CS 2051: Honors Discrete Mathematics
Spring 2023 Homework 7 Supplement outline

Sarthak Mohanty, Anant Gupta

Title: ECC is the new RSA

Overview

Over the past few weeks, we've covered number theory, as well as a nice application of it through the use of RSA cryptography.

Elliptic Curve Cryptography (ECC) is one of the most powerful cryptosystems in use today. Companies are using ECC everything to secure everything from our customers' HTTPS connections to how we pass data between our data centers. In fact, based on currently understood mathematics, ECC provides a significantly more secure foundation than first generation public key cryptography systems like RSA, for reasons we'll explore in detail later. The specific implementations you might learn in this supplement aren't as important as the general framework, understanding, and ideas you'll develop.

Note. As I was making this assignment, I stumbled across this excellent blog post, and realized it covers some of these concepts a lot better than I ever could. For this reason, some of the exposition is adapted from this resource.

Cryptography Primer

Before we continue, it is beneficial to take a step back, and discuss cryptography as a whole. In this section, we discuss the difference between symmetric and asymmetric encryption, as well as what constitutes a "hard" problem in cryptography. If you are already familiar, feel free to skip ahead.

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.[1]

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 ECCDH.

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.

[1] some of your classmates have chosen this as their group project!

As an example, the trapdoor function in RSA encryption is the modular exponentiation function, $c = m^e \mod N$, where m is the plaintext, e is the public exponent, N is the modulus and c is the ciphertext. The trapdoor information is the private key, which consists of the factors of N. In the opposite direction, decryption is performed by computing $m = c^d \mod N$, where d is the private exponent derived from the private key.

Part 1: Diffie-Hellman (10 pts)

Diffie-Hellman Key Exchange

There is a classic analogy for the DH key exchange, which we present below:

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.

In this analogy, the rum and pineapple juice represent the private keys, while the remaining ingredients represent the public key exchange. The shared secret key is the final product, which can be used for secure communication between the two parties.

Let's now describe any algorithm that can transmit any information, not just piña coladas. The **Diffie-Hellman 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:[2]

1 Alice and Bob agree to use a prime p and a primitive root g of p.

[2] taken verbatim from the CS 2051 Textbook.

2 Alice chooses a secret integer k_1 and sends $g^{k_1} \pmod p$ to Bob.

3 Bob chooses a secret integer k_2 and sends $g^{k_2} \pmod p$ to Alice.

4 Alice computes $(g^{k_2})^{k_1} \pmod p$

5 Bob computes $(g^{k_1})^{k_2)} \pmod p$.

Trapdoor Function: Discrete Log Problem

The security of DH comes from the fact that an attacker, even given g, p, $g^{k_1} \pmod p$, and $g^{k_2} \pmod p$, cannot efficiently calculate $g^{k_1 k_2} \pmod p$. This leads us to a trapdoor function known as the *discrete log problem*:

Given g, p, and $g^x \bmod p$, 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. This cool fact is hopefully something you will see in your implementations.

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 $g^{am} \pmod p$, 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 in a certain data structure (which one?)
for each a in $0, \ldots, m$
  Calculate $y \cdot g^{-am}$.
  Check to see if there exists some g^b such that $y \cdot g^{-am} = g^b$ (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 $2m$.

[3] this section was adapted from the blog post linked at the beginning of this supplement.

- When $a = 2$ we are comparing y against all exponents from $2m$ to $3m$.
- ...
- 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 $\mathcal{O}(\sqrt{p})$.

General Number Field Sieve (GNFS)

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 then finding the roots of that polynomial. 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. The gap between the difficulty of factoring large numbers and multiplying large numbers is shrinking as the number (i.e. the key's bit length) gets 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. The gap between factoring and multiplying is not sustainable in the long term.

All this means is that RSA is not the ideal system for the future of cryptography. In an ideal Trapdoor Function, the easy way and the hard way get harder at the same rate with respect to the size of the numbers in question. For this reason (and a few others discussed later), we need a better public key system. In 1985, such a system was found, revolving around an (then) arcane field known as elliptic curves.

In this part, you'll 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.

To-do: Rewrite this, talk about built in pow method
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. The details of this algorithm can be found in Section 4.2 of the textbook, under "Modular Expoonentiation".

Part 2: Enter Elliptic Curves (8 pts)

The standard form for an elliptic curve is

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

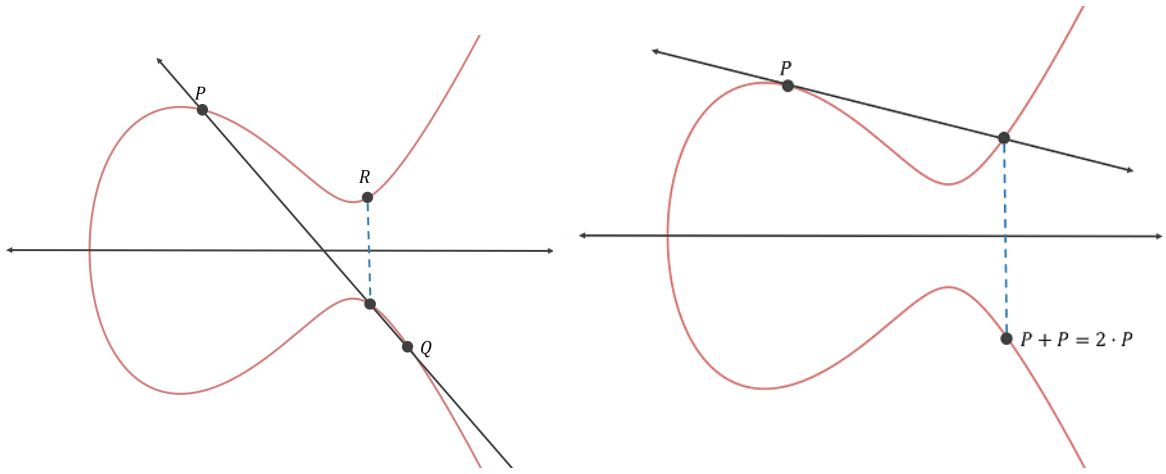
where a and b are constants. This is referred to as the Weierstrass equation for an elliptic curve. For reasons that will be discussed later, also include the point $\{\infty\}$ in this function.

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, and you'll prove it when trying to calculate the exact value of this point below!) Finally, reflect R' across the x-axis to obtain R. We define

$$P + Q = R.$$

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



(a) Point Addition (b) Point scalar multiplication

Figure 1: Group law visualized.

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

- `point_addition(elliptic_curve, P, Q)`: This function takes in an elliptic curve
- `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.

You can run `visualize_addition` and `visualize_multiplication` to generate the corresponding visualizations mentioned above. If you want more examples, there's an interactive tool here Notes: This part is just algebra and geometry. Since I know it's been a while since many of you touched this, here's a few steps you get you started.

- For point addition, we assumed that $P \neq Q \neq \infty$. For this reason, we can draw the line L 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 $P_3 = (x_3, y_3)$.
- For point scalar multiplication, we have that $P = Q = (x_1, y_1)$. 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.

Part 3: In The Field (12 pts)

The elliptic curve cryptography (ECC) uses elliptic curves taken over the finite field $\mathbb{F}_p$ (where p is prime and $p > 3$). For example, the "Bitcoin curve" secp256k1 takes the form:

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

Firstly, what do these curves look like? Well, they actually look pretty random. For example, the bitcoin curve mapped over mod 3 looks like

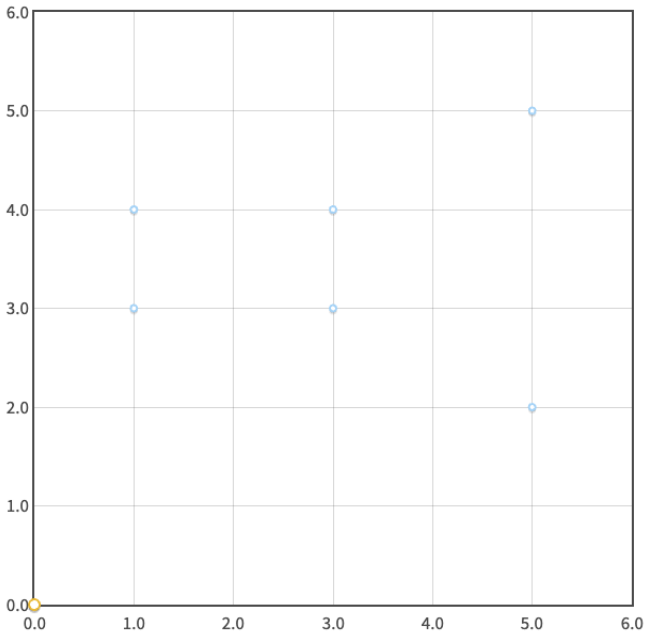


Figure 2: An elliptic curve over a Galois field.

The group law over $\mathbb{F}_p$ is similar, but not identical to in the field of real numbers. It is stated (and proven) in this research paper.

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 a set of all valid points in the curve over that field.
- `point_addition(P, Q, a, b, p)`: This function takes in an elliptic curve and two points, and adds them.
- `point_scalar_multiplication(P, n, a, b, p)`: This function takes in an elliptic curve of the form described, a scalar integer k, and a point P.

After completing each function, you should be able to run `visualize_curve`, `visualize_point_addition`, and `visualize_point_scalar_multiplication` to check your work.

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

The Elliptic-curve Discrete Log Function

Trapdoor function in Elliptic Curve Cryptography (ECC). In ECC, the trapdoor function is the scalar multiplication operation $Q = kP$, where P is a point on the elliptic curve, k is the scalar and Q is the result of the scalar multiplication. The trapdoor information is the private key, which is the scalar k. Decryption is performed by computing $Q = -kP$.
For carefully chosen (by cryptographers) finite fields and elliptic curves, the ECDLP problem has no efficient solution.
The multiplication of elliptic curve points in the group $\mathbb{F}_p$ is similar to exponentiation of integers in the group $\mathbb{Z}_p$ (this is known as multiplicative notation) and this is how the ECDLP problem is similar to the DLP problem (discrete logarithm problem).
See this video for implementation details

Finally, the part you've been waiting for. 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.

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, 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.

Conclusion

RSA vs ECC

Since RSA and ECC essentially serve the same purpose, there have been many heated discussions as to which implentation 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 a few flaws with ECC. First, there are only a few curves that work and the NIST is in contorl of most of them, but people distrust NIST. Second,

Finally, neither ECC nor RSA are secure against quantum computers.

Real World Application

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,...

Two prominent examples:

- ChatGPT
- ProtonVPN

to do: explain how you can see ecc implentation on website. Click on lock on top left of website, click on "connection is secure", click on "certificate is valid", you should see ECC used for certificate
assymetric not often used since takes long time. Instead, key exchange is done using assymetric andthen symmetric is used to send the actual messages, as can be seen 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

- explanation for group law name is here
- why infinity is added here
- This (heavy) textbook is a good resource if you want to explore the mathematics of elliptic curves in further detail.
- Again, the amazing blog post mentioned above.