CS 2051: Honors Discrete Mathematics
Spring 2023 Homework 7 Supplement

Sarthak Mohanty

Overview

In recent weeks, we have delved into number theory and explored its practical application in RSA cryptography. However, there is another formidable cryptosystem called Elliptic Curve Cryptography (ECC) that is widely used today to secure various aspects of online communication, ranging from HTTPS connections to inter-data center transmissions. ECC is considered more robust than early public key cryptography systems like RSA, based on current mathematical knowledge. In this supplement, you will learn the algorithmic and mathematical background to understand this cryptosystem, following which you will implement a rudimentary version yourself![1]

Cryptography Primer

In this section, we discuss the difference between symmetric and asymmetric encryption, as well as what constitutes a "hard" problem in cryptography.

Symmetric vs Asymmetric Systems

In cryptography, there are two main types of encryption algorithms: symmetric and asymmetric.

Symmetric, or **private-key** encryption uses the same key to both encrypt and decrypt the message. Symmetric encryption algorithms are generally faster and more efficient, but they require the sender and the recipient to share a secret key. The most prominent example is the AES cipher.[2]

On the other hand, **asymmetric**, or **public-key encryption** algorithms are slower and less efficient, but they allow the sender to send a message securely to the recipient without sharing a secret key. Examples include RSA, Diffie-Hellman, and ECDH.

Trapdoor Functions

Trapdoor functions are an essential component of asymmetric encryption algorithms. A trapdoor function is a function that is easy to compute in one direction, but difficult to compute in the opposite direction without knowledge of additional information, such as a secret key.

As an example, the trapdoor function in RSA encryption is the *integer factorization problem*:

$$\text{Let } N = pq \text{ be a product of 2 primes with } \log(p) \approx \log(q). \text{ Given } N, \text{ extract } p, q.$$

This is a difficult problem to solve, especially for large values of p and q. However, if one has knowledge of the factors p and q, computing the value of N is straightforward. This property makes the integer factorization problem an excellent candidate for a trapdoor function.

[1] A good resource for the implementation of this supplement is [2].

[2] Some of your classmates have chosen this as their group project!

Part 1: Diffie-Hellman (10 pts)

Before we begin, consider the following story:

> Professor Brito and Lianet (Brito's wife!) have a famous Piña Colada recipe that they keep secret from the rest of the world. It is so secret in fact, that neither Brito nor Lianet know the full parts of the recipe! However, using their knowledge on how the Diffie-Hellman key exchange algorithm works, they have devised a unique way of making the drink.
>
> To make their famous Piña Colada, Brito and Lianet have their own secret ingredients, which they mix separately. Brito mixes a secret amount of rum to a glass, while Lianet mixes a secret amount of pineapple juice to another glass. These individual mixtures represent the private keys of each party:
>
> Brito: 🍸 + 🥥 + 🥃 = 🍸   Lianet: 🍸 + 🥥 + 🍍 = 🍸
>
> Next, they swap glasses. This represents the public key exchange. Note that because the ingredients were mixed in, it is very difficult for either Brito or Lianet to determine the exact contents of the current glass.
>
> Brito: 🍸   Lianet: 🍸
>
> Once they have exchanged their mixtures (or "public keys"), they again use their own secret ingredients (or "private keys") to mix with the others' drink to generate a shared "secret key", in this case the final drink.
>
> Brito: 🍸 + 🥃 = 🍹   Lianet: 🍸 + 🍍 = 🍹
>
> This shared secret key can be used to establish a secure communication channel, just like the delicious piña colada they created together.

Let's now describe any algorithm that can transmit any information, not just piña coladas.

Diffie-Hellman Key Exchange

The Diffie-Hellman (DH) key exchange is a cryptographic protocol that allows two parties to generate a shared secret key over an insecure communication channel. The protocol is named after its inventors, Whitfield Diffie and Martin Hellman, and is performed as follows:

1. Alice and Bob agree to use a prime p and a primitive root g of p.
2. Alice chooses a secret positive integer $a < p$ and computes her public key, given by $g^a \pmod{p}$. Bob also chooses a secret positive integer $b < p$ and computes his public key, given by $g^b \pmod{p}$.
3. Alice and Bob exchange their respective public keys over an (possibly) insecure channel.
4. Alice calculates $S = (g^b \pmod{p})^a$ (using her private key and Bob's public key) and Bob calculates $S = (g^a \pmod{p})^b$ (using his private key and Alice's public key). Note that they have now computed a shared secret, as
$$S = (g^a \pmod{p})^b = g^{ab} \pmod{p}.$$

Trapdoor Function: Discrete Log Problem

The security of DH comes from the fact that an attacker, even given g, p, $g^a \pmod{p}$, and $g^b \pmod{p}$, <u>cannot</u> efficiently calculate $g^{ab} \pmod{p}$. This leads us to a trapdoor function known as the **discrete log**

problem:

$$\text{Given } g,\ p, \text{ and } g^x \bmod p, \text{ determine } x.$$

We will first demonstrate that we can break the discrete log problem if the parameters x and p are too small. Then, we will see through our implementations that as we increase the size of our numbers, the power of our attacks gets weaker.

Brute Force

Recall that we want to find a given g, p, and $g^x \bmod p$. The initial approach one might have would be to try all possible values of x. This takes $\mathcal{O}(p)$ time, which follows from the fact that g is a generator.

Baby-Step Giant-Step algorithm [3]

Before we describe the algorithm, first note that any integer x can be written as $x = am + b$, where a, m, and b are arbitrary integers. For example, we can write $10 = 2 \cdot 3 + 4$. With this knowledge, we can rewrite the equation for the discrete log problem as follows:

$$\begin{aligned} g^x &\equiv y \pmod{p} \\ g^{am+b} &\equiv y \pmod{p} \\ g^b &\equiv y \cdot g^{-am} \pmod{p} \end{aligned}$$

The baby-step giant-step is a ”meet in the middle” algorithm. In essence, instead of computing $g^x \pmod{p}$ for all values x, we instead calculate a few values for $g^b \pmod{p}$, as well as calculate a few values for $g^{-am} \pmod{p}$. The algorithm works as follows:

Algorithm 1 BabyStepGiantStep

```
$m = \lceil \sqrt{p}\, \rceil$
for each $b$ in $0, \ldots, m$
    Store $g^b \pmod{p}$ in a certain data structure (which one?)
for each $a$ in $0, \ldots, m$
    Check to see if there exists some $g^b$ such that $y \cdot g^{-am} \equiv g^b \pmod{p}$ (in $\mathcal{O}(1)$ time!)
return $x = am + b$
```

To understand why this algorithm works, forget the modulo for a second and take the equation $y = g^{am+b}$. Consider what follows:

- When $a = 0$ we are checking whether y is equal to g^b, where b is one of the integers from 0 to m.
- When $a = 1$ we are checking whether y is equal to g^{m+b}. We are comparing y against all exponents from m to $2m$.
- When $a = 2$ we are comparing y against all exponents from $2m$ to $3m$.

  $\vdots$

- When $a = m - 1$, we are comparing y against all exponents from $(m - 1)m$ to $m^2 = p$.

In other words, **we are checking all exponents from** 0 **to** p. The time complexity of this algorithm is $\mathcal{O}(\sqrt{p})$.

[3] Adapted from [2].

General Number Field Sieve (GNFS)[4]

The General Number Field Sieve is a factorization algorithm that is particularly efficient for numbers with large prime factors. It is a multi-step process that involves finding a smooth polynomial and calculating its roots. GNFS is one of the most efficient known factorization algorithms for large integers.

We mention this attack not because of the implementation, but rather due to the implication. Algorithms such as GNFS get more efficient as the size of the numbers being factored get larger. As the resources available to decrypt numbers increase, the size of the keys need to grow even faster. This is not a sustainable situation for mobile and low-powered devices that have limited computational power. For this reason (and others), we need a better public key system.

In this part, you'll first implement the `Actor` class, which contains methods to emulate the Diffie-Hellman key exchange:

Actor

A given actor has access to the public parameters p and g, as well as its own private key k.

- `computePublicKey(self)`: Computes and returns the public key.
- `computeSecret(self, offer)`: Takes in a public key and assigns to the actor the corresponding shared secret.

You'll then try to break this key exchange by implementing the `BadActor` class, containing the following methods:

BadActor

The bad actor has access to the same public parameters as above, but not the private key.

- `brute(self, y)`: This function solves the discrete log problem using brute force.
- `bsgs(self, y)`: This function solves the discrete log problem using the baby-step giant-step algorithm.
- `stealSecret(self, actor1, actor2, attack)`: This method will use the attacks outlined above to intercept the Actors' shared secret.

These attacks should be reasonably efficient. For example, with parameters $p = 351416641$, $g = 19$ and $a = 137679007$, the following code took 10-20 seconds to brute-force a solution:

```
alice = Actor("Alice", p, g, a)
petty_thief = sol.BadActor("Petty Thief", p, g)
petty_thief.brute_force(alice.computePublicKey())
```

On the other hand, the baby-step giant-step algorithm below took less than a millisecond!

```
crime_boss = BadActor("Crime Boss", p, g)
crime_boss.bsgs(alice.computePublicKey())
```

Tip: Calculating g^a directly and then finding its remainder when divided by p is impractical and will not work with large numbers. This is because g^a will be a huge number for large g, a. Instead, you should use an algorithm that employs the binary expansion of the exponent a.[5]Luckily, Python provides an inbuilt `pow` method that performs this algorithm internally.

[4] Adapted from [3].

(Optional)[6] Part 2: Enter Elliptic Curves

In this section, we delve into the mathematics behind elliptic curves. The standard form for an elliptic curve is

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

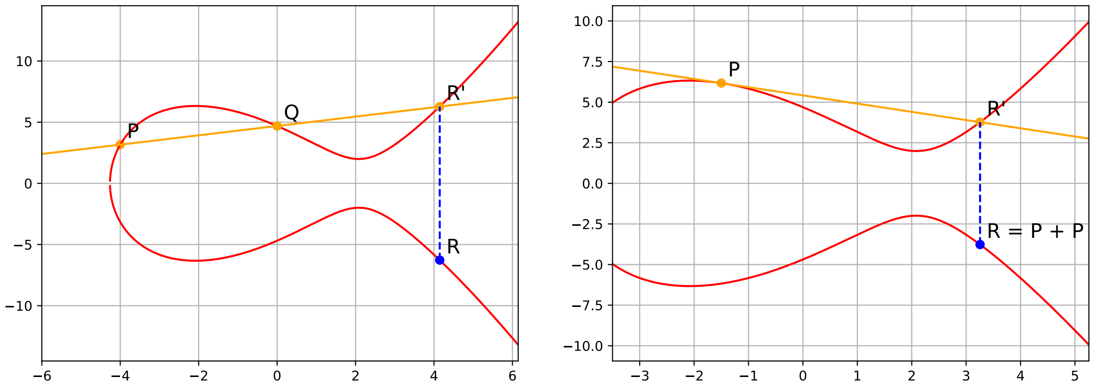
where a and b are constants. This is referred to as the Weierstrass equation for an elliptic curve. For reasons that will be discussed later, also include the point $\{\infty\}$ in this function.

The Group Law

One important aspect of elliptic curves is that we can start with two points, or even one point on an elliptic curve, and produce another point. To do this, start with two points $P = (x_1, y_1)$, $Q = (x_2, y_2)$ on an elliptic curve E. Now draw a line L through P and Q. This line will intersect E at a third point R'. (such a point always exists, as you'll prove below!) Finally, reflect R' across the x-axis to obtain R. We define

$$P + Q = R.$$

You can see an example of such addition in Figure (1).



(a) Point addition with $P \neq Q$

(b) Point addition with $P = Q$

Figure 1: Group law visualized.

[5] The details of this algorithm can be found in Section 4.2 of the textbook, under "Modular Expoonentiation".

[6] Not really optional, since Part 3 builds upon this section.

In this part, you'll implement the following functions:

- `point_addition(elliptic_curve, P, Q)`: This function takes in an elliptic curve of the form described as well as two points on the curve P and Q, and returns their "sum".
- `point_scalar_multiplication(elliptic_curve, k, P))`: This function takes in an elliptic curve of the form described, a scalar integer k, and a point P, and repeatedly multiplies this result to output a number.

I recommend using this interactive webpage to validate your results.

Hint: This part does not require any fanciful coding; just a bit of algebra and calculus. Since I know it's been a while since many have touched those concepts, here's a few steps to get started.

- Suppose we are adding two points $P = (x_1, y_1)$ and $Q = (x_2, y_2)$ such that $P \neq Q$. Consider the line L drawn through P_1 and P_2. Its slope is

  $$m = \frac{y_2 - y_1}{x_2 - x_1}.$$

  If $x_1 = x_2$, then L is vertical, so the resulting point is just ∞. Now let's assume that $x_1 \neq x_2$. The equation of L is then

  $$y = m(x - x_1) + y_1.$$

  To find the intersection of this line with the elliptic curve, we can substitute the above equation into the formula for an elliptic curve

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

  Finally, we can reflect the above point across the x-axis to obtain the point $P_3 = (x_3, y_3)$.

- Now suppose we are adding $P = Q = (x_1, y_1)$. When two points on a curve are very close to each other, the line through them approximates a tangent line. Therefore, when the two points coincide, we take the line L through them to be the tangent tine. Implicit differentiation allows us to find the slope m of L:

  $$2y\frac{dy}{dx} = 3x^2 + a, \quad \text{so} \quad m = \frac{dy}{dx} = \frac{3x_1^2 + a}{2y_1}.$$

  If $y_1 = 0$ then the line is vertical and we set $P_1 + P_2 = \infty$, as before. Otherwise, we plug in values into L and solve as before. To make this algorithm efficient, you may want to look up the "square-and-multiply" method.

Part 3: In The Field (12 pts)

The simplified curve above is great for visualization and to explain the general concept, but it doesn't really represent how the curves used for cryptography look like. Just like in RSA, we have to restrict ourselves to a fixed range of numbers.

The elliptic curve cryptography (ECC) uses elliptic curves taken over the finite field $\mathbb{F}_p$, where p is a prime number. For example, the "Bitcoin curve" secp256k1 takes the form:

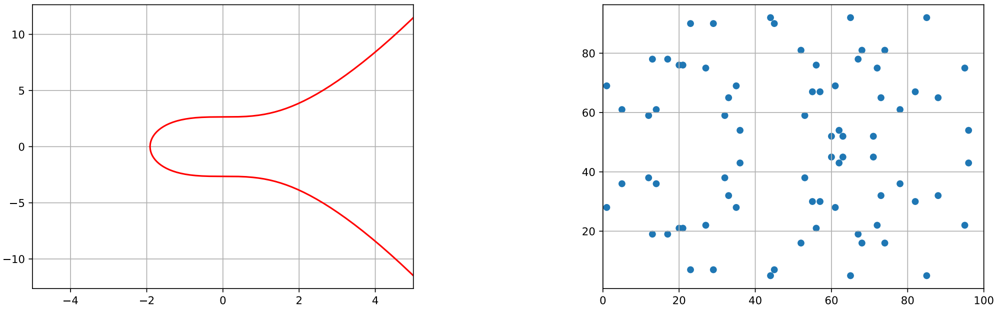
$$y^2 \equiv x^3 + 7 \pmod{p}.$$

Figure 2 shows an illustration of the curve taken over $\mathbb{R}$ and $\mathbb{F}_{97}$.

The key difference in point multiplication over a prime field lies in the implementation of the arithmetic operations used in the group law - namely, addition and doubling. In ECC over a prime field, these operations are performed modulo p. This means the usual line intersections for point addition and tangents for point doubling that we are familiar with in real number field don't translate directly.

As an example, consider the division operator used in the slope calculation for point addition. In the field of real numbers, we can directly divide two numbers. However, in a finite field, this isn't possible as standard division could lead to non-integer results. So instead, we must take the **multiplicative inverse**.

For a deeper understanding of the alterations made to these operations, as well as a thorough exploration of their implementation, one can refer to [2].



(a) The bitcoin curve over $\mathbb{R}$

(b) The bitcoin curve over $\mathbb{F}_{97}$.

Figure 2: Visualizing elliptic curves over a field

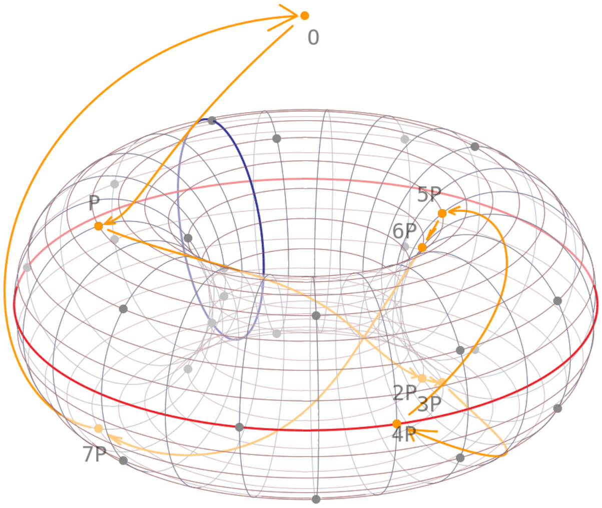


Figure 3: Visualizing point multiplication over a field. (Source: [4])

In this part, you'll visualize elliptic curves over finite fields, as well as implement point addition and point multiplication over these fields

In this part, you'll implement the following functions:

- `generate_point_cloud(a, b, p)`: This takes in an elliptic curve and returns the set of all valid points in the curve over that field, including the point at infinity.
- `point_addition(P, Q, a, b, p)`: This function takes in an elliptic curve (now over a finite field) and two points. It returns their point-wise sum.
- `point_scalar_multiplication(P, k, a, b, p)`: This function takes in an elliptic curve of the form above, a scalar integer k, and a point P, and repeatedly multiplies P k times.

After completing each function, you should again be able to check your work with the interactive tool mentioned above.

Part 4: ECC in Action: Elliptic-curve Diffie Hellman (10 pts)

The Elliptic-curve Discrete Log Function

Finally, the part you've been waiting for. With our newfound knowledge about elliptic curves, we can make the DH key exchange much more powerful, as follows:

1. Alice and Bob agree to use a curve $C = (a, b, p)$ and a generator point P on the curve.
2. Alice chooses a secret positive integer $k_a < p$ and computes her public key, given by k_aP. Bob also chooses a secret positive integer $k_b < p$ and computes his public key, given by k_bP.
3. Alice and Bob exchange their respective public keys over an (possibly) insecure channel.
4. Alice calculates $S = k_a(k_bP)$ (using her private key and Bob's public key) and Bob calculates $S = k_b(k_aP)$ (using his private key and Alice's public key). Note that they have now computed a shared secret, as
$$S = k_a(k_bP) = k_b(k_aP).$$[7]

The Elliptic-curve Discrete Log Function

The security of this cryptosystem follows from yet another trapdoor function, let's call it the **elliptic-curve discrete log problem**:

Given a curve, Q, and P, determine k such that $Q = kP$.

The inherent complexity stems from the nature of scalar multiplication in elliptic curves. When a point P on an elliptic curve is added to itself k times, resulting in a point Q, the process is straightforward. However, if points Q and P are given, finding the integer k that was used in the scalar multiplication operation is a different matter. As of now, there is no known efficient method for computing it.

Note the similarity between the elliptic-curve discrete log problem and the original discrete log problem. In fact, the attacks used to breaj them (such as baby-step giant-step) share similarities as well. More information can be found in [2].

[7] The proof of this property can be found in [1]

In this part, you'll implement an Elliptic-Curve Diffie Hellman key change algorithm. The functions you will implement are similar to those in Part 1. First, you'll implement the `Actor` class:

Actor

A given actor has access to the public curve parameters (a, b, n), the public generator point P, and a private key k.

- `computePublicKey(self)`: Computes and returns the public key.
- `computeSecret(self, offer)`: Takes in a public key and assigns to the actor the corresponding shared secret.

Then, the `BadActor` class.

BadActor

The bad actor has access to the same public parameters as above, but not the private key.

- `brute(self, Q)`: This function solves the ECDLP using brute force.
- `bsgs(self, Q)`: This function solves the ECDLP using the baby-step giant-step algorithm.
- `stealSecret(self, actor1, actor2, attack)`: This method will use the attacks outlined above to intercept the Actors' shared secret.

Again, to receive full credit, these attacks should be reasonably efficient. You may also want to double-check the efficiency of your methods developed in Part 3, since they should be used in this section.

Conclusion

RSA vs ECC

Since RSA and ECC essentially serve the same purpose, there have been many heated discussions as to which implementation to use. The main pro of using ECC is the fact that it uses fewer memory and CPU resources, important as mobile computing becomes more ubiquitous. However, there are still quite a few flaws with ECC, as discussed in this article.

Additionally, neither ECC nor RSA are secure against quantum computers. Even if quantum computers are not ready to be deployed at scale yet, lots of information is being stored now with intentions of being decrypted later. This has spurred calls to create quantum-resistant encyption schemes.

Real World Applications

Even with all those downsides, more and more companies are using ECC, including

Amazon, Bing, Dropbox, Facebook, Flickr, GitHub, Instagram,
LinkedIn, MSN, Netflix, Pinterest, PirateBay, Quora, Snapchat,
SoundCloud, Spotify, StackOverflow, Tumblr, Twitter, Uber,
Vimeo, Vine, Yahoo, Yelp, YouTube, Wikipedia, Wordpress,. . .

Even ChatGPT is using ECC to establish their connections, which you can check yourself. In Chrome, clicking on the lock in the address bar and traversing to the connection tab shows which cryptographic protocols were used in establishing the secure connection. ECC should be listed as one of them.

However, as noted in the beginning of this supplement, symmetric encryption schemes are much faster than their asymmetric counterparts. This is why in practice, most information is transferred using symmetric encryption schemes such as AES. This does not mean asymmetric schemes are useless; in practice, both encryption schemes are usually used in tandem. You can see an example of such "hybrid" symmetric-asymmetric schemes here.

Submission Instructions (10 pts)

After you fill the appropriate functions, submit the following files to Gradescope and make sure you pass all test cases:

- `diffie_hellman.py`
- `elliptic_curves.py`
- `ECDH.py`

Notes

- The autograder may not reflect your final grade on the assignment. We reserve the right to run additional tests during grading.
- Do not import additional packages, as your submission may not pass the test cases or manual review.

References

[1] Washington, L. C. (2008). *Elliptic curves: number theory and cryptography (2nd ed.).* Chapman and Hall/CRC.

[2] Corbellini, A. (2015). *Elliptic Curve Cryptography: a Gentle Introduction.* https://andrea.corbellini.name/2015/05/17/elliptic-curve-cryptography-a-gentle-introduction/

[3] Sullivan, N. (2023, October 23). *A (Relatively Easy To Understand) Primer on Elliptic Curve Cryptography.* The Cloudflare Blog. https://blog.cloudflare.com/a-relatively-easy-to-understand-primer-on-elliptic-curve-cryptography/.

[4] Pantůček, D. J. (2018, April 26). *Elliptic curves: prime-order curves.* Trustica. https://trustica.cz/2018/04/26/elliptic-curves-prime-order-curves/