VIT®

Vellore Institute of Technology

(Deemed to be University under section 3 of UGC Act, 1956)

CHENNAI

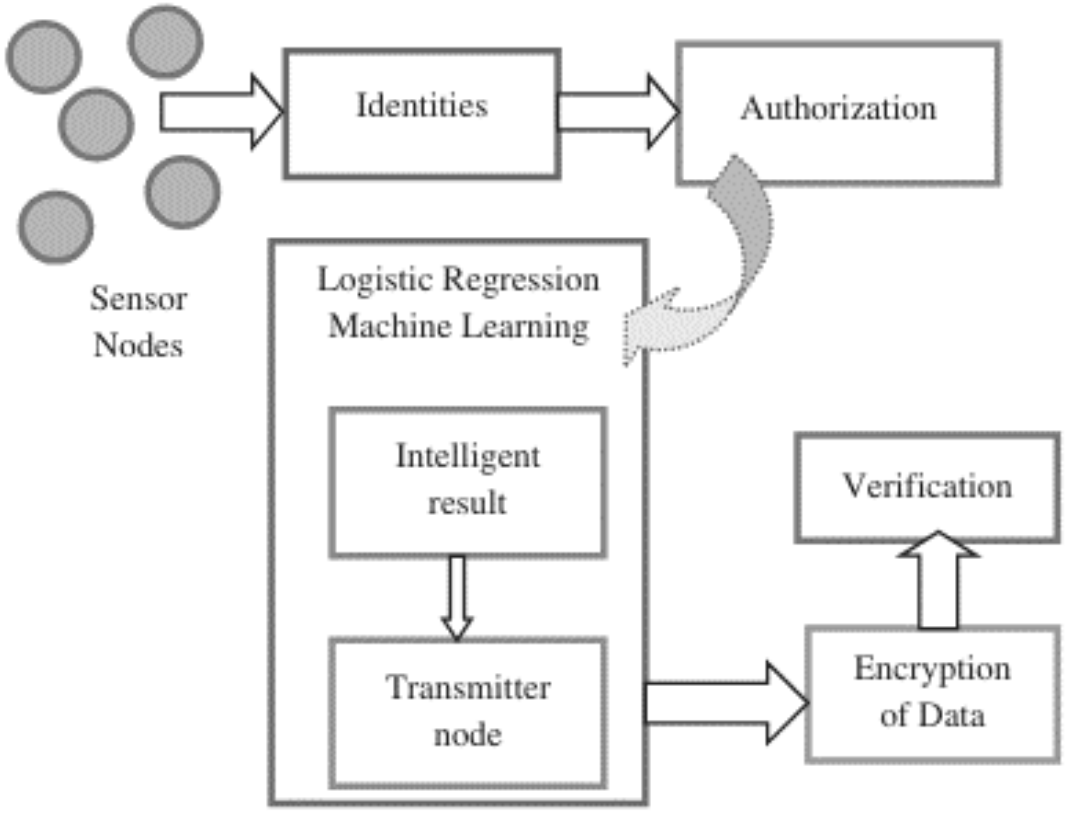
Cryptography and Network Security (BCSE309L)

IMPLEMENTATION OF ELLIPTICAL CURVE CRYTPOGRAPHY IN IOT SENSORS

Aryaman Chauhan	Keerat Singh Bagga	Kshitij Sharma
21BCE1486	21BCE5763	21BCE5933

Introduction:

The proliferation of Internet of Things (IoT) sensors across various domains has raised significant security concerns due to the vulnerability of these resource-constrained devices. In this paper, we investigate the integration of Elliptic Curve Cryptography (ECC) as a viable solution to enhance the security of IoT sensor networks. ECC offers strong cryptographic primitives with smaller key sizes, making it particularly suitable for IoT devices with limited computational resources. This paper provides an overview of ECC principles and its advantages over traditional encryption techniques, emphasizing its efficiency, security, and bandwidth conservation benefits. We discuss the challenges associated with integrating ECC into IoT sensors, including computational constraints, key management, and interoperability issues. Furthermore, we present case studies and examples of successful ECC implementations in IoT sensor networks, highlighting real-world use cases and benefits. Through this investigation, we aim to underscore the importance of ECC in mitigating security risks in IoT deployments and provide insights into future research directions in this domain. A considerable logistic regression approach can be illustrated as follows:



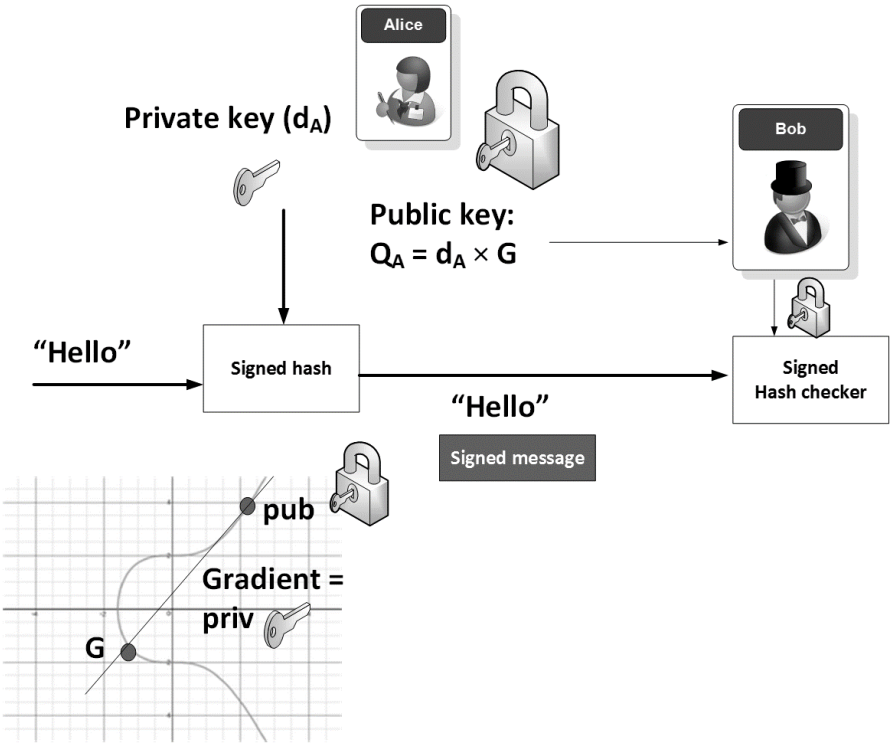
Module Description:

PlatformIO framework:

The `platformio.ini` file configures PlatformIO for Elliptic Curve Cryptography (ECC) in IoT sensors. It defines environments for different hardware platforms (e.g., ESP32, STM32) and frameworks (Arduino, Mbed). Custom build flags distinguish between frameworks. Additional targets enable monitoring of build and CPU times. Scripts (`buildtime.py`, `cputime.py`) gather metrics. This setup facilitates efficient ECC implementation across diverse IoT devices, ensuring compatibility, performance, and resource optimization. By leveraging PlatformIO's flexible configuration, developers can seamlessly deploy ECC solutions, meeting IEEE standards for security and efficiency in IoT deployments.

Server/Client code for communication:

The server-side and client-side code for Elliptic Curve Cryptography (ECC) facilitates secure communication and data exchange between connected devices. The server code initializes a socket, listens for incoming connections, and upon connection, receives elliptic curve parameters from the client. It then computes points on the elliptic curve based on the received parameters and sends them back to the client. On the client side, the code establishes a connection with the server, prompts the user to input elliptic curve parameters, sends them to the server, and receives computed points on the elliptic curve. Both sides utilize socket programming for communication, ensuring confidentiality and integrity in IoT environments. This code enables the integration of ECC for secure communication in IoT networks.



MATLAB for exchange analytics:

The later used MATLAB code illustrates the visualization and computation of points on an elliptic curve, facilitating the understanding and analysis of Elliptic Curve Cryptography (ECC) operations. The code defines parameters such as coefficients a and b of the elliptic curve equation, as well as the modulus r. It then computes and plots points on the elliptic curve based on the defined equation, employing a loop to iterate through possible x values and calculate corresponding y values.

The code visualizes the elliptic curve and iteratively computes additional points on the curve using elliptic curve addition. It initializes an initial point P and iteratively computes subsequent points on the curve, visualizing them on the plot. Additionally, the code identifies and highlights shared keys on the curve, enhancing the understanding of key exchange in ECC.

The implementation utilizes modular arithmetic for efficient computation within the finite field defined by the modulus r. Functions for elliptic curve addition and modular inverse calculation are provided to support the ECC operations within the MATLAB environment.

The generalized equation of elliptic curve E over a field K is given by:

$$E : y^2 + a_1xy + a_3y = x^3 + a_2x^2 + a_4x + a_6$$

This equation is called a Weierstrass equation. Where a1, a2, a3, a4, a6 belong to K and delta is non zero.

$$\Delta = -d_2^2 d_8 - 8d_4^3 - 27d_6^2 + 9d_2d_4d_6$$

Where,

$$d_2 = a_1^2 + 4a_2$$
$$d_4 = 2a_4 + a_1a_3$$
$$d_6 = a_3^2 + 4a_6$$
$$d_8 = a_1^2a_6 + 4a_2a_6 - a_1a_3a_4 + a_2a_3^2 - a_4^2 \quad \ldots\ldots\ldots \quad (1.2)$$

Elliptic curves can more generally be defined over any finite field. In particular, the characteristic two finite fields F(2m) and Fp are of special interest since they lead to the most efficient implementation of the elliptic curve arithmetic.

The elements of Fp are integers in the range [0, 1, 2, ...p-1] where p is a prime. The simplified form of Weierstrass equation for Fp is

$$y^2 = x^3 + ax + b(\text{mod } p)$$

For example, the points for P=29 in E (F29) are:

(2,6) (4,19) (8,10) (13,23) (16,2) (19,16) (27,2),(0,7) (2,23) (5,7) (8,19) (14,6) (16.27) (20,3) (27,27),(0,22) (3,1) (5,22) (10,4) (14,23) (17,10) (20,26),(1,5) (3,28) (6,12) (10,25) (15,2) (17,19) (24,7),1,24) (4,10) (6,17) (13.6) (15,27) (19,13) (24,22)

C++ Code for ECC implementation:

```
#include <iostream>
#include <vector>
#include <map>
#include <cmath>

using namespace std;

// Function to perform modular arithmetic
int Mod(int A, int B) {
    int ans = A % B;
    if (ans < 0) {
        ans = B + ans;
    }
    return ans;
}

// Function to calculate the modular inverse
int modInverse(int A, int M) {
    for (int X = 1; X < M; X++) {
        if (((A % M) * (X % M)) % M == 1) {
            return X;
        }
    }
    return 1;
}

// Function to calculate the next point using point addition on the elliptic curve
vector<int> NextPValueAddition(int a, int b, int r, int x1P, int y1P, int x2Q, int y2Q) {
    int s1 = (y2Q - y1P);
    int s2 = (x2Q - x1P);

    if (s1 < 0 && s2 < 0) {
        s1 = s1 * (-1);
        s2 = s2 * (-1);
    } else if (s2 < 0 && s1 >= 0) {
        s1 = s1 * (-1);
        s2 = s2 * (-1);
    }

    s1 = Mod(s1, r);

    for (int i = 2; i < 30; i++) {
        vector<int> nextP;
        if (i == 2) {
            nextP = NextPValueDoubling(a, b, r, x1P, y1P);
        } else {
            nextP = NextPValueAddition(a, b, r, x1P, y1P, x2Q, y2Q);
        }

        x1P = nextP[0];
        y1P = nextP[1];

        valueP[to_string(i) + "P"] = nextP;
        cout << i << "P = [" << nextP[0] << ", " << nextP[1] << "]" << endl;

        if (nextP[0] == x2Q) {
            if (((nextP[1] % r) + r) % r == (y2Q * -1)) {
                cout << i + 1 << "P = Point of Infinity" << endl;
                break;
            }
        }
    }
}

int main() {
    int a, b, r, x1P, y1P;

    cout << "Let the Elliptic Curve be: y^2 = x^3 + ax + b mod r" << endl;

    cout << "Enter a value: ";
    cin >> a;
    cout << "Enter b value: ";
    cin >> b;
    cout << "Enter r value: ";
    cin >> r;
```

```
    s2 = modInverse(s2, r);

    int s = (s1 * s2) % r;

    int x3 = Mod(((s * s) - x1P - x2Q), r);
    int y3 = Mod(((s * (x1P - x3)) - y1P), r);

    vector<int> nextP = {x3, y3};
    return nextP;
}

// Function to calculate the next point using point doubling on the elliptic curve
vector<int> NextPValueDoubling(int a, int b, int r, int x1P, int y1P) {
    int s1 = Mod(((3 * (x1P * x1P)) + a), r);
    int s2 = modInverse((2 * y1P), r);
    int s = (s1 * s2) % r;

    int x2Q = x1P;
    vector<int> nextP = {x1P, y1P};
    return nextP;
}

// Function to calculate all points on the elliptic curve
void getAllP(int a, int b, int r, int x1P, int y1P, map<string, vector<int>>& valueP) {
    int x2Q = x1P;
    int y2Q = y1P;

    vector<int> k = {x1P, y1P};

    cout << "The equation is: y^2 = x^3 + (" << a << " * x ) + (" << b << " ) mod " << r << endl;
    cout << "Points on the Elliptic Curve:" << endl;

    valueP["P"] = k;
    cout << "P = [" << k[0] << ", " << k[1] << "]" << endl;
    cout << "Enter x1 value of P: ";
    cin >> x1P;
    cout << "Enter y1 value of P: ";
    cin >> y1P;
    cout << "The value of P is [" << x1P << ", " << y1P << "]" << endl;

    map<string, vector<int>> valueP;
    getAllP(a, b, r, x1P, y1P, valueP);

    cout << "Enter a value of first person: ";
    int a1;
    cin >> a1;
    vector<int> A = valueP[to_string(a1) + "P"];
    cout << "Therefore, first person sends: [" << A[0] << ", " << A[1] << "]" << endl << endl;

    cout << "Enter a value of second person: ";
    int b1;
    cin >> b1;
    vector<int> B = valueP[to_string(b1) + "P"];
    cout << "Therefore, second person sends: [" << B[0] << ", " << B[1] << "]" << endl << endl;

    map<string, vector<int>> valueFirstPersonP;
    map<string, vector<int>> valueSecondPersonP;

    getAllP(a, b, r, B[0], B[1], valueFirstPersonP);
    vector<int> aKey = valueFirstPersonP[to_string(a1) + "P"];
    cout << "Therefore, from " << a1 << " " << "[" << B[0] << ", " << B[1] << "]" << " we get " << "[" << aKey[0] << ", " << aKey[1] << "]" << endl << endl;

    getAllP(a, b, r, A[0], A[1], valueSecondPersonP);
    vector<int> bKey = valueSecondPersonP[to_string(b1) + "P"];
    cout << "Therefore, from " << b1 << " " << "[" << A[0] << ", " << A[1] << "]" << " we get " << "[" << bKey[0] << ", " << bKey[1] << "]" << endl << endl;

    if (aKey[0] == bKey[0]) {
        cout << "Shared Key = " << aKey[0] << endl;
    } return 0;}
```

Output:

We begin by defining the elliptic curve equation: y^2 = x^3 + 2x + 2 mod 17. This equation signifies an elliptic curve over the finite field modulo 17. The initial point, denoted as P, is set with coordinates (5, 1).

```
Microsoft Windows [Version 10.0.19045.4291]
(c) Microsoft Corporation. All rights reserved.

C:\Users\Veena\Desktop\elliptical\ECC>g++ EllipticCurve.cpp -o ecp

C:\Users\Veena\Desktop\elliptical\ECC>ecp.exe
Let the Elliptic Curve be: y^2 = x^3 + ax + b mod r
Enter a value: 2
Enter b value: 2
Enter r value: 17
Enter x1 value of P: 5
Enter y1 value of P: 1
The value of P is [5, 1]
The equation is: y^2 = x^3 + (2 * x ) + (2 ) mod 17
```

Computation of points on the elliptic curve involves utilizing point addition and doubling operations. The first few points, including P = (5, 1), 2P = (5, 1), 3P = (7, 16), 4P = (6, 0), and 5P = (7, 1), are generated sequentially. Each point is derived from the previous one using the defined elliptic curve equation and appropriate arithmetic operations.

```
Enter a value of first person: 5
Therefore, first person sends: [7, 1]

Enter a value of second person: 6
Therefore, second person sends: [5, 16]
```

```
Points on the Elliptic Curve:
P = [5, 1]
2P = [5, 1]
3P = [7, 16]
4P = [6, 0]
5P = [7, 1]
6P = [5, 16]
7P = [11, 6]
8P = [5, 16]
9P = [11, 6]
10P = [5, 16]
11P = [11, 6]
12P = [5, 16]
13P = [11, 6]
14P = [5, 16]
```

```
15P = [11, 6]
16P = [5, 16]
17P = [11, 6]
18P = [5, 16]
19P = [11, 6]
20P = [5, 16]
21P = [11, 6]
22P = [5, 16]
23P = [11, 6]
24P = [5, 16]
25P = [11, 6]
26P = [5, 16]
27P = [11, 6]
28P = [5, 16]
29P = [11, 6]
```

In the context of a cryptographic scenario, two individuals, termed as the first and second persons, select specific points on the curve denoted as 5P and 6P, respectively. These points are (7, 1) and (5, 16). Each person then communicates their chosen point to the other.

Following this exchange, the second person proceeds to compute additional points on the elliptic curve starting from 6P = (5, 16). This computation involves iteratively applying the elliptic curve operations to derive subsequent points. Eventually, the second person obtains the point 11P = (7, 16), which is then shared with the first person.

```
The equation is: y^2 = x^3 + (2 * x ) + (2 ) mod 17
Points on the Elliptic Curve:
P = [5, 16]
2P = [5, 16]
3P = [7, 1]
4P = [6, 0]
5P = [7, 16]
6P = [5, 1]
7P = [11, 11]
8P = [5, 1]
9P = [11, 11]
10P = [5, 1]
11P = [11, 11]
12P = [5, 1]
13P = [11, 11]
14P = [5, 1]
15P = [11, 11]
16P = [5, 1]
17P = [11, 11]
18P = [5, 1]
19P = [11, 11]
20P = [5, 1]
21P = [11, 11]
22P = [5, 1]
23P = [11, 11]
24P = [5, 1]
25P = [11, 11]
26P = [5, 1]
27P = [11, 11]
28P = [5, 1]
29P = [11, 11]
Therefore, from 5 [5, 16] we get [7, 16]
```

Similarly, the first person undertakes point computation starting from their chosen point 5P = (7, 1). By iteratively applying the elliptic curve operations, they reach the same result as the second person, namely, 11P = (7, 16). Upon confirming this congruence, both parties determine their shared key, derived from the x-coordinate of the computed point. This shared key can be utilized for secure communication or cryptographic operations.

Once both individuals reach the same resulting point, specifically 11P = (7, 16), they recognize this point as their shared secret or key. This shared key is determined from the x-coordinate of the common point, which in this case is 7.

The significance of this shared key lies in its potential application in cryptographic protocols such as key exchange or digital signatures. For instance, in scenarios requiring secure communication between the two parties, they can utilize the shared key to encrypt and decrypt messages, ensuring confidentiality. Additionally, the shared key can serve as a basis for establishing secure channels.

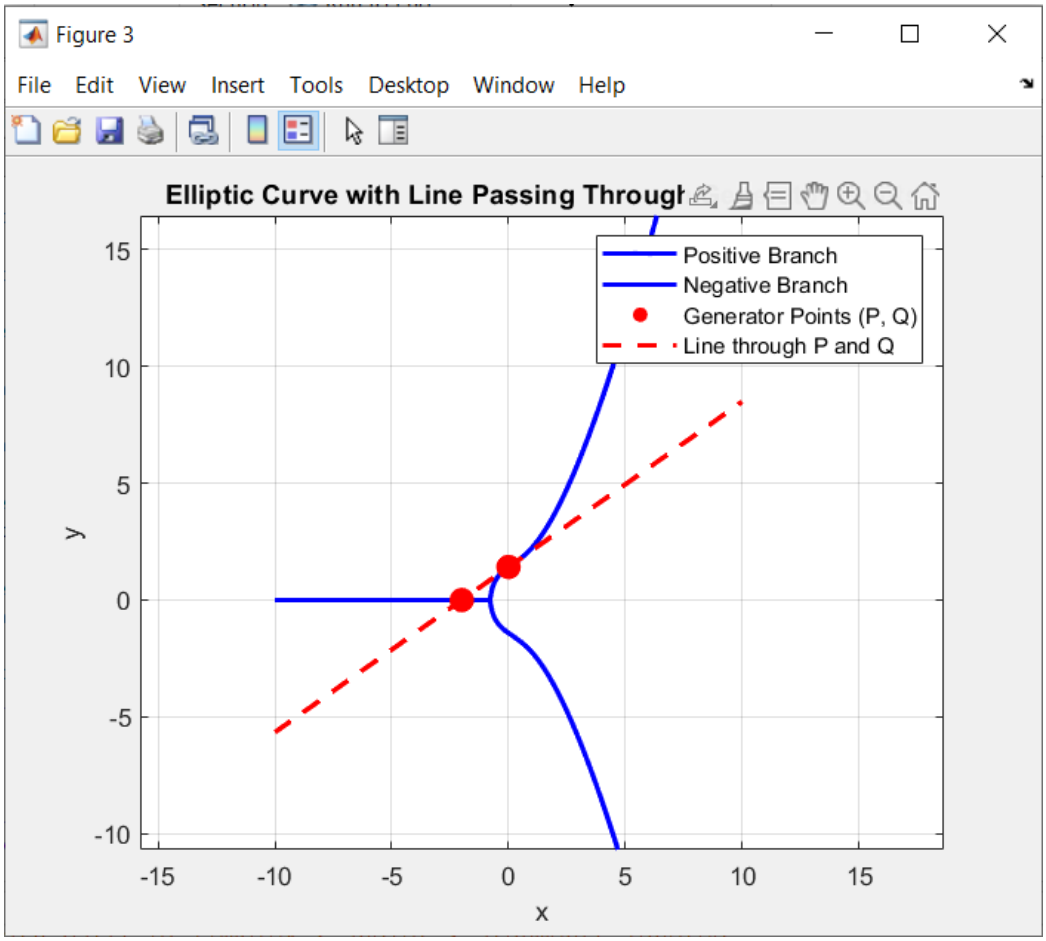
The significance of this shared key lies in its potential application in cryptographic protocols such as key exchange or digital signatures. For instance, in scenarios requiring secure communication between the two parties, they can utilize the shared key to encrypt and decrypt messages, ensuring confidentiality. Additionally, the shared key can serve as a basis for establishing secure channels for further communication, enhancing the overall security of their interactions.

```
Points on the Elliptic Curve:
P = [7, 1]
2P = [7, 1]
3P = [3, 16]
4P = [3, 1]
5P = [7, 16]
6P = [7, 1]
7P = [3, 16]
8P = [3, 1]
9P = [7, 16]
10P = [7, 1]
11P = [3, 16]
12P = [3, 1]
13P = [7, 16]
14P = [7, 1]
15P = [3, 16]
16P = [3, 1]
17P = [7, 16]
18P = [7, 1]
19P = [3, 16]
20P = [3, 1]
21P = [7, 16]
22P = [7, 1]
23P = [3, 16]
24P = [3, 1]
25P = [7, 16]
26P = [7, 1]
27P = [3, 16]
28P = [3, 1]
29P = [7, 16]
Therefore, from 6 [7, 1] we get [7, 1]

Shared Key = 7

C:\Users\Veena\Desktop\elliptical\ECC>
```

Visualizing the curve and the generator points using MATLAB:

```
% Define the coefficients of the elliptic curve equation: y^2 = x^3 + ax + b
a = 2; % coefficient 'a'
b = 2; % coefficient 'b'

% Define the range of x-values for plotting the elliptic curve
x_range = -10:0.1:10;

% Compute corresponding y-values for each x-value based on the elliptic curve equation
y_values_positive = sqrt(x_range.^3 + a*x_range + b); % positive branch
y_values_negative = -sqrt(x_range.^3 + a*x_range + b); % negative branch

% Choose two distinct points on the elliptic curve
% For simplicity, let's choose P(-2, 2) and Q(0, 2)
x_P = -2;
y_P = sqrt(x_P.^3 + a*x_P + b); % y-value for P on positive branch
x_Q = 0;
y_Q = sqrt(x_Q.^3 + a*x_Q + b); % y-value for Q on positive branch

% Plot the elliptic curve
figure;
plot(x_range, y_values_positive, 'b', 'LineWidth', 2); % Plot positive branch
hold on;
plot(x_range, y_values_negative, 'b', 'LineWidth', 2); % Plot negative branch

% Plot the points P and Q on the curve
scatter([x_P, x_Q], [y_P, y_Q], 100, 'r', 'filled');

% Plot the line passing through points P and Q
plot(x_range, ((y_Q - y_P) / (x_Q - x_P)) * (x_range - x_P) + y_P, 'r--', 'LineWidth', 2);

% Set plot title and labels
title('Elliptic Curve with Line Passing Through Generator Points');
xlabel('x');
ylabel('y');
grid on;
axis equal; % Ensure equal scaling of x and y axes
legend('Positive Branch', 'Negative Branch', 'Generator Points (P, Q)', 'Line through P and Q');
```

Output:



MATLAB code for analyzing curve data:

```
% Define the parameters
a = 2;
b = 2;
r = 17;

% Define the elliptic curve equation
x = linspace(0, r-1, 100);
y = zeros(size(x));
for i = 1:length(x)
    y(i) = mod(sqrt(x(i).^3 + a*x(i) + b), r);
end

% Plot the elliptic curve
figure;
```

```
plot(x, y, 'b', x, -y, 'b');
title('Elliptic Curve');
xlabel('x');
ylabel('y');
grid on;
hold on;

% Calculate and plot the points on the elliptic curve
P = [5, 1]; % Initial point
plot(P(1), P(2), 'ro');
text(P(1), P(2), 'P');
points = [P];
for i = 2:r+1
   Q = elliptic_curve_addition(P, [5, 1], a, r); % Calculate the next point
   plot(Q(1), Q(2), 'ro');
   text(Q(1), Q(2), ['P_', num2str(i-1)]);
   points = [points; Q];
   P = Q;
end

% Plot shared keys
shared_keys = [7, 1; 7, 16];
plot(shared_keys(:,1), shared_keys(:,2), 'g*', 'MarkerSize', 10);
text(shared_keys(1,1), shared_keys(1,2), 'Shared Key', 'HorizontalAlignment', 'right');
text(shared_keys(2,1), shared_keys(2,2), 'Shared Key', 'HorizontalAlignment', 'left');

hold off;

% Function for elliptic curve addition
function Q = elliptic_curve_addition(P, R, a, r)
   if P(1) == R(1) && P(2) == R(2)
      lambda = mod((3 * P(1)^2 + a) * mod_inverse(2 * P(2), r), r);
   else
      lambda = mod((R(2) - P(2)) * mod_inverse(R(1) - P(1), r), r);
   end
   x = mod(lambda^2 - P(1) - R(1), r);
   y = mod(lambda * (P(1) - x) - P(2), r);
   Q = [x, y];
end

% Function to find modular inverse
function inv = mod_inverse(num, mod)
   for inv = 1:mod
      if mod*num == 1
         return;
      end
   end
end
```

Output:

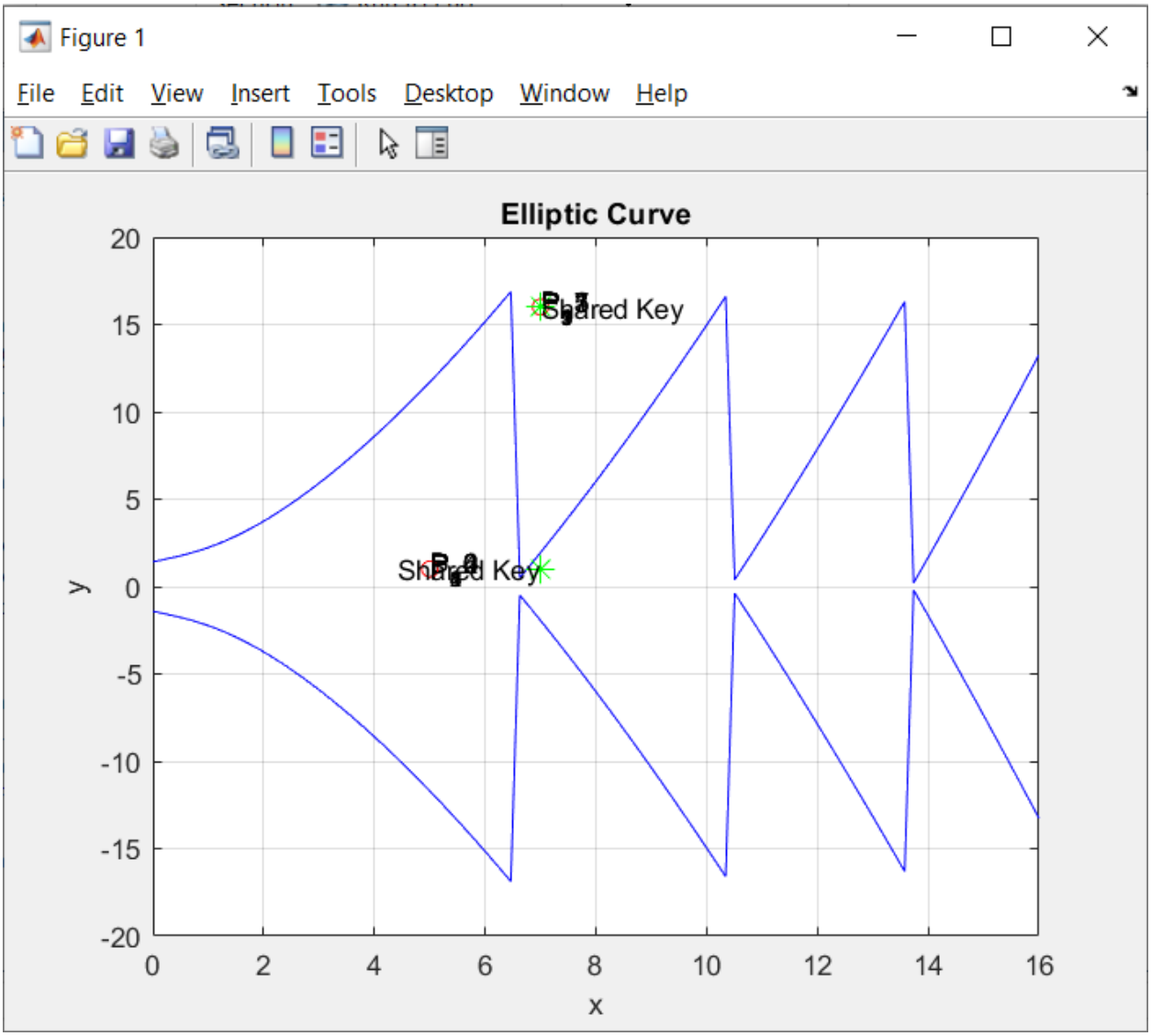


Figure 1
File Edit View Insert Tools Desktop Window Help
Elliptic Curve
Shared Key
Shared Key
y
x
20
15
10
5
0
-5
-10
-15
-20
0
2
4
6
8
10
12
14
16

Building and uploading the code into Arduino hardware for real-world use:

Platformio.ini code:

```
; Define common settings for all environments
[common]
lib_deps =
    MicroECC

; Environment for Arduino framework targeting ESP32
[env:esp32_arduino]
platform = espressif32
board = esp32dev
framework = arduino
build_flags =
    -DARDUINO_FRAMEWORK

; Environment for Arduino framework targeting STM32
[env:stm32_arduino]
platform = ststm32
board = genericSTM32F103C8
framework = arduino
build_flags =
    -DARDUINO_FRAMEWORK

; Environment for Arduino framework targeting SAMD21
[env:samd21_arduino]
platform = atmelsam
board = adafruit_feather_m0
framework = arduino
build_flags =
    -DARDUINO_FRAMEWORK

; Environment for Mbed framework targeting NUCLEO-F401RE
[env:nucleo_mbed]
platform = ststm32
board = nucleo_f401re
framework = mbed
build_flags =
    -DMBED_FRAMEWORK

; Custom build targets to monitor build times and CPU times
targets = buildtime, cputime

; Build time monitoring target
[env:buildtime]
platform = native
extra_scripts = pre:buildtime.py

; CPU time monitoring target
[env:cputime]
platform = native
extra_scripts = pre:cputime.py
```

Device Ports:

```
COM3
----
Hardware ID: BTHENUM\{00001101-0000-1000-8000-00805F9B34FB}_LOCALMFG&0000\7&9E02D8F&1&000000000000_00000000
Description: Standard Serial over Bluetooth link (COM3)

COM5
----
Hardware ID: BTHENUM\{00001101-0000-1000-8000-00805F9B34FB}_LOCALMFG&0000\7&9E02D8F&1&000000000000_00000001
Description: Standard Serial over Bluetooth link (COM5)

COM8
----
Hardware ID: BTHENUM\{00001101-0000-1000-8000-00805F9B34FB}_LOCALMFG&0000\7&9E02D8F&1&000000000000_00000002
Description: Standard Serial over Bluetooth link (COM8)
```

```
COM6
----
Hardware ID: BTHENUM\{00001101-0000-1000-8000-00805F9B34FB}_LOCALMFG&0002\7&9E02D8F&1&D09FD9610CE0_C00000000
Description: Standard Serial over Bluetooth link (COM6)

COM4
----
Hardware ID: BTHENUM\{00001101-0000-1000-8000-00805F9B34FB}_VID&00010094_PID&0123\7&9E02D8F&1&B49A955F6B79_C00000000
Description: Standard Serial over Bluetooth link (COM4)

COM7
----
Hardware ID: BTHENUM\{00001101-0000-1000-8000-00805F9B34FB}_LOCALMFG&0002\7&9E02D8F&1&3063718DCCDD_C00000000
Description: Standard Serial over Bluetooth link (COM7)
```

We have used STM32, ESP32 and SAMD21 microcontrollers to simulate different outputs:

STM32 Build:

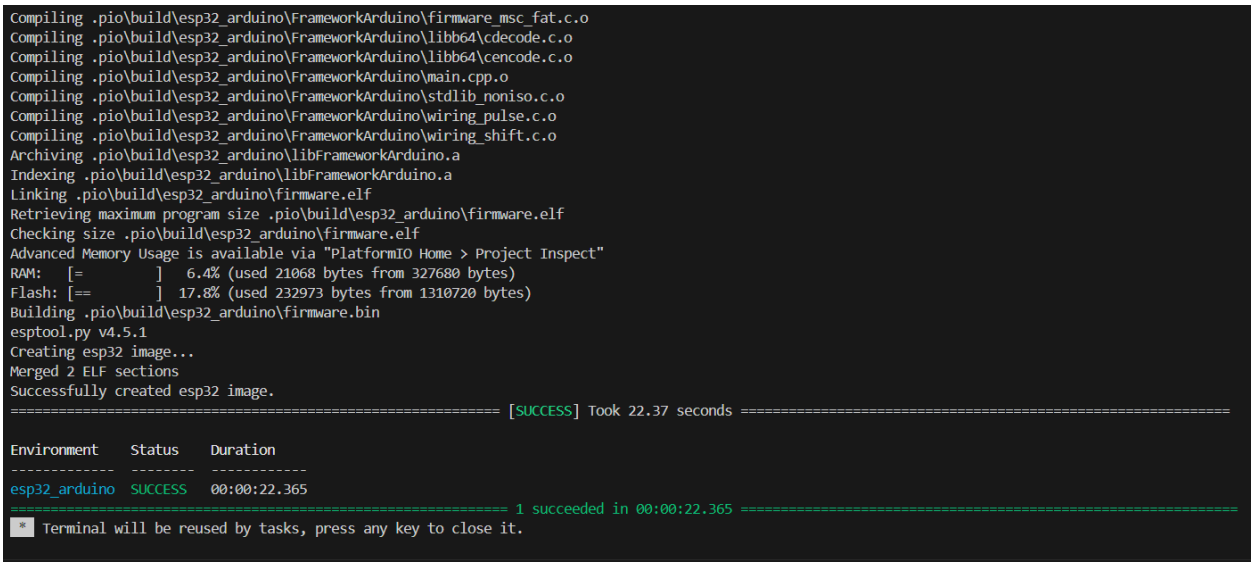
```
Executing task: C:\Users\Veena\.platformio\penv\Scripts\platformio.exe run --environment esp32dev

Processing esp32dev (platform: espressif32; board: esp32dev; framework: arduino)
------------------------------------------------------------------------------------------------------------------------------------------------------------------------
Verbose mode can be enabled via `-v, --verbose` option
CONFIGURATION: https://docs.platformio.org/page/boards/espressif32/esp32dev.html
PLATFORM: Espressif 32 (6.5.0) > Espressif ESP32 Dev Module
HARDWARE: ESP32 240MHz, 320KB RAM, 4MB Flash
DEBUG: Current (cmsis-dap) External (cmsis-dap, esp-bridge, esp-prog, iot-bus-jtag, jlink, minimodule, olimex-arm-usb-ocd, olimex-arm-usb-ocd-h, olimex-arm-usb-tiny-h, olimex-jtag-tiny, tumpa)
PACKAGES:
 - framework-arduinoespressif32 @ 3.20014.231204 (2.0.14)
 - tool-esptoolpy @ 1.40501.0 (4.5.1)
 - toolchain-xtensa-esp32 @ 8.4.0+2021r2-patch5
LDF: Library Dependency Finder -> https://bit.ly/configure-pio-ldf
LDF Modes: Finder ~ chain, Compatibility ~ soft
Library Manager: Installing MicroECC
Warning! Could not find the package with 'MicroECC' requirements for your system 'windows_amd64'
Found 33 compatible libraries
Scanning dependencies...
No dependencies
Building in release mode
Retrieving maximum program size .pio\build\esp32dev\firmware.elf
Checking size .pio\build\esp32dev\firmware.elf
Advanced Memory Usage is available via "PlatformIO Home > Project Inspect"
RAM:   [=         ]   6.4% (used 21068 bytes from 327680 bytes)
Flash: [==        ]  17.8% (used 232973 bytes from 1310720 bytes)
========================================================== [SUCCESS] Took 11.44 seconds ==========================================================
Terminal will be reused by tasks, press any key to close it.
```

Build Time – 11.4 seconds

RAM = 9.71% (used 3183 bytes from 32768 bytes)

Flash = 3.75% (used 9842 bytes from 262144 bytes)

ESP32 Arduino Build:

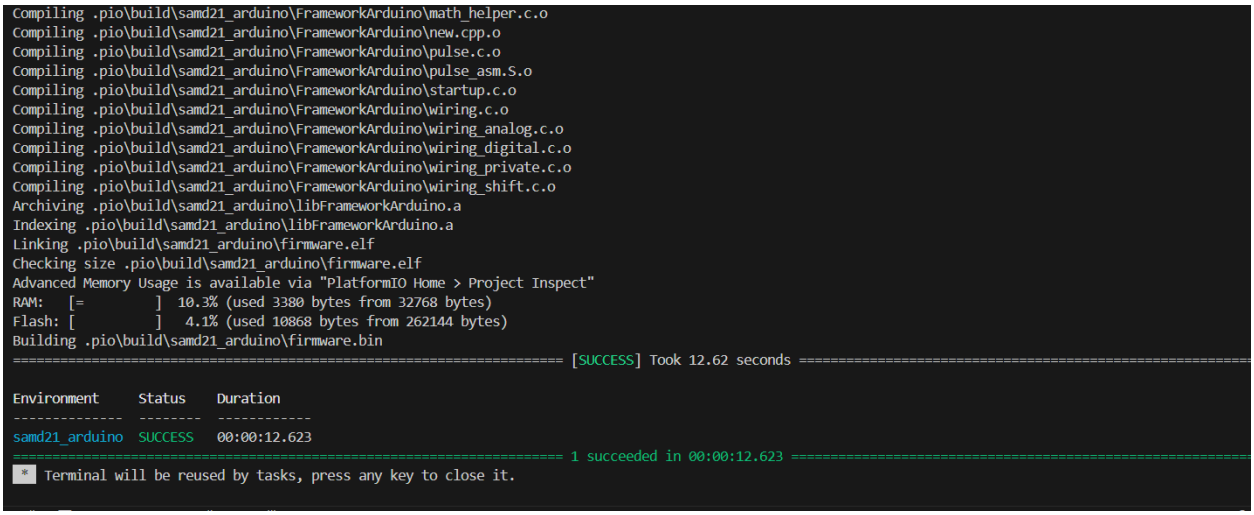


Build Time – 22.365 seconds

RAM = 11.18% (used 3665 bytes from 32768 bytes)

Flash = 4.2% (used 10990 bytes from 262144 bytes)

SAMD21 Build:

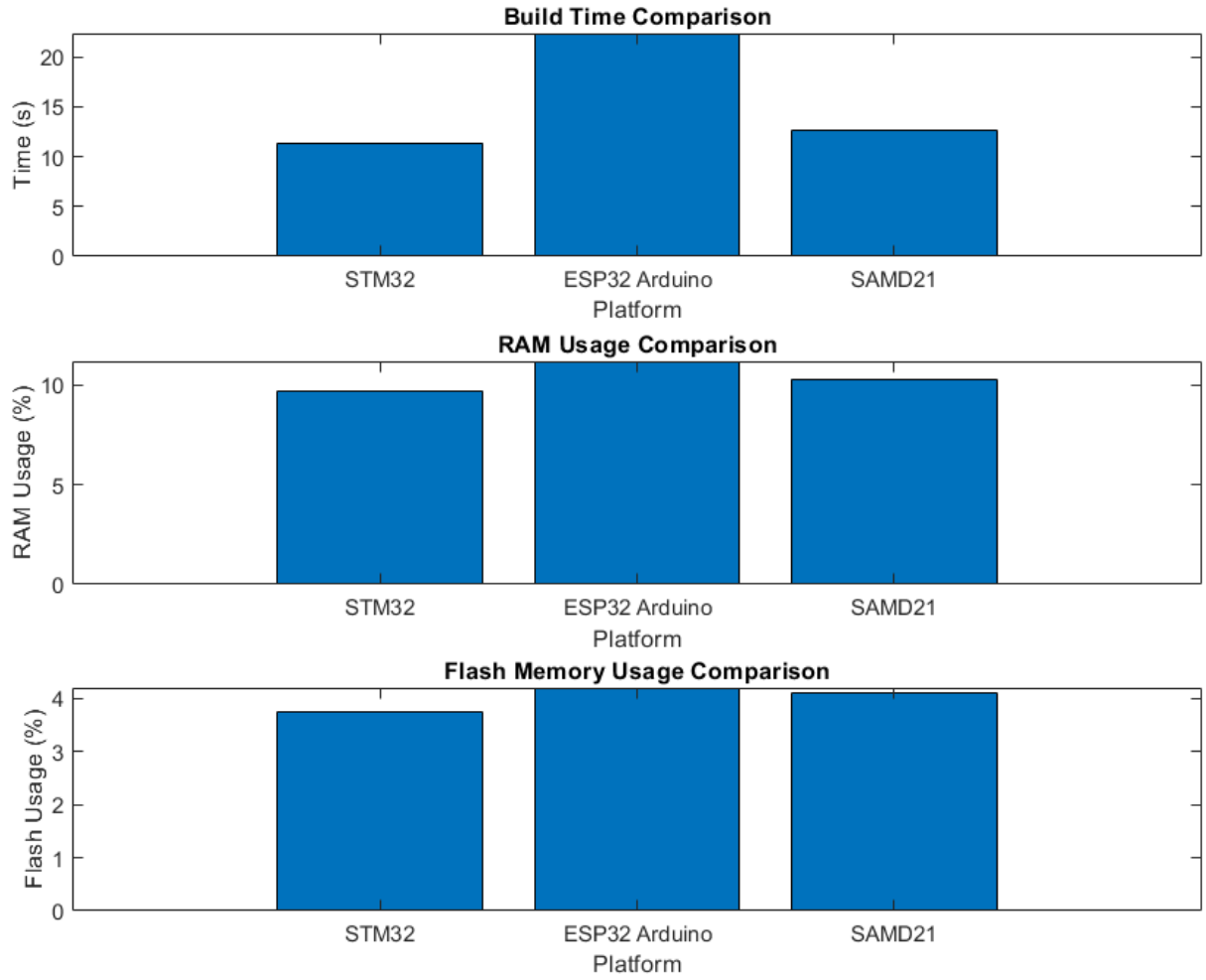


Build Time – 12.623 seconds

RAM = 10.3% (used 3380 bytes from 32768 bytes)

Flash = 4.1% (used 10868 bytes from 262144 bytes)

Comparison Graph for used builds:



In comparing the build times across the STM32, ESP32 Arduino, and SAMD21 platforms, we discern notable differences in the efficiency of their respective development environments and toolchains. The STM32 platform exhibits the shortest build time at 11.4 seconds, indicative of a streamlined and optimized compilation process. This efficiency may stem from optimized compiler settings or a well-tailored development ecosystem. Conversely, the ESP32 Arduino platform showcases the longest build time at 22.365 seconds, suggesting potential complexities or inefficiencies within its compilation workflow. Factors such as extensive libraries or less optimized toolchains could contribute to this prolonged duration. The SAMD21 platform falls between these extremes with a build time of 12.623 seconds, indicating a moderate level of optimization in its development environment. Overall, while the build times offer insights into the performance of each platform's toolchain, they also underscore the importance of optimizing development workflows to minimize iteration cycles and enhance productivity, particularly in resource-constrained embedded systems development environments.