographic algorithm: Diffie-Hellman key exchange (DH). We have already done all the heavy lifting in our exposition of cyclic groups. If you understand those, the DH algorithm is borderline trivial.

Before describing the algorithm itself, let us describe the problem that the algorithm is designed to solve: Alice wants to transmit an encrypted message to Bob. The only communications channel that Alice and Bob have access to is insecure; all of their communications can be monitored (but not modified) by Eve the eavesdropper. In order for Bob to be able to decrypt Alice's message, Alice and Bob first need to agree on a cryptographic key with which the message will be encrypted. But they cannot simply communicate this key directly because if they did then Eve would intercept it.

Instead they do this: First, Alice and Bob agree on a cyclic group with a generator g. Then Alice picks a random number a, and Bob picks a random number b. Alice computes g^a and sends it to Bob, and Bob computes g^b and sends that to Alice. When Alice receives g^b from Bob she computes $(g^b)^a$. When Bob receives g^a from Alice he computes $(g^a)^b$. (All of these copmutations are done modulo p of course.) Note that $(g^a)^b = g^{a+b} = g^{b+a} = (g^b)^a$, so the final result computed by Alice and Bob are the same. But Eve, who knows both g^a and g^b cannot compute this number. to compute $(g^a)^b = (g^b)^a$ she would have to know either a or b, and that would require Eve to solve the discrete logarithm problem.

Curves

Now we are finally in a position to talk about elliptic curves. Elliptic curve cryptography is essentially the same as "ordinary" cryptography, but with modular exponentiation replaced by elliptic curve scalar point multiplication (which we will describe shortly). Everything else stays the same.

As mentioned earlier, an elliptic curve is an algebraic curve with the general form:

$$Ax^3 + Bx^2y + Cxy^2 + Dy^3 + Ex^2 + Fxy + Gy^2 + Hx + Iy + J = 0$$

To achieve security it is not necessary to use all of this complexity, and we can safely set most

of the coefficients to zero. There is a lot of research that goes into figuring out which simplifications produce the best tradeoffs in terms of security versus code complexity and efficiency, but there are three forms which are in widespread pratical use. The first is:

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

usually with a=3. This is the form used by the National Institute of Standards and Technology (NIST) standard curves. The second form is:

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

This is the form used by Curve25519 and some other related curves like Curve448. This form is called a Weierstrass curve.

The third form is called an *Edwards Curve*, and I'll defer the discussion of those to a later section because they involve some more conceptual groundwork that we have yet to lay. There are many other kinds of curves as well (Koblitz curves, Hessian curves, twists, etc. etc. etc.), but the two variants of the Weirstrass curve and Edwards curves are currently the most widely used.

There are a few things to notice about the two equations above: they are both of the form: $y^2 = [\text{a third-degree polynomial in x}]$. Accordingly, their graph is going to look more or less the same. It is going to be symmetric about the x axis (because solving for y involves taking a square root which will have positive and negative values) and, if we restrict ourselves to real numbers, the graph will exist only where the third degree polynomial is positive. So our elliptic curve is going to look like one of the following graphs:

[insert artwork here]

It turns out that the middle graph, the one that represents the transition point between the one with a separate "island" an one without, has a very undesirable property for our purposes, namely, that the curve is not differentiable at the "cusp" where y=0. So we are going to impose a constraint that we not choose parameters that yield curves of this form. That constraint turns out to be:

$$4A^3 + 27B^2 \neq 0$$

The details don't really matter for our purposes, but if you are curious you can find them on Mathworld.

Point addition

There are two key observations about elliptic curves which allow us to define an addition operation on curve points: If you have two distinct points on an elliptic curve P1=(x1, y1) and P2=(x2, y2), then:

1) if x1=x2 then y1 = -y2. This is because, as noted previously, elliptic curves are symmetric

about the x axis.

2) if x1≠x2 then a line between P1 and P2 will intersect the curve at a third point P3, distinct from P1 and p2. This is a little harder to show, but the sketch of the proof is that if you transform the curve so that the line becomes the x axis, the resulting curve is still a cubic and so has three roots. So there must be a third curve point on the line.

With these two facts in mind, let us try to define point addition. Remember that an addition operation must meet the following criteria: it has to be commutative and associative, i.e. for all points P1, P2 and P3 it must be the case that P1+P2=P2+P1 and (P1+P2)+P3=P1+(P2+P3). Futhermore, there must exist an *additive identity* which we will call P0 with the property that P+P0=P for any point P. Finally, every point P must have an additive inverse -P such that P+(-P)=P0. You might want to try playing around with various possibilities to see if you can come up with an operation that meets all these criteria before proceeding.

The answer turns out to be:

1. The points (x,y) and (x,-y) are additive inverses of each other, i.e. if you add two distinct points with the same x coordinate the result is P0.
2. The sum of two points P1 and P2 with different x coordinates is the *additive inverse* of the third point on the line joining P1 and P2.

In other words, to add P1 and P2, draw a line between them, find the third point where that line intersects the curve, and then reflect that point about the x axis to get the result.

This last step--the reflection about the x-axis--often confuses people because of its apparent arbitrariness. Why not just take the third point on the line to be the sum? The answer is that if you try to do this, the resulting operation is not consistent as an addition operation. For example, consider three points on a line, P1, P2 and P3. If the sum of two points is simply the third point without reflection, then we have not only P1+P2=P3 but also P2+P3=P1 and P1+P3=P2. If you do some elementary algebra on those equations you will see that the only way to satisfy them is if P1=P2=P3=P0, which contradicts our assumption that all of these points are distinct.

Another way to think of the addition rule is this: for any line that intersects an elliptic curve at more than one point, the sum of all the points of intersection is the additive identity, P0.

The projective plane

In the literature you will often see P0 referred to as "the point at infinity" and written simply as 0 or $\mathcal{O}$ rather than P0. It can be a little confusing to see 0 referred to as anything having to do with infinity. This section explains the reason for this weird terminology, but is otherwise unimportant. Unless you really want a deep understanding for what is going on under the hood mathematically, feel free to skip this section.

To this point we have been treating P0 as an abstract entity rather than an actual point on the

curve. So let us ask: where exactly *is* P0?

A little reflection (no pun intended) will reveal that it cannot possibly be on the curve itself, because if it were then it could not be the additive identity. Why? Because *by definition* adding P0 to *any* point P produces the *same* point P, but adding two points on the curve *always* produces a point *distinct* from the points being added.

So if P0 is not on the curve, where is it?

Recall that to add points we needed *two* rules, one for the case where we are adding additive inverses (i.e. distinct points with the same x coordinate) and one for the general case. This too is a bit of a wart on our math. Is there any way to get rid of it?

Let's think about what this would involve: we would like to be able to apply the general rule for adding points (draw a line, find the third point of intersection, reflect about the x axis) to the case where the two points have the same x coordinate and so the line between them does not have a third point of intersection. What we want is that by some mathematical magic, P0 somehow *becomes* that third point of intersection.

But this is going to be some dark magic indeed, because P0 has some really weird properties. For example, P0 reflected about the x axis is P0. Naively this would seem to indicate that P0 must be *on* the x axis. Maybe P0 is (one of) the point(a) where the curve intersects the x axis? No, this is not possible, because those points are *not* additive identities, they are just regular points on the curve, and if you add them to some other point you will get a third, distinct point.

It would seem that we have proven that P0 cannot actually exist, but there is a mathematically legitimate way to bring it into being, and that is to *change coordinate systems*. In ordinary Cartesian coordinates, P0 does in fact not exist. But in a *different* coordinate system, called *projective* coordinates, P0 does exist. Changing to projective coordinates is complicated, and the upshot is that everything stays exactly the same except that P0 comes into being in a non-kludgey way (unless, of course, you consider the switch to projective coordinates *itself* to be a kludge). But here is how it works:

Mapping the Cartesian plane onto projective coordinates is done by "wrapping" the plane around a sphere so that the origin is at one pole, and all of the "points at infinity" are mapped onto a single point at the opposite pole. That single point is now *the* (unique) "point at infinity", and *that* point is P0.

This is how it works: start by placing a sphere so that it contacts the Cartesian plane at the origin. Now draw a line from the origin through the center of the sphere to the other side. Where that line intersects the sphere is P0. To map a point P on the Cartesian plane to the sphere, draw a line between P and P0. That line will intersect the sphere at exactly one point, which is the mapping of P into to the sphere, a.k.a. the projective plane.

The important thing is that by doing this mapping, *P0 is now a unique point*. Moreoever, and more importantly, *all elliptic curves constain P0!* and *all lines pass through P0*.

Point doubling

We have left one loose end dangling, and that is: what happens when we add a point *to itself*, i.e. when we double a point? The answer is essentially the same as adding two distinct points, except that instead of taking a line through two distinct points we take the tangent line to the curve at the point we are doubling. That this is the right answer is easily seen by observing that the tangent line is the limit of adding two points that move progressively closer and closer to each other until they become the same point.

Mathematicians, of course, can be much more verbose about this, but that's all you need to know to do cryptography, so I'll leave it at that.

Doubling and addition formulas

So far I have described how to add curve points *conceptually* but I have not described how to actually *compute* these values. Deriving formulas for adding curve points is initially just an exercise in elementary geometry. If you want to add $P_1 = (x_1, y_1)$ and $P_2 = (x_2, y_2)$ on a curve $y^2 = x^3 + ax^2 + bx + c$ you first compute the slope of the line between P_1 and P_2:

$$\lambda = (y_2 - y_1)/(x_2 - x_1)$$

Then find the third point where this line intersects the curve, and invert the sign on its y coordinate:

$$x_3 = \lambda^2 - a - x_1 - x_2$$

$$y_3 = \lambda(x_1 - x_3) - y_1$$

The formulas for point doubling are exactly the same, except that to find the slope you take the derivative of the curve at P=(x,y):

$$\lambda = (3x^2 + 2ax + b)/2y$$

That's it. You can actually use these formulas to do point addition, and hence to do elliptic curve cryptography, and they will give you the right answers, and they will even do it in a reasonable amount of time. But under no circumstances should you actually do this except as an exercise because, as mentioned before, there are myriad details that need to be dealt with in order to make this secure.

Details

Just to give you an idea of what lies ahead, here is a short list of some of the details that need to be dealt with:

1. Representing P0. Recall that P0, the additive identity, is actually "at infinity" and so

cannot be represented as an ordered pair of real numbers (or, in our case, integers, since we will be switching to modular arithmetic).
2. Constant-time operation. It is crucial that cryptographic algorithms not leak secret information through side-channels. A naive implementation of ECC using the straightforward formulas above will leak like a sieve because point doubling involves different mathematical operations than point addition.
3. Efficiency. We want our cryptographic operations to run as fast as possible, and the naive algorithms given above are far from optimal.

But before we deal with these details, let us first finish the program of using elliptic curves to actually do cryptography.

Elliptic curve cryptography

Elliptic curve cryptography is exactly the same as regular cryptography described in section [X] except that instead of generating a cyclic group by multiplying integers, we instead generate such a group by adding curve points. Otherwise everything remains pretty much the same.

In particular, we begin as we did before by switching from real numbers to modular integers, whereupon, again as before, all the equations exactly the same.

Next, we need to find a cyclic group with a generator. The generator is now a point on the curve rather than an integer, and instead of multiplying the generator by itself to generate the group we instead *add* the generator to itself (because that's all we can do with elliptic curve points) to generate the group. The result is that instead of computing modular *exponents* we are *multiplying* the generator by an integer. The resulting mathematically hard problem is still called the discrete logarithm problem even though it really should be called the discrete division problem (because it is the inverse of multiplication rather than exponentiation).

How do we find generators of elliptic curve groups? We cannot simply adapt the algorithms we used for finding generators in "ordinary" cryptography. There is no such thing as a "Schnorr group" or a "Sophie-Germain group" for elliptic curves. We need an entirely new method. The details of how these things are done are beyond the scope of this survey, but finding a generator is done using Lagrange's theorem and finding the group order is done using Schoof's algorithm.

In fact, there are a few other things that change as well when we switch from simple numbers to points on curves. For example, the size of the group we generated with numbers was always the same as our modulus P (we guaranteed this by using Schnorr groups). But this will not necessarily be the case when we switch to curves. We still want to make groups with a prime number of elements, but the number of points on a curve is not in general going to be a prime number.

At this point we have also encountered a notational difficulty: we have overloaded the meaning

of the symbol P to mean both a point on a curve and a prime number. This can lead to potentially catastrophic confusion because points and numbers are incommensurate. Despite the fact that you can perform many of the same operations on curve points as you can on numbers (for example, you can add both curve points and numbers) they are *not* the same thing. Curve points are ordered pairs of numbers (with the exception of P0). So from now on in order to avoid confusion I am going to use a capital P exclusively to denote curve points, and lower case letters p and q to denote prime numbers (because it turns out we will be needing more than one).

So here's the general plan: we're going to pick some elliptic curve by choosing parameters for the coefficients of the polynomial. Then we're going to pick a prime number p and so all of our math on the curve modulo p. We are then going to try to find a point on the curve which, when repeatedly added to itself, generates a finite cyclic group (of points, not numbers) which we will then use to do cryptography. When we did this with numbers, we could choose a generator that produced *all* of the numbers between 0 and p, and so the mathematical modulus we used turned out to be -- by design -- the same as the size of the group we produced. With elliptic curves we still want to produce a group with a prime number of elements, but that number will generally not be equal to p, and so I will denote it as q. It would make sense to call q the "size" of the group, but mathematicians seem to gravitate towards abstruse terminology, so q is called the group's "order" instead.

To take a real example, for Curve25519, p=$2^{255} - 19$ (hence the name of the curve) and q= $2^{252} + 27742317777372353535851937790883648493$. It is crucial not to confuse these two. Mathematical operations on *curve points* are all done modulo p, but *group operations* are all done modulo q. (Just for completeness, the *generator* for the Curve25519 group is the point whose x coordinate is 9. The y coordinate of this point is ...)

Why do we want a group whose order (i.e. its size) is prime? It's because we need to find a generator, and the way we are going to do that is to try points on the curve at random until we find one where qP=P0. When q is prime, this condition insures that P is a generator.

It makes a useful exercise to prove this. Here is a sketch of the proof: by definition, P is a generator of G iff the sequence S: P, 2P, 3P ... qP=P0 contains every element of G. Because the length of S equals the order (size) of G (i.e. q) then if we can show that every element of S is *unique*, then by the pigeonhole principle the sequence must contain every element of G, and so P must be a generator. So it suffices to show that the sequence cannot contain any duplicates. Filling in the rest is left as an exercise, but here is a hint: show that if the sequence contains duplicates then it must consist of a repeating sub-sequence, and so its length cannot be a prime number.

Montgomery curves

Let us summarize what we have learned so far: Public key cryptography in general works by

finding a *group* where the *discrete logarithm problem* is hard. We have seen two ways to find such a group. The first is to generate a Schnorr group, i.e. a group over modular integers which is generated by modular exponentiation. The second is to produce a group using an elliptic curve. We generate an elliptic curve group by counting the number of points n on the curve using Schoof's algorithm, taking the largest prime factor q of n, and then searching for a point P such that qP=P0, the additive identity, a.k.a. the "point at infinity". We then use that group to implement, for example, Diffie-Helman key exchange. We can also use it to implement secure digital signatures. We haven't discussed those yet. I mention them here just for completeness.

That really is it for the big picture. You now know all you need to know to actually implement elliptic curve cryptographic algorithms that will work in the sense that they will produce the correct results. However, algorithms implemented in a straightforward way will *not* be secure. They will also run much more slowly than they need to.

The biggest problem in a straightforward implementation of elliptic curves as we have described them so far is that it would be subject to side-channel attacks. The double-and-add algorithm steps through the bits in the secret key and performs different operations depending on whether the value of the current bit is 1 or 0. These differences can be detected by looking at things like the time and power consumption while the computation is taking place, and that can in turn be used to find the secret key.

The other problem is that the naive formulas for doubling and adding curve points involve division, which is to say, finding modular multiplicative inverses, which is a very expensive operation relative to addition, subtraction and multiplication.

The general strategy for fixing these problems is to use a different coordinate system, a.k.a. a change of variables. It turns out that *projective coordinates*, which we introduced in order to justify including the point at infinity on the curve, are also more efficient for point addition. The general idea is that you do one set of inversions to do the coordinate transform, then you do all the math in the new coordinates (which now does not require inversions), and then you do one more set of inversions at the end to transform the results back into regular Cartesian coordinates. The details are beyond the scope of this paper.

There are various ways to solve the side-channel attack problem. The most common is called a Montgomery ladder.