COMP-4541 – Final-Project

Farshid Adavi - 4021305965007

# 1 Voting Protocol with Blind Signature

## Manager

1. **Management Introduction:** This project explores the implementation of a secure voting protocol using a blind signature process. A contract manager oversees the voting process, ensuring its integrity and preventing fraud. The administrator signs purchased ballots using an RSA algorithm with a private key, while the corresponding public key is publicly disclosed.

2. **Motivation for Management Responsibility:** The responsibility of managing the voting process is significant, but it comes with financial compensation. The rationale for accepting this responsibility lies in the reliability of the manager's real-world identity. Moreover, if fraud is detected, the election will be rerun, motivating the manager to uphold honesty.

3. **Reasons for Manager Integrity:** The manager is deterred from cheating by two key factors. Firstly, fraud is easily recognizable, and individuals can object to it. In such cases, the manager's deposit will be deducted, and the amount will be refunded to affected parties. Additionally, the first person to report fraud will be rewarded. Secondly, the manager cannot predict the outcome of the election until she have the opportunity to cheat, thus discouraging manipulation.

4. **Accountability Measures:** There is also an address of a recognized court where someone reports fraud to the administrator and the election is cancelled. Everyone except the manager and the complainant take their money, but the money of the manager and the complainant can only be taken by the court.

## Protocol Overview

1. **Registration Phase:** Voters register as many times as they wish within a specified period, paying a fee and submitting their preferred ballot format to the contract.

2. **Ballot Generation:** Each ballot is generated by combining the round's series, a 256-bit random nonce, and the ID of the voter's chosen candidate. This combination is hashed and multiplied with $r^e$.(r: a big random integer e: a public key that generate by manager in RSA cryptography).

3. **Signature Verification Phase:** The manager signs all the ballots during the second phase, and voters have an opportunity to verify the signatures for validity during the third phase.

4. **Third phase:** During this phase, voters are granted the opportunity to scrutinize the signatures on their ballots to verify their validity. Additionally, it is imperative that all ballots are signed within this timeframe. Failure to do so empowers the voter to invoke a function and halt the process. It is important to note that there are no advantages to initiating baseless legal action. Firstly, if the plaintiff fails to substantiate their claim of managerial fraud in court, they will be liable for associated costs. Furthermore, until the window for protest remains open, no definitive conclusions can be drawn about the outcome of the election.

5. **Commitment Phase:** In the fourth phase, voters divide their signed ballots by the $r$ and commit the result using an anonymous address.

6. **Signature Validation Phase:** The administrator then verifies the signatures of the committed ballots during the fifth phase and sets validity flags.

7. **Protest Phase:** Voters can protest against incorrect flags or unflagged ballots during the sixth phase, initiating a reevaluation of the administrator's performance.

8. **Choice Disclosure Phase:** In the seventh phase, voters reveal their choices. The pickwinner function is then called to announce the election result.

9. **Withdrawal Phase:** After this, voters can withdraw a portion of their registration fee for each purchased ballot.

## Wrong attempt to implement checksign with ECC

I also tried to implement checksign, which is compatible with the blind signature process, to eliminate the phases of checking the signatures of the ballots that were done by the voters. In such a way also the role of the court will been removed and the manager can't cheat at all. And if she fail to sign the ballots, she will be fined automatically. Also consume less storage lead to consume less gas.
I first realized that these existing signatures and signature check functions are based on ECC and have nothing to do with RSA. In line with the implementation of the function, I noticed that ECC is much faster and consumes less gas. I decided to study it to implement blind signature with it.
I spent a few days studying ECC (the report is at the end of this PDF file) and finally I realized that the blind signature process in EEC is very expensive and difficult.
But I also lost an opportunity to implement this function with RSA. I hope I can implement them in the future.

## 1.1 Variables

– `election_manager_address`: Ethereum address of the election manager, responsible for signing and verifying ballots.

- `election_number`: Number assigned to each election to prevent replay attacks.
- `initial_block_number`: Block number when the contract was deployed.
- `total_ballots`: Total number of registered ballots.
- `total_signed_ballots`: Total number of signed ballots.
- `all_Signatures_are_ready`: Boolean flag indicating if all ballots have been signed.
- `register_is_busyr`: Flag to prevent re-entry during the registration process.
- `voters`: Mapping of voter addresses to their blurred ballots and signed blurred ballots.
- `Votes_of_candidates`: Mapping of candidate IDs to the number of votes received.
- `first_candida`: ID of the candidate with the highest number of votes.
- `isEnoughToPickWinner`: Indicates if there are enough moves to pick a winner.
- `winnerPicked`: Boolean flag indicating whether the winner has been determined.
- `end_of_ellection`: Boolean flag indicating if the election has ended.
- `withdraw_is_busy`: Flag to prevent re-entry during the withdrawal process.

## 1.2 Functions

1. `constructor(uint16 election_number_) payable`
   50K gas

   - **Purpose:** Initializes the Voting smart contract by setting up critical parameters needed for its operation, including security features to prevent replay attacks and fraud, and ensures the election manager is accountable by requiring a significant Ether deposit.
   - **Parameters:**
     - `election_number_` – A unique identifier for the election to prevent replay attacks using previously signed ballots from other elections.
   - **Usage:** This function is automatically called at the time of contract deployment to the Ethereum blockchain. It must be called with a value of 35 Ether, which serves as a security deposit and operational fund for managing the election.
   - **Effects:**
     - Sets `election_manager_address` to the address deploying the contract, which should ideally be hardcoded in a production environment.

- Records the block number at deployment to `initial_block_number` for future reference in time-sensitive operations.
- Establishes `election_number` with the given parameter to uniquely identify the current election and secure it against replay attacks.
- Requires an exact deposit of 35 Ether to cover the maximum potential gas costs for all ballot registrations and provide a 2 Ether bounty for identifying fraud or errors, ensuring the election manager has a financial stake in the integrity of the election process.

- **Security Considerations:**
  - **Accountability of Election Manager:** By requiring the election manager to deposit 35 Ether, the contract ensures there is a significant financial deterrent against fraudulent behavior and a reserve for compensating voters if the election's integrity is compromised.
  - **Replay Attack Prevention:** Utilizing a unique election number for each event helps prevent the misuse of ballots from previous elections, securing the current election from such vulnerabilities.
  - **Financial Guarantees:** The inclusion of a 2 Ether reward for detecting errors or fraud incentivizes vigilance among participants, further safeguarding the election process.
- **Note:** This constructor function is critical for setting up the election environment and should always remain part of the contract. It is not meant for modification or removal after deployment, except for the testing addresses which should be hardcoded prior to production deployment to enhance security.

2. `register(bytes32 blurred_ballot) public payable`
   first time call for each address 57K and then 40K gas

- **Purpose:** Allows voters to register their blurred ballots for the election, ensuring that the process adheres to the established timeframe and securing the integrity of each vote through cryptographic means.
- **Parameters:**
  - `blurred_ballot` – The hashed or encoded representation of a voter's ballot, which maintains the confidentiality of the voter's choice until the signature process.
- **Usage:** This function is called by voters wishing to participate in the election. Each call to this function requires an exact payment of 1 Ether as a registration fee and must be made within a specified timeframe relative to the deployment of the contract.
- **Effects:**
  - Checks if is not currently busy to prevent double entries.
  - Sets the 'register_is_busy' flag to true to lock the function against re-entry while it is executing.
  - Validates that the registration is within the allowed timeframe (10 days from the start, defined as 5 times the variable 't').
  - Confirms the receipt of exactly 1 Ether for the registration fee.
  - Ensures that the total number of ballots does not exceed the predefined limit (65535, which is $2^{16} - 1$).

- Registers the ‘blurred_ballot‘, marking it as awaiting a signature from the election manager, and increments the voter’s weight by 1.
- Increases the total number of ballots registered.
- Resets the ‘register_is_busy‘ flag to false once registration is complete.

- **Security Considerations:**
  - **Preventing Overflow and Re-Entrancy:** The function ensures that the maximum ballot count is not exceeded and that the registration process cannot be entered again while it is already running, thereby maintaining orderly and secure registration.
  - **Blurred Ballots and Blind Signatures:** The use of blinded ballots and ECC (Elliptic-curve cryptography) for signatures ensures that voting remains confidential and secure, preventing potential cryptographic attacks that could compromise RSA in a blockchain context.
- **Note:** This registration function is crucial for maintaining a fair and orderly voting process. It should be carefully monitored and tested to ensure that it handles edge cases and high traffic scenarios effectively.

3. `Signing(address voter_address, bytes32 blurred_ballot, bytes memory signed_blurred_ballot) public` 104k gas

- **Purpose:** Enables the election manager to sign the blurred ballots of registered voters, thus validating them for the upcoming stages of the election. This process ensures that each ballot is authenticated while maintaining voter confidentiality.
- **Parameters:**
  - `voter_address` – The address of the voter whose ballot is being signed.
  - `blurred_ballot` – The identifier of the voter’s blurred ballot that needs to be signed.
  - `signed_blurred_ballot` – The cryptographic signature applied to the blurred ballot, performed by the election manager.
- **Usage:** This function is called by the election manager after the registration phase has concluded but within a specific timeframe designed to prevent delays in the election process. It is critical for the progression to the voting phase.
- **Effects:**
  - Verifies that the function caller is the election manager, ensuring that no unauthorized person can sign ballots.
  - Checks that the function is called after the registration period has ended and within the designated signing period, ensuring adherence to the timeline.
  - Validates that the blurred ballot awaiting signature exists and is assigned to the specified voter, preventing errors or fraudulent attempts to sign non-existent or incorrect ballots.
  - Assigns the signed version of the blurred ballot back to the voter’s record, increasing the count of total signed ballots.

- Verifies if all registered ballots have been signed, and if so, sets the flag indicating that the signing phase is complete and the election can proceed to the next stage.

- **Security Considerations:**
  - **Authorization Check:** By restricting the ability to sign ballots to the election manager, the function safeguards the integrity of the election process.
  - **Temporal Constraints:** Enforcing function execution within specific block number ranges ensures that the signing process adheres to the planned schedule, reducing the risk of tampering or delays.
  - **Integrity Checks:** The requirement for ballots to be pre-registered and awaiting signature before they can be signed prevents unauthorized or duplicated ballots from being processed.
- **Note:** While currently, the signature verification functionality is not active and needs further implementation, it is crucial for ensuring the authenticity of signed ballots. This function's proper operation is essential for maintaining the election's legitimacy and should be rigorously tested under various scenarios to ensure its robustness and security.

4. `stop_election() public`
   10k gas

- **Purpose:** Provides a mechanism to halt the election process in the event that not all ballots are signed within the designated timeframe. This function serves as a safeguard to ensure the integrity of the election is maintained in case of incomplete or faulty signing phases.
- **Usage:** This function can be called by any participant or observer after the designated signing period has concluded, provided the conditions for stopping the election are met.
- **Effects:**
  - Checks if the current block number has surpassed the allowed time for signing (initial block number plus 7 times the variable `t`, representing the end of the signing window).
  - Verifies that not all ballots have been signed and that the election has not already been stopped ('should_be_false' is still false).
  - Sets the 'should_be_false' flag to true, effectively stopping the election process and preventing any further election-related actions from being processed.
- **Security Considerations:**
  - **Timing Restrictions:** The function enforces strict timing checks to ensure that the election can only be stopped after the signing phase has officially ended, preventing premature termination of the process.
  - **Conditional Execution:** The requirement that not all ballots have been signed ensures that the function is only used in scenarios where the signing process has failed to complete properly, thereby safeguarding the election from being halted without just cause.

- **Note:** The ability to stop the election is a critical fail-safe feature that ensures the election does not proceed under conditions where the integrity of the vote could be compromised. This function should be tested extensively to confirm that it activates under the correct conditions and that it properly halts all further election activities.

5. `commit(bytes32 hash_ballot, bytes memory signed_hash_ballot) public`
   53k gas

   - **Purpose:** Facilitates the secure submission of voter commitments after the signing phase, ensuring that each vote is both confirmed and cryptographically verified before the actual vote revelation.
   - **Usage:** This function is used by voters to commit their encrypted or hashed ballot along with a signature obtained in the signing phase. It marks the transition from ballot preparation to the final voting stage.
   - **Effects:**
     - Ensures the election has not been stopped due to errors or incomplete signings, as indicated by the 'should_be_false' flag.
     - Confirms that all ballots have been duly signed as required for the commit phase to proceed.
     - Checks the current blockchain time to ensure the commit is being made within the designated time frame after the signing phase and before the end of the commit window.
     - Validates that the voter has not previously committed their ballot to prevent duplicate entries.
     - Records the hashed ballot and flags the voter's ballot as committed, setting the stage for subsequent vote revelation.
   - **Security Considerations:**
     - **Election Integrity Checks:** By verifying that the election has not been prematurely stopped and that all ballots are signed, the function upholds the integrity and sequence of the election process.
     - **Temporal Validation:** Ensuring that commits occur within a specific timeframe prevents late submissions that could disrupt the election timeline and outcomes.
     - **Non-repudiation and Duplicate Prevention:** The function prevents voters from altering their vote after committing and ensures that each voter can only commit once, reinforcing the transparency and fairness of the election.
   - **Note:** The commitment phase is crucial for maintaining the confidentiality and integrity of votes until the final reveal. This function should be rigorously tested to handle all edge cases and ensure that it operates seamlessly under high network traffic and various blockchain conditions.

6. `function reveal(uint16 k_, uint256 _nonce) public`
   49K gas

   - **Purpose:** Allows voters to reveal their previously committed votes, ensuring that these votes are counted and validated against their initial commitments. This function is critical for maintaining the transparency and integrity of the election process.

- **Usage:** This function is called by voters after the commit phase to disclose the details of their vote, allowing their ballots to be counted towards the final election results.
- **Effects:**
  - Checks that the reveal phase is being conducted within the designated time frame, after the commit phase has concluded and before the reveal window closes.
  - Validates that the voter has committed a ballot and has not previously revealed it, preventing double-voting or alterations after the commit phase.
  - Confirms that the revealed vote matches the committed hash, ensuring the integrity of the vote through cryptographic verification. This is done by reconstructing the hash from the provided election number, candidate ID (k_), and nonce, and comparing it to the stored hash.
  - Marks the ballot as revealed to prevent further changes or duplicate reveals.
  - Increments the vote count for the revealed candidate. If the candidate receives more votes than the current leading candidate, updates the election's leading candidate.
- **Security Considerations:**
  - **Time-bound Operations:** The function enforces strict timing controls to ensure that reveals are only made within the correct phase of the election, adding a layer of security and timing integrity to the process.
  - **Non-repudiation:** By requiring the hash of the revealed vote to match the previously committed hash, the function ensures that voters cannot change their votes after committing them, thus maintaining the authenticity of each vote.
  - **Vote Integrity Verification:** The use of cryptographic hashing for vote verification ensures that the election cannot be tampered with, as altering the vote would require changing the nonce or candidate ID, which would result in a different hash.
- **Note:** The reveal function is a vital component of the voting process, providing the necessary mechanism for vote verification and finalization. It should be tested thoroughly to ensure that it handles all possible scenarios effectively, maintaining the security and reliability of the election process.

7. `pickWinner() public`
   17K gas

- **Purpose:** Finalizes the election process by determining and announcing the candidate with the highest number of valid votes, thus concluding the election and setting the stage for post-election procedures.
- **Usage:** This function is used to officially conclude the election, confirm the winning candidate, and ensure that the results are properly recorded and recognized.
- **Effects:**
  - Validates that the election has not been halted due to any detected irregularities or failures (checked by 'should_be_false').
  - Ensures that the winner has not already been picked to prevent multiple executions and potential discrepancies in election results.

- Checks that the current blockchain time is after the designated reveal phase, ensuring all votes have been accounted for before deciding the winner.
- Marks the election as completed by setting 'winnerPicked' and 'end_of_ellection' flags to true, thereby locking down the election for any further modifications or actions.
- Emits an event with the winner's ID, publicly announcing the result and providing a transparent record of the outcome.

- **Security Considerations:**
  - **Prevention of Premature Execution:** The function includes checks to ensure that it cannot be executed before all votes are revealed, thus maintaining the integrity of the election results.
  - **Single Execution Enforcement:** By verifying that the winner has not been previously picked, the function ensures the election's conclusion is a one-time, irreversible action.
  - **Event Transparency:** The emission of a winner event helps provide transparency, allowing anyone to verify the conclusion of the election and the identity of the winning candidate.
- **Note:** The 'pickWinner' function is a critical element of the election process, encapsulating the transition from voting to resolution. It should be rigorously tested to handle all edge cases and ensure that the transition is seamless, secure, and reflective of the actual votes cast.

8. `withdraw() public`
   40k gas

- **Purpose:** Facilitates the secure distribution of funds to participants after the conclusion of the election, ensuring that voters and the election manager are compensated according to their involvement and the results of the election.
- **Usage:** This function is called by voters and the election manager to withdraw their respective shares of the funds based on the outcome of the election or in case the election was stopped due to any discrepancies.
- **Effects:**
  - Ensures that the function is not currently busy to prevent re-entry issues which could lead to errors in fund distribution.
  - Validates that the election has ended or has been stopped, ensuring withdrawals are only processed post-election.
  - If the election ended successfully:
    * Voters receive 0.9 Ether for each vote weight they hold if they are not the election manager.
    * The election manager receives 0.1 Ether for each total ballot plus the initial deposit if they are the claimant.
    * Resets the voter's weight or total ballots to zero after withdrawal to prevent double spending.
  - If the election was stopped:
    * Voters receive their deposited funds back along with compensation for their transaction costs.

        * Adjusts the manager's compensation based on the remaining ballots and subtracts transaction costs.
    - Verifies successful transfer of funds, ensuring that all transactions complete without error.
- **Security Considerations:**
    - **Concurrency Control:** By setting the 'withdraw_is_busy' flag during operation, the function prevents concurrent access that could lead to inconsistencies or vulnerabilities in fund distribution.
    - **Conditional Checks:** The function includes multiple checks to ensure that withdrawals are only made under appropriate conditions, protecting against unauthorized or premature fund access.
    - **Fund Transfer Validation:** Ensures that each fund transfer is successful, adding a layer of reliability and security to the withdrawal process.
- **Note:** The withdrawal process is crucial for closing the financial aspects of the election and must be handled with utmost integrity and accuracy. This function should be thoroughly tested to ensure it handles all potential edge cases and high-load scenarios effectively.

# 2

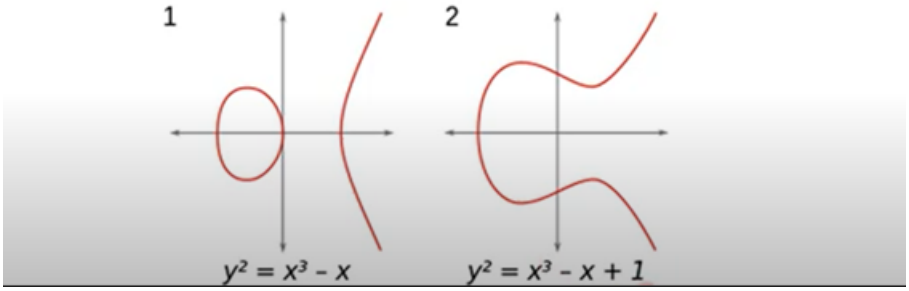
- Denial of service: No voter can force another voters to receive money. Each voter only receives their own money. Malicious attempts to extort funds within the withdraw function are not possible because each voter receives only their allocated funds.
- Re-entrancy: Re-entrancy is prevented since a every where needed we use re-entrancy flag.
- Bribing miner: It's impossible to bribe miners because We have to convince all the miners not to pass the votes that we don't want in the blocks within a few days. And it is impossible to bribe all existing miners. At least it is not possible secretly.
- Overflow: Attacks through overflow are mitigated because the entire sum of money is withdrawn in a single transaction.
- Front-running attack prevention: There is no overlap between the commit and reveal time intervals, effectively preventing any front-running attacks. This means that no one can manipulate the outcome of the auction by observing and preempting the actions of other participants.
- Non-response : There is no benefit in hiding votes.
- Consumption Gas:
- Zero- Knowledge Proof:
    1. Alice is supposed to prove to Bob that c number of votes has been given to person X.
    2. All the votes of person X are separated. We take the number of these votes as n.
    3. n divided by c numbers of these n votes are randomly selected.
    4. Bob encrypts a message with the public key of each of these votes and gives them all to Alice.

5. Alice tries to find the message with the private keys of the accounts that sent her votes and give it to Bob.
6. They design a hypothesis test that the null hypothesis of this test is that the expectation of finding this message by Alice is 1.
7. They repeat steps 3, 4 and 5 more than 30 times.
The result of the hypothesis test with a probability of 1-p value proves whether Alice's claim is true or not. Without Bob even knowing which vote was for Alice or being able to prove this fact elsewhere.

# 3 Diving Deep: My Persistence and Progress Over Days

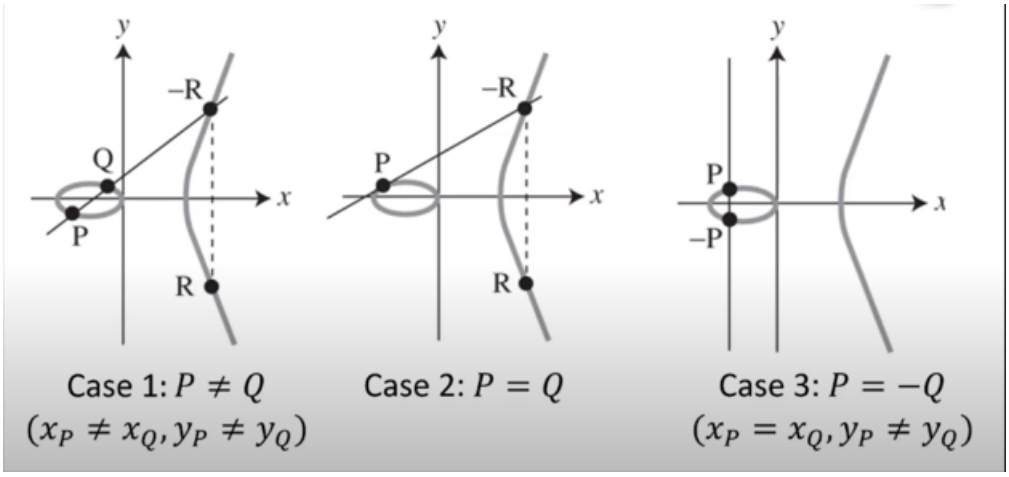
Initially, I developed the voting contract without incorporating signature verification. However, I later chose to explore adding signature checks into the contract for enhanced security. Upon further research, I discovered that both Ethereum and Bitcoin employ elliptic curve cryptography (ECC) rather than RSA. Motivated by this, I decided to delve deeper into understanding elliptic curve cryptography.

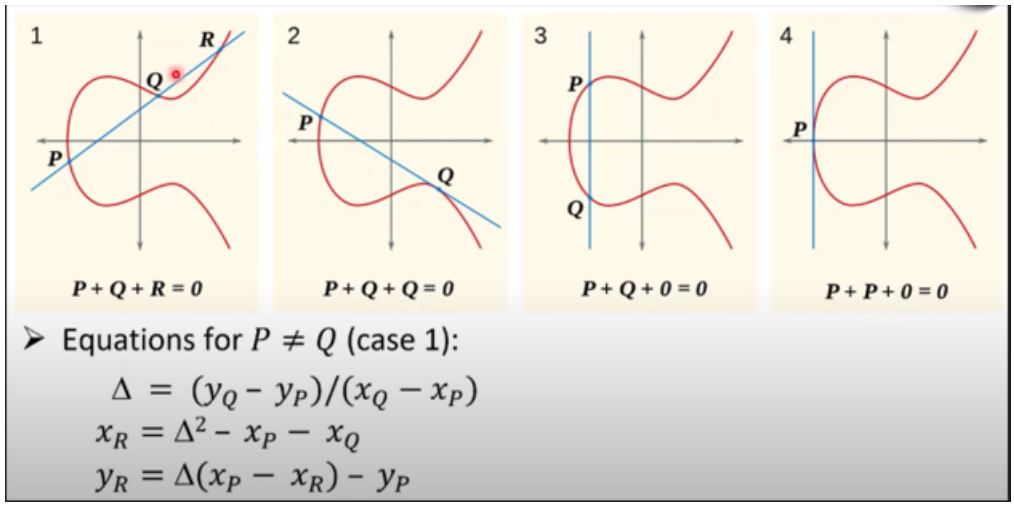
- **General Mathematical Form for Elliptic Curves (Weierstrass Equation):**
  - $y^2 = x^3 + ax + b$
  - for some $a$ and $b$ (curve parameter)




- **Encryption:**
  - Transforming points on curve $(P, K_{PU})$ into another point on the same curve $(C)$.
- **Main Idea (Abelian Group):**
  - To define the operation "+" such that the *sum* of two points on a curve also lies on the same curve. This is expressed as:
  - $R = P + Q$ where $P = (x_P, y_P), Q = (x_Q, y_Q),$ and $R = (x_R, y_R)$
  - Here, $P$ and $Q$ are points on the curve, and $R$ is the resultant point after adding $P$ and $Q$.
  - Additive Identity and Point at Infinity.
  - ($R =$ "0") (additive identity)

- Point at infinity: $(\infty)$
- $(0 = -0)$
- $(P + (-P) = 0)$




- Equations for $(P \neq Q)$ (case 1):

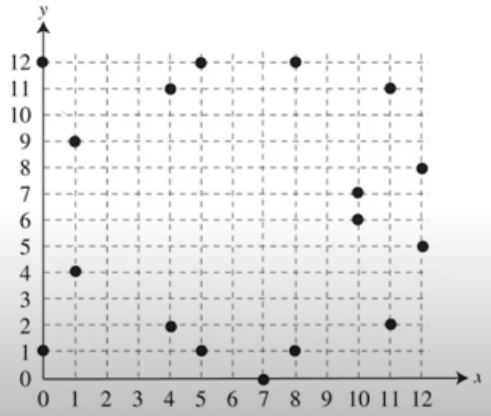
$$\Delta = \frac{(y_Q - y_P)}{(x_Q - x_P)}$$
$$x_R = \Delta^2 - x_P - x_Q$$
$$y_R = \Delta(x_P - x_R) - y_P$$

- **Elliptic curves over $Z_p$**
  - **Encryption Using Elliptic Curves and Modular Arithmetic**
  - Must be difficult to recover original points from R.
  - Modular arithmetic prevents "working backward", as in RSA

- Define "curve" as $E_p(a, b)$, where $p$ is the modulus and $a$, $b$ are the coefficients of $y^2 = x^3 + ax + b$
- We seek points $(x, y)$ such that:
- $y^2 \equiv (x^3 + ax + b) \mod p$
- For example, with $a = b = 1$ and $p = 13$, setting $x = 0$ leads to:
- $y^2 \mod 13 = 1 \mod 13$
- This results in two points: $(0, 1)$ and $(0, 12)$.
- Points on the elliptic curve $y^2 = x^3 + x + 1$ over GF(13):

| | |
|---|---|
| (0, 1) | (0, 12) |
| (1, 4) | (1, 9) |
| (4, 2) | (4, 11) |
| (5, 1) | (5, 12) |
| (7, 0) | (7, 0) |
| (8, 1) | (8, 12) |
| (10, 6) | (10, 7) |
| (11, 2) | (11, 11) |

Points

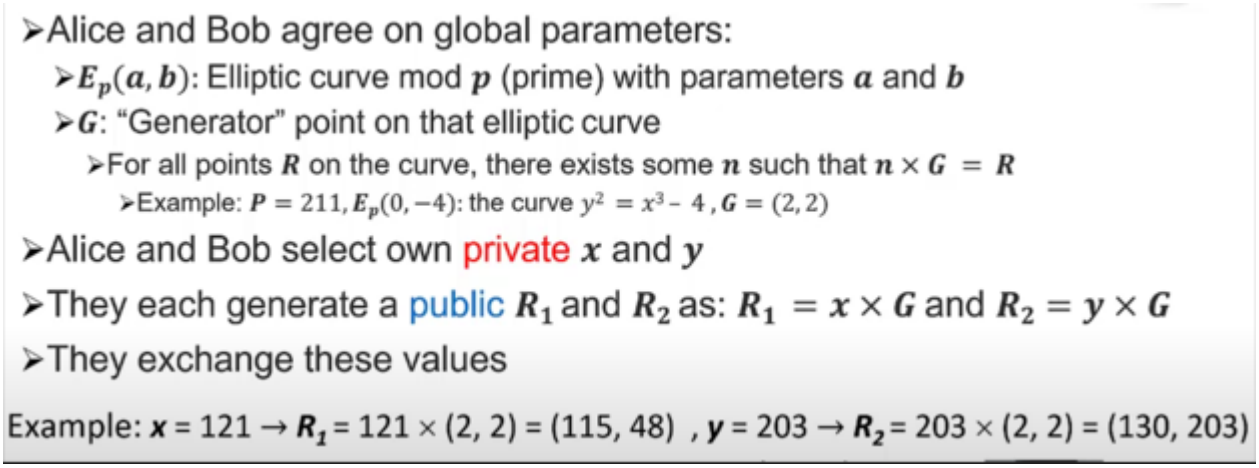


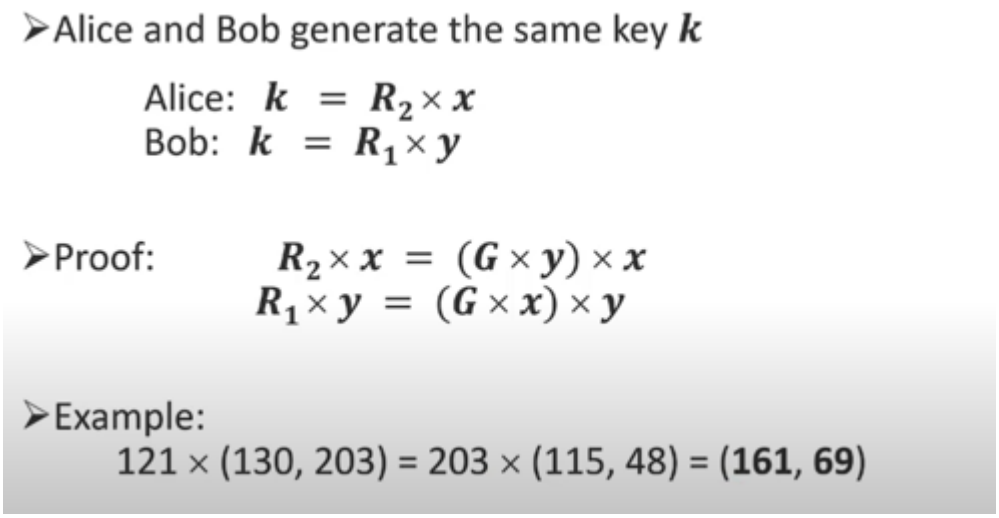
Graph

- **madular calculation:**
  - Computing $(x_R, y_R) = (x_P, y_P) + (x_Q, y_Q)$ involves the addition of two points on an elliptic curve. This process is essential for transforming two points, typically representing a key and plaintext, into a single point that corresponds to ciphertext.
  - The operation follows the same rules as addition of points on curves in space.
  - Addition, subtraction, and multiplication modulo a prime $(p)$.
  - Division is treated as multiplication by the modular inverse modulo a prime $(p)$.
- **Example of Modular Calculation:**
  - **Step 1: compute** $\Delta$ $\Delta = \frac{(y_Q - y_P)}{(x_Q - x_P)}$
    $\Delta = \frac{6-2}{(10-4)} \mod 13 = (6-2) \times (10-4)^{-1} \mod 13 = 4 \times 6^{-1} \mod 13 = 4 \times 11 \mod 13 = 5$
  - **Step 2: compute** $x_R = \Delta^2 - x_P - x_Q$
    $x_R = 25 - 4 - 10 \mod 13 = 11$
  - **Step 3: compute**$y_R = \Delta(x_P - x_R) - y_P$
    $y_R = 5 \times (4 - 11) - 2 \mod 13 = 2$
    $(4, 2) + (10, 6) = (11, 2) \rightarrow$ note: also on curve!

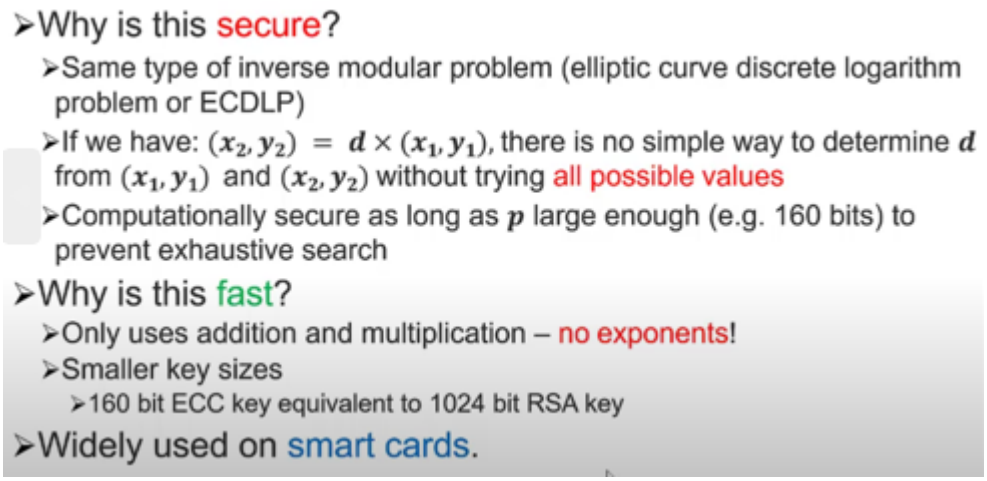
After exploring elliptic curve cryptography (ECC), I began researching earlier protocols that utilized ECC.

- **Diffie-Hellman:**




- **Security and speed of ECC:**




- **Signature generation**
- **parameter:**
  - **CURVE:** the elliptic curve field and equation used, a and b are curve parameters.
  - **G:** is the base point (generator point) of the curve, a point on the curve that generates a subgroup of large prime order n.
  - **n:** n is the order of G.
  - **p:** is the prime modulus.
  - **m:** the message to send.
- **algorithm:**

The order $n$ of the base point $G$ must be prime. Indeed, we assume that every nonzero element of the ring $\mathbb{Z}/n\mathbb{Z}$ is invertible, so that $\mathbb{Z}/n\mathbb{Z}$ must be a field. It implies that $n$ must be prime (cf. Bézout's identity).

Alice creates a key pair, consisting of a private key integer $d_A$, randomly selected in the interval $[1, n-1]$; and a public key curve point $Q_A = d_A \times G$. We use $\times$ to denote elliptic curve point multiplication by a scalar.

For Alice to sign a message $m$, she follows these steps:

1. Calculate $e = \text{HASH}(m)$. (Here HASH is a cryptographic hash function, such as SHA-2, with the output converted to an integer.)
2. Let $z$ be the $L_n$ leftmost bits of $e$, where $L_n$ is the bit length of the group order $n$. (Note that $z$ can be greater than $n$ but not longer.)
3. Select a cryptographically secure random integer $k$ from $[1, n-1]$.
4. Calculate the curve point $(x_1, y_1) = k \times G$.

5. Calculate $r = x_1 \bmod n$. If $r = 0$, go back to step 3.
6. Calculate $s = k^{-1}(z + rd_A) \bmod n$. If $s = 0$, go back to step 3.
7. The signature is the pair $(r, s)$. (And $(r, -s \bmod n)$ is also a valid signature.)

As the standard notes, it is not only required for $k$ to be secret, but it is also crucial to select different $k$ for different signatures. Otherwise, the equation in step 6 can be solved for $d_A$, the private key: given two signatures $(r, s)$ and $(r, s')$, employing the same unknown $k$ for different known messages $m$ and $m'$, an attacker can calculate $z$ and $z'$, and since $s - s' = k^{-1}(z - z')$ (all operations in this paragraph are done modulo $n$) the attacker can find $k = \frac{z-z'}{s-s'}$. Since $s = k^{-1}(z+rd_A)$, the attacker can now calculate the private key $d_A = \frac{sk-z}{r}$.

- **Signature verification algorithm:**
  For Bob to authenticate Alice's signature $(r, s)$ on a message $m$, he must have a copy of her public-key curve point $Q_A$. Bob can verify $Q_A$ is a valid curve point as follows:

  1. Check that $Q_A$ is not equal to the identity element $O$, and its coordinates are otherwise valid.
  2. Check that $Q_A$ lies on the curve.
  3. Check that $n \times Q_A = O$.

  After that, Bob follows these steps:

  1. Verify that $r$ and $s$ are integers in $[1, n - 1]$. If not, the signature is invalid.
  2. Calculate $e = \text{HASH}(m)$, where HASH is the same function used in the signature generation.
  3. Let $z$ be the $L_n$ leftmost bits of $e$.
  4. Calculate $u_1 = zs^{-1} \bmod n$ and $u_2 = rs^{-1} \bmod n$.
  5. Calculate the curve point $(x_1, y_1) = u_1 \times G + u_2 \times Q_A$. If $(x_1, y_1) = O$ then the signature is invalid.

  The signature is valid if $r \equiv x_1 \bmod n$, invalid otherwise.

  Note that an efficient implementation would compute inverse $s^{-1} \bmod n$ only once. Also, using Shamir's trick, a sum of two scalar multiplications $u_1 \times G + u_2 \times Q_A$ can be calculated faster than two scalar multiplications done independently.

- **Blind Signature Based on ECC:**

In 2010, Jeng et al. [4] proposed a fast blind signature scheme, based on the ECDLP. This scheme does not compute modular exponentiation consecutively. Instead, a user can obtain a signature and verify it only through scalar multiplication of points on elliptic curves, for example, point addition and point doubling. ECC requires much lesser numbers for its operations; hence the scheme is very efficient. Let an elliptic group $E_p(a,b)$ be formed as $y^2 = x^3 + ax + b(\mathrm{mod}\, p)$, where $4a^3 + 27b^2 \neq 0 \bmod p$ such that $E_p(a,b)$ is appropriate for cryptography. And then a base point $G$ on $E_p$ is determined whose order is a very large value $u$ such that $u \cdot G = O$. The protocol is described below.

(i) *Initialization.* $R$ randomly selects a secret key $n_i$ and generates the corresponding public key $P_i$ as $P_i \equiv n_i \cdot G(\mathrm{mod}\, p)$. Likewise, $S$ chooses a random number $n_j$ as the secret key, and the corresponding public key is $P_j \equiv n_j \cdot G(\mathrm{mod}\, p)$.

(ii) *Blinding.* $R$ retains a message $m$, sets $\alpha \equiv m \cdot (n_i \cdot P_i) \cdot (\mathrm{mod}\, p)$, and sends the blinded message $\alpha$ to $S$.

(iii) *Signing.* $S$ arbitrarily chooses another blinding factor $n_v$ and creates a pair of blind signatures $(r,s)$, where $r \equiv n_v \cdot \alpha(\mathrm{mod}\, p)$ and $s \equiv (n_v + n_s) \cdot \alpha \cdot (\mathrm{mod}\, p)$ . Then $S$ forwards the message-signature pair $(\alpha,(r,s))$ to $R$ and keeps $(\alpha, n_v)$ in private.

(iv) *Unblinding.* $R$ removes the blind signature $(r,s)$ by applying the secret key $n_i$, along with $S$'s public key $P_j$ to yield $s' \equiv s - m \cdot n_i P_j (\mathrm{mod}\, p)$. And then $R$ calculates $m' = n_i \cdot (n_i - 1)m$.

(v) *Verification.* Anyone can use $S$'s public key $P_j$ to verify the authentication of the signature $(m', s', r)$ by checking whether the given formula $r \equiv s' - m' \cdot P_j (\mathrm{mod}\, p)$ has been satisfied.

- **Sources:**


Introduction to Cryptocurrencies

wikipedia.Elliptic_Curve_Digital_Signature_Algorithm

www.academia.edu

www.hindawi.com