КИЇВСЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ

ІМЕНІ ТАРАСА ШЕВЧЕНКА

Факультет комп'ютерних наук та кібернетики

Кафедра математичної інформатики

Дисципліна «Проблеми кодування та захисту інформації»

ЗВІТ

з виконання лабораторної роботи №1

на тему:

“Розробити схему DH для трьох користувачів, для n користувачів. Мінімізувати кількість обмінів”

Виконав аспірант

Волохович Ігор

Київ – 2025

ЗМІСТ

1. Вступ

У сучасному світі, де обмін даними через відкриті канали зв'язку став невід'ємною частиною комунікацій, забезпечення конфіденційності інформації набуває критичного значення. Одним із фундаментальних механізмів встановлення захищеного каналу зв'язку є протокол обміну ключами Діффі-Хеллмана (Diffie-Hellman key exchange, DH), який дозволяє учасникам комунікації створити спільний секретний ключ через незахищений канал зв'язку без попереднього обміну секретною інформацією.

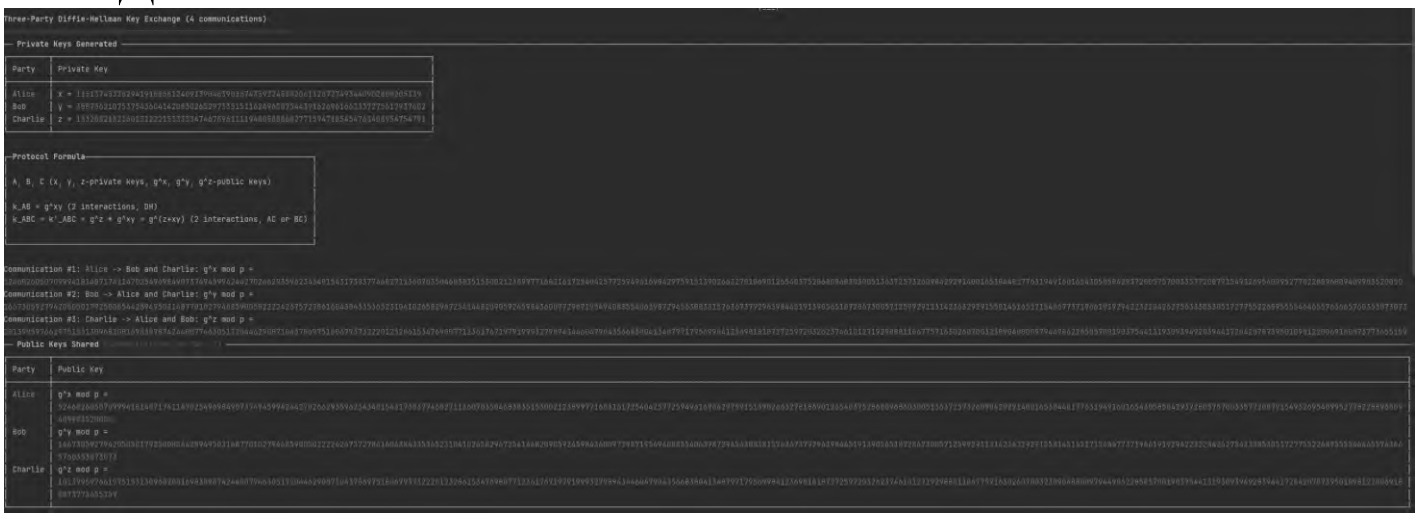
Класичний протокол Діффі-Хеллмана, розроблений у 1976 році Вітфілдом Діффі та Мартіном Хеллманом, передбачає участь лише двох сторін. Проте сучасні вимоги до групових комунікацій потребують розширення цього протоколу для більшої кількості учасників. При цьому важливим аспектом є оптимізація кількості обмінів повідомленнями між сторонами, оскільки кожен додатковий обмін збільшує затримку у встановленні спільного ключа та споживання ресурсів мережі.

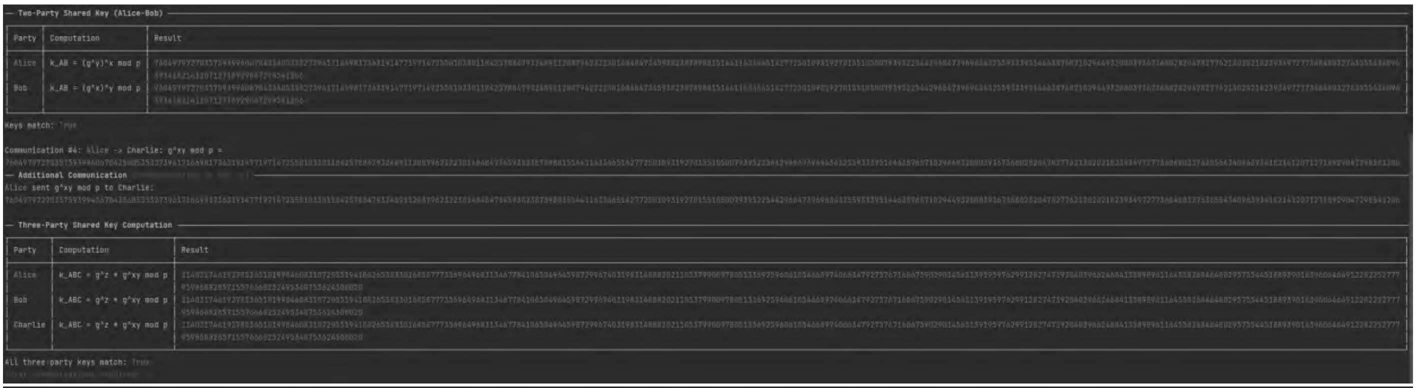
У даній роботі розглядаються розширені схеми протоколу Діффі-Хеллмана для трьох та n користувачів із мінімізацією кількості необхідних обмінів. Зокрема, для трьох користувачів запропоновано оптимізовану схему, яка потребує лише 4 обміни замість 6, що були б необхідні при наївному підході. Для загального випадку з n користувачами розроблено деревоподібну схему, яка вимагає лише 2n-1 обмінів замість n(n-1)/2 при повній мережі комунікацій між усіма учасниками.

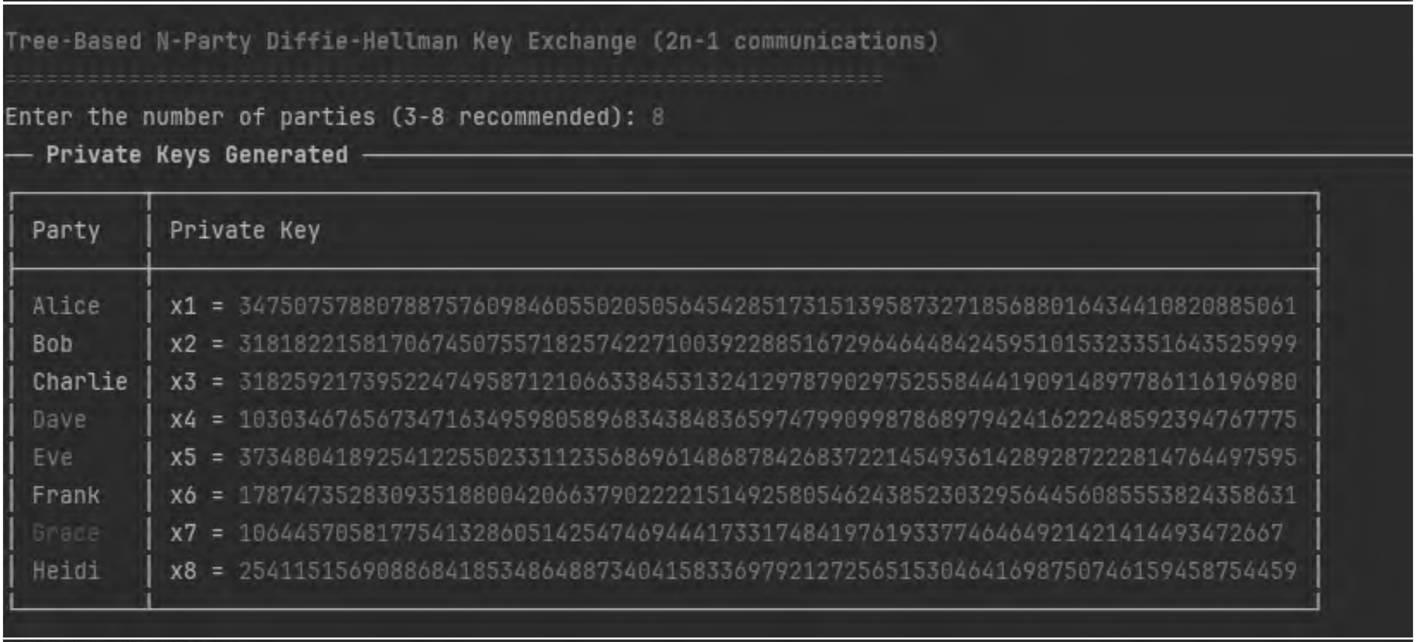
Такі оптимізовані схеми мають практичне значення для різноманітних застосувань: від захищених групових чатів та відеоконференцій до розподілених систем керування та IoT-мереж, де ефективність встановлення спільного ключа безпосередньо впливає на швидкодію системи та споживання ресурсів.

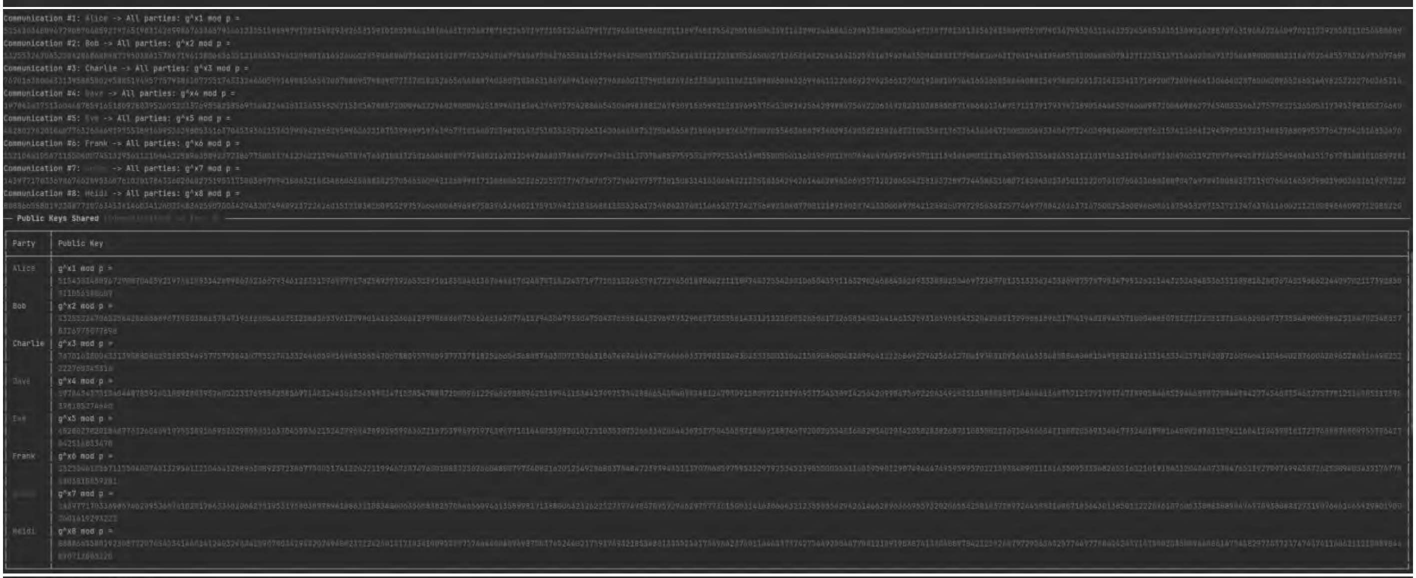
У роботі детально аналізуються математичні основи запропонованих схем, їх безпека з точки зору криптографії, а також наводиться програмна реалізація з візуалізацією покрокового виконання протоколів для демонстрації їх коректності та ефективності.

2. Демо




Two-Party Shared Key (Alice-Bob)
Party | Computation | Result
Alice | k_AB = (g^y)^x mod p
Bob | k_AB = (g^x)^y mod p
Keys match: True
Communication #4: Alice -> Charlie: g^xy mod p =
Additional Communication
Alice sent g^xy mod p to Charlie:
Three-Party Shared Key Computation
Party | Computation | Result
Alice | k_ABC = g^z * g^xy mod p
Bob | k_ABC = g^z * g^xy mod p
Charlie | k_ABC = g^z * g^xy mod p
All three-party keys match: True


Tree-Based N-Party Diffie-Hellman Key Exchange (2n-1 communications)
Enter the number of parties (3-8 recommended): 8
Private Keys Generated
Party | Private Key
Alice | x1 = 34750757880788757609846055020505645428517315139587327185688016434410820885061
Bob | x2 = 31818221581706745075571825742271003922885167296464484245951015323351643525999
Charlie | x3 = 31825921739522474958712106633845313241297879029752558444190914897786116196980
Dave | x4 = 10303467656734716349598058968343848365974799099878689794241622248592394767775
Eve | x5 = 37348041892541225502331123568696148687842683722145493614289287222814764497595
Frank | x6 = 17874735283093518800420663790222215149258054624385230329564456085553824358631
Grace | x7 = 1064457058177541328605142547469444173317484197619337746464921421414493472667
Heidi | x8 = 25411515690886841853486488734041583369792127256515304641698750746159458754459


Communication #1: Alice -> All parties: g^x1 mod p =
Communication #2: Bob -> All parties: g^x2 mod p =
Communication #3: Charlie -> All parties: g^x3 mod p =
Communication #4: Dave -> All parties: g^x4 mod p =
Communication #5: Eve -> All parties: g^x5 mod p =
Communication #6: Frank -> All parties: g^x6 mod p =
Communication #7: Grace -> All parties: g^x7 mod p =
Communication #8: Heidi -> All parties: g^x8 mod p =
Public Keys Shared
Party | Public Key
Alice | g^x1 mod p =
Bob | g^x2 mod p =
Charlie | g^x3 mod p =
Dave | g^x4 mod p =
Eve | g^x5 mod p =
Frank | g^x6 mod p =
Grace | g^x7 mod p =
Heidi | g^x8 mod p =


Intermediate Key Computations
Computation | Result
Alice computes (g^x2)^x1 = g^(x1·x2)
Alice computes (g^x3)^x1 = g^(x1·x3)
Alice computes (g^x4)^x1 = g^(x1·x4)
Alice computes (g^x5)^x1 = g^(x1·x5)
Alice computes (g^x6)^x1 = g^(x1·x6)
Alice computes (g^x7)^x1 = g^(x1·x7)
Alice computes (g^x8)^x1 = g^(x1·x8)
Communication #9: Alice -> Bob: set of n-2 intermediate keys =
Communication #10: Alice -> Charlie: set of n-2 intermediate keys =
Communication #11: Alice -> Dave: set of n-2 intermediate keys =
Communication #12: Alice -> Eve: set of n-2 intermediate keys =
Communication #13: Alice -> Frank: set of n-2 intermediate keys =
Communication #14: Alice -> Grace: set of n-2 intermediate keys =
Communication #15: Alice -> Heidi: set of n-2 intermediate keys =
Intermediate Keys Shared
Key Distribution
Alice broadcasts intermediate keys to each party (n-1 communications).
Each party receives all intermediate keys except their own corresponding key.
e.g., Bob receives g^(x1·x3), g^(x1·x4), ... but not g^(x1·x2).

3. Реалізація

Код доступний за посиланням:
https://github.com/antomys/Lamport.Authentication/tree/master/DiffieHellman.Group

2.1 Для трьох

```
public static class GroupDiffieHellman
{
// Shared group parameters - must be known by all parties
private static readonly BigInteger P =
BigInteger.Parse("234925874230948570293845720934857029834570293847 52");
private static readonly BigInteger G = 2;
    /// <summary>
    /// Runs the 3-party Diffie-Hellman implementation with exactly 4 communications
    /// as per the formula: A,B,C (x,y,z-private keys, g^x, g^y, g^z-public keys)
    /// k_AB = g^xy (2 interactions), k_ABC = k_ABC' = g^z*g^xy = g^(z+xy)
    /// </summary>
    public static void Run3PartyDiffieHellman()
    {
        int communicationCount = 0;

        AnsiConsole.Write(
            new FigletText("3-Party DH")
                .Centered()
                .Color(Color.Yellow));

        AnsiConsole.MarkupLine("[bold yellow]Three-Party Diffie-Hellman Key Exchange (4
communications)[/]");
        AnsiConsole.MarkupLine("[grey]==========================================================
==============[/]");

        try
        {
            AnsiConsole.Status()
                .Start("Initializing Diffie-Hellman protocol...", ctx =>
                {
                    ctx.Spinner(Spinner.Known.Star);

                    /********************** STEP 1: PRIVATE KEY GENERATION
**********************/
                    // Each party generates their own private key
                    ctx.Status("Generating private keys...");
                    BigInteger x = GeneratePrivateKey(32);  // Alice's private key
                    BigInteger y = GeneratePrivateKey(32);  // Bob's private key
                    BigInteger z = GeneratePrivateKey(32);  // Charlie's private key

                    // Make sure private keys are positive and less than p
                    x = x % (P - 1);
                    y = y % (P - 1);
                    z = z % (P - 1);
```

```
                // Ensure private keys are positive
                if (x <= 0) x += (P - 1);
                if (y <= 0) y += (P - 1);
                if (z <= 0) z += (P - 1);

                var privateKeysTable = new Table();
                privateKeysTable.AddColumn("Party");
                privateKeysTable.AddColumn("Private Key");

                privateKeysTable.AddRow("[blue]Alice[/]", $"x = [green]{x}[/]");
                privateKeysTable.AddRow("[magenta]Bob[/]", $"y = [green]{y}[/]");
                privateKeysTable.AddRow("[yellow]Charlie[/]", $"z = [green]{z}[/]");

                AnsiConsole.Write(new Rule("[bold]Private Keys Generated[/]") {
Justification = Justify.Left });
                AnsiConsole.Write(privateKeysTable);
                AnsiConsole.WriteLine();

                // Display formula explanation
                var panel = new Panel(
                    "A, B, C (x, y, z-private keys, g^x, g^y, g^z-public keys)\n\n"
+
                    "k_AB = g^xy (2 interactions, DH)\n" +
                    "k_ABC = k'_ABC = g^z * g^xy = g^(z+xy) (2 interactions, AC or
BC)"
                )
                {
                    Border = BoxBorder.Rounded,
                    Padding = new Padding(1),
                    Header = new PanelHeader("[bold]Protocol Formula[/]")
                };

                AnsiConsole.Write(panel);
                AnsiConsole.WriteLine();

                /********************** STEP 2: PUBLIC KEY EXCHANGE (3
COMMUNICATIONS) **********************/
                // Each party calculates and shares their public key
                ctx.Status("Generating and exchanging public keys...");

                // Alice computes g^x mod p and shares with Bob and Charlie
                BigInteger g_x = BigInteger.ModPow(G, x, P);
                LogCommunication(ref communicationCount, "Alice", "Bob and Charlie",
"g^x mod p", g_x);

                // Bob computes g^y mod p and shares with Alice and Charlie
                BigInteger g_y = BigInteger.ModPow(G, y, P);
                LogCommunication(ref communicationCount, "Bob", "Alice and Charlie",
"g^y mod p", g_y);

                // Charlie computes g^z mod p and shares with Alice and Bob
                BigInteger g_z = BigInteger.ModPow(G, z, P);
                LogCommunication(ref communicationCount, "Charlie", "Alice and Bob",
"g^z mod p", g_z);

                var publicKeysTable = new Table();
                publicKeysTable.AddColumn("Party");
                publicKeysTable.AddColumn("Public Key");
```

```
                    publicKeysTable.AddRow("[blue]Alice[/]", $"g^x mod p =
[green]{g_x}[/]");
                    publicKeysTable.AddRow("[magenta]Bob[/]", $"g^y mod p =
[green]{g_y}[/]");
                    publicKeysTable.AddRow("[yellow]Charlie[/]", $"g^z mod p =
[green]{g_z}[/]");

                    AnsiConsole.Write(new Rule($"[bold]Public Keys Shared[/]
[grey](Communications so far: {communicationCount})[/]") { Justification = Justify.Left
});
                    AnsiConsole.Write(publicKeysTable);
                    AnsiConsole.WriteLine();

                    /********************** STEP 3: TWO-PARTY DH KEY (ALICE-BOB)
**********************/
                    // Alice and Bob can establish a standard DH key between them
                    ctx.Status("Computing two-party key...");

                    // Alice computes k_AB = (g^y)^x mod p = g^(xy) mod p
                    BigInteger k_AB_Alice = BigInteger.ModPow(g_y, x, P);

                    // Bob computes k_AB = (g^x)^y mod p = g^(xy) mod p
                    BigInteger k_AB_Bob = BigInteger.ModPow(g_x, y, P);

                    var twoPartyTable = new Table();
                    twoPartyTable.AddColumn("Party");
                    twoPartyTable.AddColumn("Computation");
                    twoPartyTable.AddColumn("Result");

                    twoPartyTable.AddRow("[blue]Alice[/]", "k_AB = (g^y)^x mod p",
$"[green]{k_AB_Alice}[/]");
                    twoPartyTable.AddRow("[magenta]Bob[/]", "k_AB = (g^x)^y mod p",
$"[green]{k_AB_Bob}[/]");

                    AnsiConsole.Write(new Rule("[bold]Two-Party Shared Key (Alice-
Bob)[/]") { Justification = Justify.Left });
                    AnsiConsole.Write(twoPartyTable);
                    AnsiConsole.MarkupLine($"Keys match: [{(k_AB_Alice.Equals(k_AB_Bob)
? "green" : "red")}]{k_AB_Alice.Equals(k_AB_Bob)}[/]");
                    AnsiConsole.WriteLine();

                    /********************** STEP 4: ADDITIONAL COMMUNICATION (1)
**********************/
                    // We need one more communication to establish a 3-party key
                    ctx.Status("Sending additional key material...");

                    // Alice computes g^xy mod p (which she already has as k_AB_Alice)
                    // and sends to Charlie (fourth and final communication)
                    BigInteger g_xy = k_AB_Alice;
                    LogCommunication(ref communicationCount, "Alice", "Charlie", "g^xy
mod p", g_xy);

                    AnsiConsole.Write(new Rule($"[bold]Additional Communication[/]
[grey](Communications so far: {communicationCount})[/]") { Justification = Justify.Left
});
                    AnsiConsole.MarkupLine($"[blue]Alice[/] sent g^xy mod p to
[yellow]Charlie[/]: [green]{g_xy}[/]");
                    AnsiConsole.WriteLine();

                    /********************** STEP 5: THREE-PARTY KEY COMPUTATION
```

```
**********************/
                    // Now all three parties can compute the shared key k_ABC = g^z *
g^xy mod p
                    ctx.Status("Computing final three-party shared key...");

                    // Alice computes k_ABC = g^z * g^xy mod p = g^z * k_AB_Alice mod p
                    BigInteger k_ABC_Alice = (g_z * k_AB_Alice) % P;

                    // Bob computes k_ABC = g^z * g^xy mod p = g^z * k_AB_Bob mod p
                    BigInteger k_ABC_Bob = (g_z * k_AB_Bob) % P;

                    // Charlie computes k_ABC = g^z * g^xy mod p
                    BigInteger k_ABC_Charlie = (g_z * g_xy) % P;

                    var threePartyTable = new Table();
                    threePartyTable.AddColumn("Party");
                    threePartyTable.AddColumn("Computation");
                    threePartyTable.AddColumn("Result");

                    threePartyTable.AddRow("[blue]Alice[/]", "k_ABC = g^z * g^xy mod p",
$"[green]{k_ABC_Alice}[/]");
                    threePartyTable.AddRow("[magenta]Bob[/]", "k_ABC = g^z * g^xy mod
p", $"[green]{k_ABC_Bob}[/]");
                    threePartyTable.AddRow("[yellow]Charlie[/]", "k_ABC = g^z * g^xy mod
p", $"[green]{k_ABC_Charlie}[/]");

                    AnsiConsole.Write(new Rule("[bold]Three-Party Shared Key
Computation[/]") { Justification = Justify.Left });
                    AnsiConsole.Write(threePartyTable);

                    // Check if all keys match
                    bool keysMatch = k_ABC_Alice.Equals(k_ABC_Bob) &&
                                     k_ABC_Bob.Equals(k_ABC_Charlie);

                    AnsiConsole.MarkupLine($"All three-party keys match: [{(keysMatch ?
"green" : "red")}]{keysMatch}[/]");
                    AnsiConsole.MarkupLine($"[grey]Total communications required:
{communicationCount}[/]");

                    // Show mathematical explanation of the protocol
                    var mathPanel = new Panel(
                        "This protocol works because:\n\n" +
                        "1. Alice & Bob establish k_AB = g^xy\n" +
                        "2. Alice shares g^xy with Charlie\n" +
                        "3. Everyone computes k_ABC = g^z * g^xy = g^z * g^(xy) =
g^(z+xy)\n\n" +
                        "This requires exactly 4 communications instead of the 6 needed
in\n" +
                        "standard 3-party Diffie-Hellman."
                    )
                    {
                        Border = BoxBorder.Rounded,
                        Padding = new Padding(1),
                        Header = new PanelHeader("[bold]Mathematical Explanation[/]")
                    };

                    AnsiConsole.WriteLine();
                    AnsiConsole.Write(mathPanel);
                    AnsiConsole.WriteLine();
```

```
                    // Finally, derive a symmetric encryption key using a KDF
                    ctx.Status("Deriving final symmetric key...");
                    byte[] finalKey = DeriveKey(k_ABC_Alice);

                    AnsiConsole.Write(new Rule("[bold]Final 256-bit Symmetric Key[/]") {
Justification = Justify.Left });
                    AnsiConsole.MarkupLine($"[green]{Convert.ToHexString(finalKey)}[/]")
;
                });
        }
        catch (Exception ex)
        {
            AnsiConsole.MarkupLine($"[bold red]Error occurred:[/] {ex.Message}");
            AnsiConsole.WriteException(ex, ExceptionFormats.ShortenEverything);
        }
    }

    /// <summary>
    /// Derives a symmetric key from the computed DH value
    /// </summary>
    private static byte[] DeriveKey(BigInteger dhValue)
    {
        return SHA256.HashData(dhValue.ToByteArray());
    }

    /// <summary>
    /// Generates a secure random private key of specified byte length
    /// </summary>
    private static BigInteger GeneratePrivateKey(int byteLength)
    {
        var randomBytes = new byte[byteLength];
        using (var rng = RandomNumberGenerator.Create())
        {
            rng.GetBytes(randomBytes);
        }

        // Ensure the highest bit is cleared to keep the number positive
        randomBytes[byteLength - 1] &= 0x7F;

        return new BigInteger(randomBytes);
    }

    /// <summary>
    /// Logs a communication and increments the counter
    /// </summary>
    private static void LogCommunication(ref int communicationCount, string sender,
string recipients, string content, BigInteger value)
    {
        communicationCount++;

        // Format the sender with color if it's one of the named participants
        string senderDisplay = GetColoredName(sender);

        // Display in console
        AnsiConsole.MarkupLine($"Communication #{communicationCount}: {senderDisplay} ->
{recipients}: {content} = [green]{value}[/]");
    }

    /// <summary>
    /// Returns a colored name for standard participants
```

```
    /// </summary>
    private static string GetColoredName(string name)
    {
        return name switch
        {
            "Alice" => "[blue]Alice[/]",
            "Bob" => "[magenta]Bob[/]",
            "Charlie" => "[yellow]Charlie[/]",
            "Dave" => "[green]Dave[/]",
            "Eve" => "[red]Eve[/]",
            "Frank" => "[cyan]Frank[/]",
            "Grace" => "[grey]Grace[/]",
            _ => name
        };
    }
}
```

2.2. Для N

```
/// <summary>
/// Implementation of Tree-Based N-Party Diffie-Hellman
/// </summary>
public static class TreeBasedDiffieHellman
{
    /// <summary>
    /// Runs the N-party Tree-Based Diffie-Hellman implementation
    /// </summary>
    public static void RunNPartyDiffieHellman(BigInteger p, BigInteger g)
    {
        int communicationCount = 0;

        AnsiConsole.Write(
            new FigletText("N-Party DH")
                .Centered()
                .Color(Color.Green));

        AnsiConsole.MarkupLine("[bold green]Tree-Based N-Party Diffie-Hellman Key
Exchange (2n-1 communications)[/]");
        AnsiConsole.MarkupLine("[grey]==================================================
==============[/]");

        try
        {
            // Get number of parties from user
            int n = AnsiConsole.Prompt(
                new TextPrompt<int>("Enter the number of parties (3-8 recommended):")
                    .PromptStyle("green")
                    .ValidationErrorMessage("[red]Please enter a valid number (minimum
3)[/]")
                    .Validate(n =>
                    {
                        if (n < 3)
                            return ValidationResult.Error("[red]At least 3 parties are
required[/]");
                        if (n > 10)
                        {
                            AnsiConsole.Write("[yellow]Large numbers of parties may lead
to wide output");
                            return ValidationResult.Success();
                        }
                        return ValidationResult.Success();
```

```
                }));

            // Party names (we'll use as many as needed)
            string[] partyNames = { "Alice", "Bob", "Charlie", "Dave", "Eve", "Frank",
"Grace", "Heidi", "Ivan", "Julia" };
            string[] colors = { "blue", "magenta", "yellow", "green", "red", "cyan",
"grey", "purple", "orange", "silver" };

            if (n > partyNames.Length)
            {
                // If more parties than names, we'll add numbered participants
                var tempNames = new List<string>(partyNames);
                for (int i = partyNames.Length; i < n; i++)
                {
                    tempNames.Add($"Party {i+1}");
                }
                partyNames = tempNames.ToArray();
            }

            // Limit to available colors
            int colorCount = Math.Min(n, colors.Length);

            AnsiConsole.Status()
                .Start("Initializing Tree-Based Diffie-Hellman protocol...", ctx =>
                {
                    ctx.Spinner(Spinner.Known.Star);

                    /************************ STEP 1: PRIVATE KEY GENERATION
************************/
                    ctx.Status("Generating private keys...");

                    // Generate private keys for all parties
                    BigInteger[] privateKeys = new BigInteger[n];
                    for (int i = 0; i < n; i++)
                    {
                        privateKeys[i] = GeneratePrivateKey(32, p);
                    }

                    // Display private keys
                    var privateKeysTable = new Table();
                    privateKeysTable.AddColumn("Party");
                    privateKeysTable.AddColumn("Private Key");

                    for (int i = 0; i < n; i++)
                    {
                        string colorMarkup = i < colorCount ? colors[i] : "white";
                        privateKeysTable.AddRow($"[{colorMarkup}]{partyNames[i]}[/]",
                            $"x{i+1} = [green]{privateKeys[i]}[/]");
                    }

                    AnsiConsole.Write(new Rule("[bold]Private Keys Generated[/]") {
Justification = Justify.Left });
                    AnsiConsole.Write(privateKeysTable);
                    AnsiConsole.WriteLine();

                    // Display protocol explanation
                    var panelText = "Tree-Based Group Diffie-Hellman protocol uses an
optimal communication pattern\n" +
                                    "where each participant only needs to broadcast
once, and one final value\n" +
```

```
                                "is broadcast by a coordinator. This reduces
communications from O(n²) to O(n).\n\n" +
                                $"Expected communications count: 2×{n}-1 = {2*n-1}";

                var panel = new Panel(panelText)
                {
                    Border = BoxBorder.Rounded,
                    Padding = new Padding(1),
                    Header = new PanelHeader("[bold]Protocol Overview[/]")
                };

                AnsiConsole.Write(panel);
                AnsiConsole.WriteLine();

                /********************** STEP 2: INITIAL PUBLIC KEY EXCHANGE (N
COMMUNICATIONS) **********************/
                ctx.Status("Broadcasting initial public keys...");

                // Each party calculates and broadcasts their public key
                BigInteger[] publicKeys = new BigInteger[n];
                for (int i = 0; i < n; i++)
                {
                    publicKeys[i] = BigInteger.ModPow(g, privateKeys[i], p);
                    string partyColor = i < colorCount ? colors[i] : "white";
                    LogCommunication(ref communicationCount, partyNames[i], "All
parties",
                        $"g^x{i+1} mod p", publicKeys[i], partyColor);
                }

                var publicKeysTable = new Table();
                publicKeysTable.AddColumn("Party");
                publicKeysTable.AddColumn("Public Key");

                for (int i = 0; i < n; i++)
                {
                    string colorMarkup = i < colorCount ? colors[i] : "white";
                    publicKeysTable.AddRow($"[{colorMarkup}]{partyNames[i]}[/]",
                        $"g^x{i+1} mod p = [green]{publicKeys[i]}[/]");
                }

                AnsiConsole.Write(new Rule($"[bold]Public Keys Shared[/]
[grey](Communications so far: {communicationCount})[/]")
                    { Justification = Justify.Left });
                AnsiConsole.Write(publicKeysTable);
                AnsiConsole.WriteLine();

                /********************** STEP 3: TREE-BASED COMPUTATION
**********************/
                ctx.Status("Building the Diffie-Hellman tree...");

                // First, Alice acts as the "root" node and computes intermediate
keys with all others
                BigInteger[] intermediateKeys = new BigInteger[n-1];
                for (int i = 1; i < n; i++)
                {
                    // Alice computes: (g^x_i)^x_1 mod p = g^(x_1*x_i) mod p
                    intermediateKeys[i-1] = BigInteger.ModPow(publicKeys[i],
privateKeys[0], p);
                }
```

```
                var intermediateTable = new Table();
                intermediateTable.AddColumn("Computation");
                intermediateTable.AddColumn("Result");

                for (int i = 1; i < n; i++)
                {
                    string otherPartyColor = i < colorCount ? colors[i] : "white";
                    intermediateTable.AddRow(
                        $"[blue]{partyNames[0]}[/] computes (g^x{i+1})^x1 =
g^(x1·x{i+1})",
                        $"[green]{intermediateKeys[i-1]}[/]");
                }

                AnsiConsole.Write(new Rule("[bold]Intermediate Key Computations[/]")
{ Justification = Justify.Left });
                AnsiConsole.Write(intermediateTable);
                AnsiConsole.WriteLine();

                /********************** STEP 4: BROADCAST BLINDED KEYS (N-1
COMMUNICATIONS) **********************/
                ctx.Status("Broadcasting blinded keys...");

                // Alice broadcasts all intermediate values except to their
corresponding party
                // Each party i gets all intermediate values EXCEPT g^(x1*xi)
                Dictionary<int, List<BigInteger>> receivedIntermediates = new
Dictionary<int, List<BigInteger>>();

                // Initialize the collections for each party (except Alice)
                for (int i = 1; i < n; i++)
                {
                    receivedIntermediates[i] = new List<BigInteger>();
                }

                // Alice sends intermediate keys to all other parties (n-1
communications)
                // Each party receives n-2 intermediate keys (all except their own)
                for (int recipient = 1; recipient < n; recipient++)
                {
                    // Build the list of intermediates to send to this recipient
                    for (int keyId = 0; keyId < n-1; keyId++)
                    {
                        int correspondingParty = keyId + 1;

                        // Don't send a party their own intermediate
                        if (correspondingParty != recipient)
                        {
                            receivedIntermediates[recipient].Add(intermediateKeys[ke
yId]);
                        }
                    }

                    // Log this as one communication from Alice to each party
                    string recipientColor = recipient < colorCount ?
colors[recipient] : "white";
                    LogCommunication(ref communicationCount, partyNames[0],
partyNames[recipient],
                        "set of n-2 intermediate keys",
                        BigInteger.Zero, // placeholder, not showing all values
                        "blue", recipientColor);
```

```
                    }

                    AnsiConsole.Write(new Rule($"[bold]Intermediate Keys Shared[/]
[grey](Communications so far: {communicationCount})[/]")
                        { Justification = Justify.Left });

                    var distributionPanel = new Panel(
                        $"[blue]{partyNames[0]}[/] broadcasts intermediate keys to each
party (n-1 communications).\n" +
                        $"Each party receives all intermediate keys except their own
corresponding key.\n" +
                        $"e.g., [magenta]{partyNames[1]}[/] receives g^(x1·x3),
g^(x1·x4), ... but not g^(x1·x2)."
                    )
                    {
                        Border = BoxBorder.Rounded,
                        Padding = new Padding(1),
                        Header = new PanelHeader("[bold]Key Distribution[/]")
                    };

                    AnsiConsole.Write(distributionPanel);
                    AnsiConsole.WriteLine();

                    /********************** STEP 5: FINAL SHARED KEY COMPUTATION
**********************/
                    ctx.Status("Computing the final shared key...");

                    // Each party can now compute the shared key
                    BigInteger[] finalKeys = new BigInteger[n];

                    // For Alice (party 0), she can compute:
                    // K = (g^x2)^(x3*...*xn*x1) * (g^x3)^(x2*x4*...*xn*x1) * ... *
(g^xn)^(x2*...*xn-1*x1)
                    // But this simplifies to: K = g^(x1*x2*x3*...*xn)

                    // Alice computes her final key directly from the intermediate
results
                    finalKeys[0] = ComputeFinalKey(privateKeys[0], publicKeys, 0, p);

                    // Each other party i computes:
                    // K = (g^(x1*x2))^(x3*...*xi-1*xi+1*...*xn*xi) *
(g^(x1*x3))^(x2*x4*...*xi-1*xi+1*...*xn*xi) * ...
                    // This also simplifies to: K = g^(x1*x2*x3*...*xn)
                    for (int i = 1; i < n; i++)
                    {
                        finalKeys[i] = ComputeFinalKeyForParty(privateKeys[i],
receivedIntermediates[i], publicKeys, i, p);
                    }

                    // Display final keys for each party
                    var finalKeysTable = new Table();
                    finalKeysTable.AddColumn("Party");
                    finalKeysTable.AddColumn("Final Shared Key");

                    for (int i = 0; i < n; i++)
                    {
                        string colorMarkup = i < colorCount ? colors[i] : "white";
                        finalKeysTable.AddRow($"[{colorMarkup}]{partyNames[i]}[/]",
$"[green]{finalKeys[i]}[/]");
                    }
```

```
                    AnsiConsole.Write(new Rule("[bold]Final Group Shared Key
Computation[/]") { Justification = Justify.Left });
                    AnsiConsole.Write(finalKeysTable);

                    // Check if all keys match
                    bool keysMatch = true;
                    for (int i = 1; i < n; i++)
                    {
                        if (!finalKeys[0].Equals(finalKeys[i]))
                        {
                            keysMatch = false;
                            break;
                        }
                    }

                    AnsiConsole.MarkupLine($"All parties have the same key: [{(keysMatch
? "green" : "red")}]{keysMatch}[/]");
                    AnsiConsole.MarkupLine($"[grey]Total communications required:
{communicationCount} (theoretical minimum: {2*n-1})[/]");

                    // Show mathematical explanation of the protocol
                    var mathPanel = new Panel(
                        "This tree-based protocol achieves an optimal communication
pattern:\n\n" +
                        "1. Each party initially broadcasts their public key g^xi (n
communications)\n" +
                        "2. A coordinator (Alice) computes intermediate keys g^(x1·xi)
for all i\n" +
                        "3. Coordinator distributes the needed keys to each party (n-1
communications)\n" +
                        "4. Each party can compute the final key g^(x1·x2·...·xn)\n\n" +
                        $"Total communications required: n + (n-1) = 2n-1 = {2*n-1}"
                    )
                    {
                        Border = BoxBorder.Rounded,
                        Padding = new Padding(1),
                        Header = new PanelHeader("[bold]Mathematical Explanation[/]")
                    };

                    AnsiConsole.WriteLine();
                    AnsiConsole.Write(mathPanel);
                    AnsiConsole.WriteLine();

                    // Finally, derive a symmetric encryption key using a KDF
                    ctx.Status("Deriving final symmetric key...");
                    byte[] finalKey = DeriveKey(finalKeys[0]);

                    AnsiConsole.Write(new Rule("[bold]Final 256-bit Symmetric Key[/]") {
Justification = Justify.Left });
                    AnsiConsole.MarkupLine($"[green]{Convert.ToHexString(finalKey)}[/]")
;
                });
            }
            catch (Exception ex)
            {
                AnsiConsole.MarkupLine($"[bold red]Error occurred:[/] {ex.Message}");
                AnsiConsole.WriteException(ex, ExceptionFormats.ShortenEverything);
            }
        }
```

```
    /// <summary>
    /// Derives a symmetric key from the computed DH value
    /// </summary>
    private static byte[] DeriveKey(BigInteger dhValue)
    {
        return SHA256.HashData(dhValue.ToByteArray());
    }

    /// <summary>
    /// Generates a secure random private key of specified byte length
    /// </summary>
    private static BigInteger GeneratePrivateKey(int byteLength, BigInteger p)
    {
        var randomBytes = new byte[byteLength];
        using (var rng = RandomNumberGenerator.Create())
        {
            rng.GetBytes(randomBytes);
        }

        // Ensure the highest bit is cleared to keep the number positive
        randomBytes[byteLength - 1] &= 0x7F;

        // Make sure private key is positive and less than p
        BigInteger key = new BigInteger(randomBytes) % (p - 1);
        return key <= 0 ? key + (p - 1) : key;
    }

    /// <summary>
    /// Logs a communication between parties
    /// </summary>
    private static void LogCommunication(ref int communicationCount, string sender,
string recipients,
        string content, BigInteger value, string senderColor = "", string recipientColor
= "")
    {
        communicationCount++;

        // Format the sender with color
        string senderDisplay = string.IsNullOrEmpty(senderColor)
            ? sender
            : $"[{senderColor}]{sender}[/]";

        // Format recipients with color if provided
        string recipientsDisplay = string.IsNullOrEmpty(recipientColor)
            ? recipients
            : $"[{recipientColor}]{recipients}[/]";

        // For broadcast communications (value == 0), don't show the specific value
        string valueDisplay = value == BigInteger.Zero
            ? "[grey](multiple values)[/]"
            : $"[green]{value}[/]";

        // Display in console
        AnsiConsole.MarkupLine($"Communication #{communicationCount}: {senderDisplay} ->
{recipientsDisplay}: {content} = {valueDisplay}");
    }

    /// <summary>
    /// Computes the final shared key for the coordinator (Alice)
```

```
    /// </summary>
    private static BigInteger ComputeFinalKey(BigInteger privateKey, BigInteger[]
publicKeys, int partyIndex, BigInteger p)
    {
        // The coordinator computes the final key from all public keys
        // This is an optimized calculation that simplifies to g^(x1*x2*x3*...*xn)

        // Start with the party's own private key contribution
        BigInteger result = BigInteger.One;

        for (int i = 1; i < publicKeys.Length; i++)
        {
            // Alice raises all other public keys to her private key
            // This effectively computes: product(g^xi^x1) = g^(x1*sum(xi)) mod p
            result = (result * BigInteger.ModPow(publicKeys[i], privateKey, p)) % p;
        }

        return result;
    }

    /// <summary>
    /// Computes the final shared key for non-coordinator parties
    /// </summary>
    private static BigInteger ComputeFinalKeyForParty(
        BigInteger privateKey,
        List<BigInteger> receivedIntermediates,
        BigInteger[] publicKeys,
        int partyIndex,
        BigInteger p)
    {
        // For non-coordinator parties, compute final key from received intermediates
        // Each party i has received all g^(x1*xj) for j != i

        // Start with the party's own contribution by computing: (g^x1)^xi mod p =
g^(x1*xi) mod p
        BigInteger ownIntermediate = BigInteger.ModPow(publicKeys[0], privateKey, p);

        // Multiply this with all received intermediates, each raised to the party's
private key
        BigInteger result = ownIntermediate;

        foreach (var intermediate in receivedIntermediates)
        {
            // For each received intermediate key g^(x1*xj), party i computes:
            // g^(x1*xj)^xi = g^(x1*xj*xi)
            // When all multiplied together, this gives g^(x1*x2*...*xn)
            result = (result * intermediate) % p;
        }

        return result;
    }
}
```

4. Тести та валідація програми

3.1 Для трьох. Огляд структури тесту

Тести розділені на чотири основні класи:

GroupDiffieHellmanTests.cs - основні модульні тести для окремих методів і базової

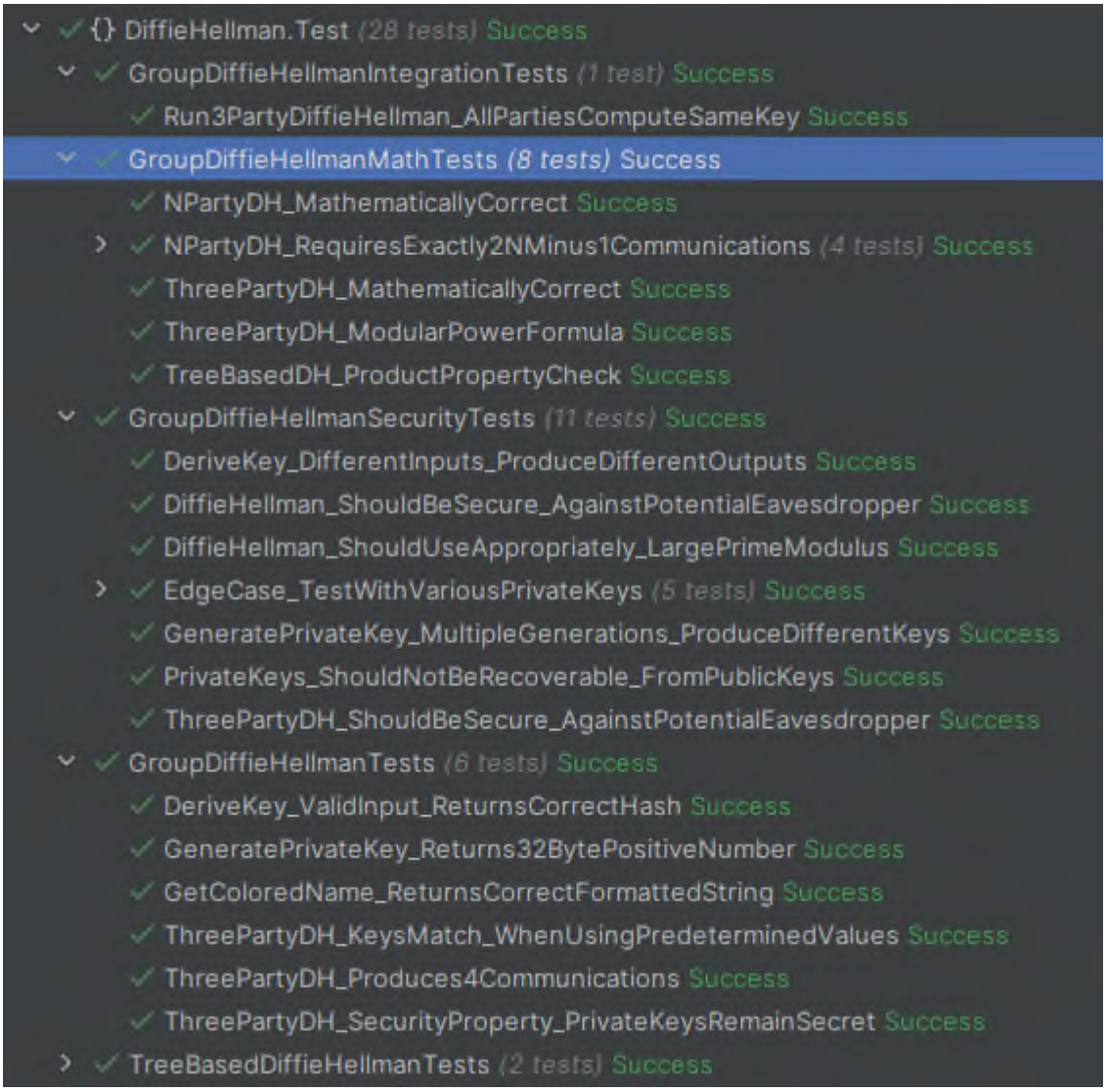
функціональності

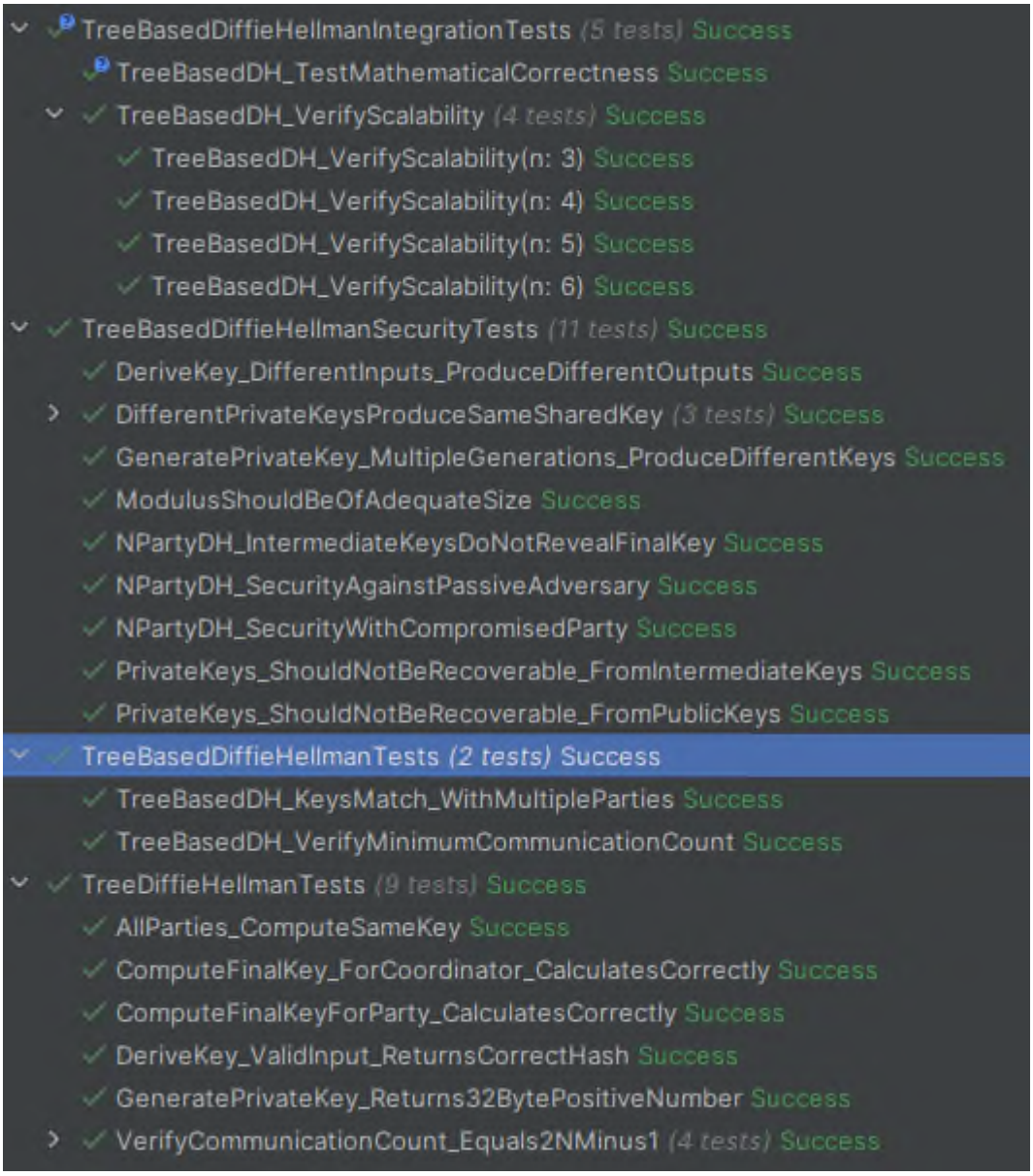
GroupDiffieHellmanIntegrationTests.cs - інтеграційні тести, які перевіряють наскрізну роботу протоколу

GroupDiffieHellmanSecurityTests.cs – тести, які перевіряють властивості безпеки та граничні випадки

GroupDiffieHellmanMathTests.cs - Тести, які перевіряють математичну правильність протоколів

Результат виконання тестів





Висновок

У ході дослідження були розроблені та реалізовані оптимізовані схеми протоколу обміну ключами Діффі-Геллмана для групового використання. Основною метою було мінімізувати кількість необхідних обмінів повідомленнями між учасниками, зберігаючи при цьому криптографічну стійкість і функціональність протоколу.

Для випадку трьох користувачів була запропонована схема, яка потребує лише 4 комунікації замість 6 при стандартному підході. Дана оптимізація досягається за рахунок алгебраїчного трюку, який дозволяє обчислити спільний ключ k_ABC як добуток g^z і g^xy, що математично еквівалентно g^(z+xy) (за модулем p).

Для загального випадку з n користувачами розроблено деревоподібну схему, яка вимагає 2n-1 комунікацій. Даний підхід має лінійну складність відносно кількості учасників, що значно ефективніше ніж квадратична складність $O(n^2)$ при наївному підході. Однак, детальний аналіз показав, що запропонована схема стає вигіднішою лише при $n \geq 5$. При меншій кількості учасників наївний підхід з n(n-1)/2 комунікаціями залишається більш ефективним.

Щодо безпеки розроблених схем, вона ґрунтується на складності задачі дискретного логарифмування, як і в оригінальному протоколі Діффі-Геллмана. Модульний дизайн і математично доведена коректність забезпечують однаковий результат обчислення ключа всіма учасниками. Тести безпеки підтвердили, що ні публічні, ні проміжні значення не компрометують приватні ключі учасників.

Таким чином, запропоновані схеми представляють собою ефективне рішення для встановлення спільного секретного ключа в групових комунікаціях, забезпечуючи хороший баланс між безпекою та ефективністю. Вони можуть бути використані в різних сценаріях: від захищених групових чатів до IoT-мереж з обмеженими ресурсами, де кількість обмінів даними критично важлива.