Protokół wymiany klucza Diffiego-Hellmana oparty na krzywych eliptycznych

Projekt

Prowadzący: **dr. inż Antoni Szczepański, dr inż. Robert Ziemba**

Wykonał:

Imię i nazwisko: **Maciej Kijko**

Nr albumu: **178081**

Rok studiów: **Informatyka II, niestacjonarne, 2024/2025**

Grupa projektowa: **P1**

1. Problem bezpiecznej wymiany klucza

Jednym z najstarszych i najpopularniejszych problemów kryptograficznych jest problem bezpiecznej/prywatnej komunikacji poprzez niezaufany kanał komunikacyjny. Historycznie, pierwsze rozwiązania tego problemu oferowała tzw. Kryptografia Symetryczna (ang. Symmetric cryptography). Kryptosystemy oparte o techniki kryptografii symetrycznej zakładają użycie pojedyńczego klucza do szyfrowania i deszyfrowywania wiadomości przez obie strony komunikacji. Wymagają one więc bezpiecznej wymiany klucza z użyciem innego, bezpiecznego kanału komunikacyjnego lub odpowiedniego protokołu pozwalającego na ustanowienie wspólnego klucza (ang. shared key) poprzez kanał niezaufany.

W latach 70, XX w. Brytyjski kryptolog James H. Ellis ukuł pojęcie “non-secret encryption” dzisiaj nazywane Kryptografią Klucza Publicznego (ang. Public Key Cryptography). Podejście to, w odróżnieniu do kryptografii symetrycznej zakłada użycie pary kluczy szyfrujących: publicznego i prywatnego, dla każdej ze stron komunikacji. Klucz publiczny każdej ze stron może zostać dowolnie udostępniony publicznie bez narażania bezpieczeństwa kryptosystemu. Klucz prywatny natomiast powinien być silnie chroniony i najlepiej nigdy nie przesyłany żadnym z kanałów komunikacji.

Naturalnie, aby kryptosystem klucza publicznego był użyteczny, oba klucze muszą być ze sobą w odpowiedni sposób powiązane. Powiązanie to, poza zapewnieniem możliwości szyfrowania i deszyfrowania wiadomości nie powinno naruszać w sposób znaczący bezpieczeństwa klucza prywatnego. Zrealizowane jest ono zwykle w taki sposób, aby pozyskanie klucza prywatnego z szyfrogramu i klucza publicznego było bardzo kosztowne obliczeniowo (np. Wymagało więcej niż 2^128 operacji).

W dzisiejszych kryptosystemach zwykle stosuje się połączenie obu tych podejść. Kryptografia klucza publicznego pozwala na bezpieczną wymianę klucza szyfrującego, który używany jest potem do zaszyfrowania wiadomości technikami z kryptografii symetrycznej. Natomiast kryptografia asymetryczna nie jest jedynym możliwym podejściem w przypadku wymiany klucza.

2. Protokół wymiany klucza Diffiego-Hellmana

Pośród rozwiązań problemu wymiany klucza, oprócz kryptografii z kluczem publicznym, wyróżniamy również zaproponowany przez Ralpha Merkle system dystrybucji przez klucz publiczny (ang. Public key distribution system) nazywany protokołem wymiany klucza Diffiego-Hellmana (ang. Diffie-Hellman key exchange protocol). Protokół ten, jest bezpieczną, matematyczną metodą **uzgodnienia** klucza szyfrującego poprzez publiczny (niebezpieczny) kanał komunikacyjny. Często odróżnia się protokoły **wymiany** klucza od protokołów **uzgodnienia** klucza. W protokołach uzgadniania klucza, klucz sam w sobie nigdy nie jest przesyłany przez kanał komunikacyjny, a jest przez obie strony ustalany, na podstawie publicznych parametrów.

2.1. Zasada działania

Oryginalnie protokół bazuje na *grupach multiplikatywnych liczb całkowitych modulo p,* gdzie *p* jest liczbą pierwszą.

Przebieg komunikacji:

1. Obie strony ustalają parametry protokołu:
   p – liczba pierwsza
   g – pierwiastek pierwotny modulo *p*

2. Każda ze stron wybiera losowo liczbę *d* z przedziału $\langle 1, p-1 \rangle$. Jest to ich klucz prywatny nigdy nie przesyłany żadnym kanałem komunikacyjnym

3. Strony przesyłają do siebie klucz publiczny *A* wyliczony jak poniżej:

$$A \equiv g^d \bmod p$$

4. Każda ze stron wylicza sekretny klucz *s* jak poniżej:

$$s \equiv A^d \bmod p$$

gdzie *A* jest kluczem publiczym przeciwnej strony, a *d* kluczem prywatnym obliczającego

5. Obie strony kończą komunikację z tą samą wartością *s*

Rozpisując obliczenia dokładniej widać zasadę działania:

$$A_1 \equiv g^{(d_1)} \, mod \; p$$
$$A_2 \equiv g^{(d_2)} \, mod \; p$$

$$s_1 \equiv \left(g^{(d_1)}\right)^{(d_2)} mod \; p$$
$$s_2 \equiv \left(g^{(d_2)}\right)^{(d_1)} mod \; p$$

Z działań na potęgach wiemy, że:

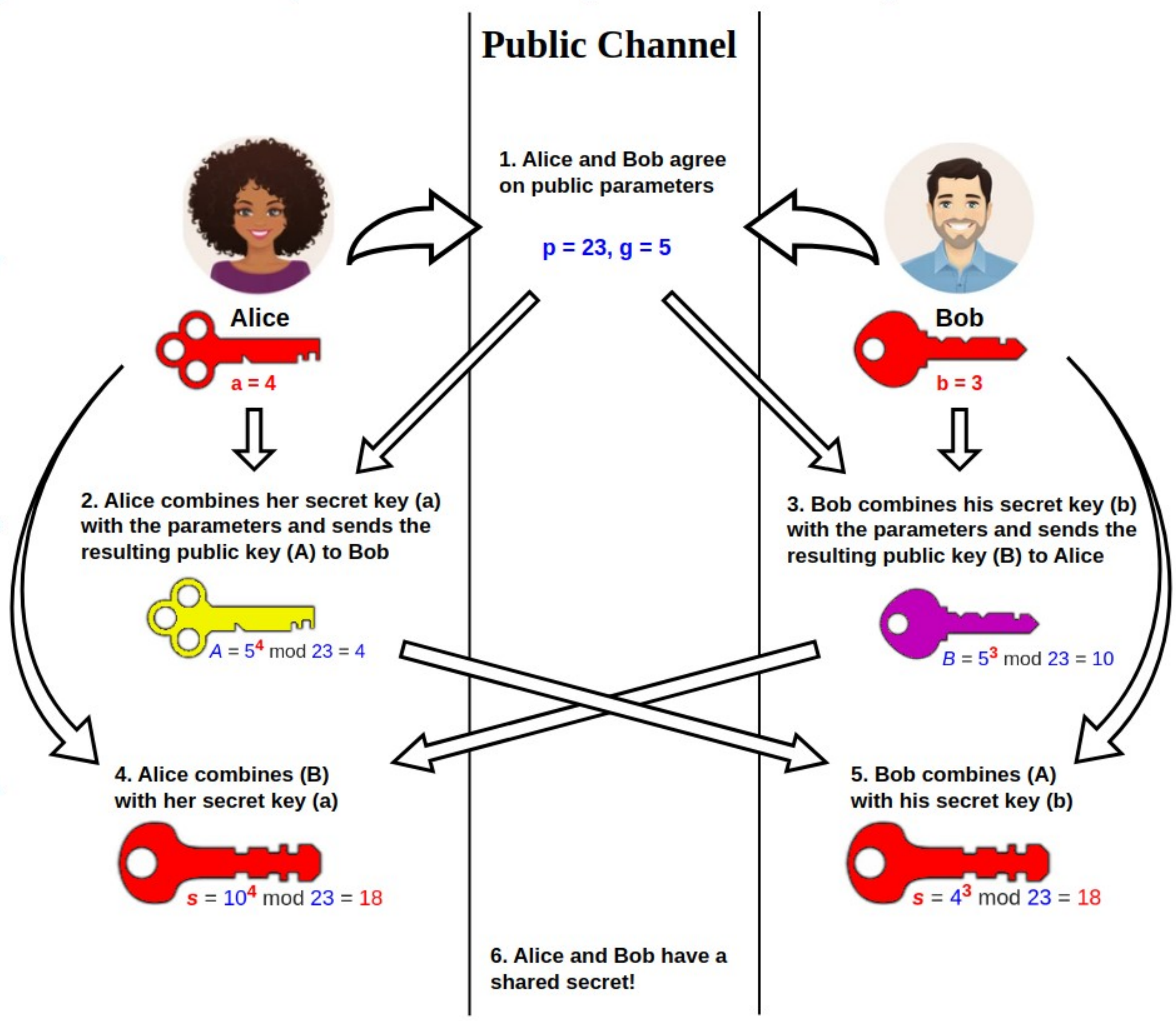
$$\left(g^{(d_1)}\right)^{(d_2)} = g^{(d_1 d_2)}$$

Z przemienności mnożenia wiemy, że:

$$g^{(d_1 d_2)} = g^{(d_2 d_1)}$$

stąd dowodzimy, że *s1 = s2*

Na obrazku poniżej znajduje się przykład DH na małych liczbach:



Źródło: Wikipedia (https://en.wikipedia.org/wiki/Diffie%E2%80%93Hellman_key_exchange#/media/File:DiffieHellman.png)

2.2. Bezpieczeństwo

Bezpieczeństwo wyżej opisanej implementacji protokołu oparte jest na trudności obliczania logarytmu dyskretnego. Nie wynaleziono jeszcze efektywnych metod pozwalających na ich obliczanie. Funkcja obliczająca klucz publiczny zwana jest “one-way-function”. Jest to funkcja, której obliczenie jest zwykle trywialne, ale za to jej “odwrócenie” (tzn. uzyskanie parametrów wejściowych z wyniku i parametrów publicznych) jest kosztowne dla dużych liczb, z uwagi na brak efektywnych metod obliczeniowych. Aktualnie, aby protokół uznawany był za bezpieczny, rekomendowany rozmiar klucza to przynajmniej 2048-bitów, ale zaleca się nawet 3072-bity. Rozmiar klucza w klasycznym DH odpowiada rozmiarowi parametru p. Generator g nie musi być duży i zwykle jest równy *2*, *3* lub *5*.

Warto również dodać, że omawiany protokół (jak i również jego wersja oparta o Krzywe Eliptyczne) nie wspiera uwierzytelnienia drugiej strony komunikacji. W realnym wykorzystaniu więc, potwierdzenie tożsamości rozmówcy zapewniane jest w inny sposób, zwykle poprzez zastosowanie dodatkowych mechanizmów.

3. Implementacja protokołu oparta o Krzywe Eliptyczne

Inną często wykorzystywaną implementacją jest implementacja oparta o Krzywe Eliptyczne (ang. ECDH - Elliptic-Curve Diffie-Hellman key exchange protocol). Występuje ona w dwóch wersjach: wersja z liczbą pierwszą (ang. Prime case) oraz wersja binarna (ang. Binary case). W niniejszym opracowaniu skupiam się jedynie na wersji z liczbą pierwszą.

3.1. Krzywe eliptyczne

Krzywe eliptyczne są strukturami algebraicznymi o genusie równym 1 z wyróżnionym punktem O zwanym "punktem w nieskończoności". Każdą krzywą eliptyczną możemy przedstawić w postaci równania:

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

W kryptosystemach stosowane są zwykle krzywe eliptyczne nad ciałami skończonymi, w opraciu o duże liczby pierwsze. Krzywe te, nie przypominają wtedy krzywych (jak wtedy, gdy są opisane np. Na zbiorze liczb rzeczywistych), a bardziej zbiór punktów na płaszczyźnie na których możemy wykonywać trzy rodzaje operacji: dodawanie dwóch punktów, podwojenie punktu oraz policzenie odwrotności punktu. Punkty te, tworzą grupę abelową, co oznacza że:

- Wynikiem dodawania dwóch punktów znajdujących się na krzywej jest również punkt na tej krzywej
- Dodawanie punktów jest łączne np. $(P+Q)+R=P+(Q+R)$
- Istnieje punkt neutralny O (punkt w nieskończoności) taki, że: $P+O=P$
- Dla każdego punktu P istnieje odwrotność -P, taka że: $P+(-P)=O$
- Dodawanie punktów jest przemienne np. $P+Q=Q+P$

Wszystkie operacje na punktach wykonujemy w arytmetyce modularnej.

3.2. Zasada działania

Przebieg komunikacji:

1. Rozpoczęcie komunikacji w kryptosystemach opartych o krzywe eliptyczne rozpoczyna się od ustalenia publicznych parametrów domenowych *(p, a, b, G, n, h)*, które determinują komunikację. W protokole ECDH ustalanych jest sześć parametrów:

   p **–** liczba pierwsza reprezentująca wielkość pola

   a* oraz *b **–** współczynniki krzywej

   G **–** punkt bazowy dla komunikacji (zwany również generatorem). Punkt ten musi znajdować się na krzywej

   n **–** rząd punktu *G* tzn. Najmniejsza liczba całkowita (zwykle pierwsza), większa od zera, dla której *nG = O* (punkt w nieskończoności). Jest to też rozmiar podgrupy cyklicznej. Większa podgrupa zwiększa bezpieczeństwo, dlatego zwykle punkt *G* dobierany jest tak, aby jego rząd był jak największy.

   h **–** kofaktor. Iloraz liczby wszystkich punktów na krzywej i rzędu punktu bazowego. Dla realnych zastosowań nie powinień przekraczać *4*, lecz im mniejszy, tym lepiej dla bezpieczeństwa komunikacji. *h = 1*, oznacza że podgrupa punktu bazowego zawiera wszystkie punkty na krzywej

2. Każda ze stron losuje swój klucz prywatny *d* z przedziału *<1, n – 1>*

3. Każda ze stron oblicza swój klucz publiczny *Q*, dodając punkt *G* do siebie *d* razy

$$Q_i \equiv d_i G \bmod p$$

4. Następuje wymiana kluczy publicznych *Q1* i *Q2* poprzez kanał komunikacji

5. Każda ze stron oblicza wspólny klucz szyfrujący *s*

$$s \equiv dQ \, mod \; p$$

gdzie *Q* jest kluczem publicznym drugiej strony, a *d* kluczem prywatnym obliczającego

6. Obie strony kończą komunikację z tą samą wartością *s*

Znów, rozpisując obliczenia dokładniej widać zasadę działania:

$$Q_1 \equiv d_1 G \, mod \; p \equiv G + G + G + \ldots mod \; p$$

$$Q_2 \equiv d_2 G \, mod \; p \equiv G + G + G \ldots mod \; p$$

$$s_1 \equiv d_1 Q_2 \, mod \; p \equiv d_1 (d_2 G) \, mod \; p \equiv G + G + G \ldots mod \; p$$

$$s_2 \equiv d_2 Q_1 \, mod \; p \equiv d_2 (d_1 G) \, mod \; p \equiv G + G + G + \ldots mod \; p$$

Jako, że *d1 * d2 = d2 * d1* (z przemienności mnożenia), obie strony dodadzą do siebie punkt bazowy G, tyle samo razy.

Stąd dowodzimy, że *s1 = s2*

Jak widać, ECDH działa bardzo podobnie do DH, z tą różnicą, że zamiast potęgowania pierwiastka pierwotnego modulo p dodajemy punkty na krzywej

3.3. Bezpieczeństwo

Podobnie jak w oryginalnym protokole, bezpieczeństwo ECDH opiera się na obliczeniu logarytmu dyskretnego, mając dany klucz publiczny oraz parametry domenowe.

Ogólnie rzecz biorąc, ECDH względem klasycznego DH uznaje się za bezpieczniejszą oraz praktyczniejszą opcje. ECDH zapewnia porównywalny poziom bezpieczeństwa przy znacznie mniejszym rozmiarze klucza. Klucz 256-bitowy w ECDH, zapewnia porównywalny poziom bezpieczeństwa co klasyczny DH z kluczem 2048-bitowym. Dodatkowo, obliczenia na krzywych eliptycznych są znacznie mniej obciążające.

3.4. Parametry domenowe (ang. Domain parameters)

Parametry domenowe nie są zwykle generowane przez strony komunikacji z uwagi na kosztowność policzenia wszystkich punktów na krzywej. Z tego powodu organizacje standaryzujące tj. NIST (National Institute of Standards and Technology) czy SECG (Standards for Efficient Cryptography Group) publikują rekomendowane parametry domenowe. Poniżej zamieszczam kilka wybranych, z dokumentu “SEC 2: Recommended Elliptic Curve Domain Parameters”. Liczby są zapisane w systemie szesnastkowym. Każda z nich jest wielkości określonej w nazwie zestawu (256, 384, 521)

Nazwa	Parametry
secp256k1 (wielkość 256 bitów)	$p=2^{256}-2^{32}-2^{9}-2^{8}-2^{7}-2^{6}-2^{4}-1$
	$a=0$
	$b=7$
	G = (x=79BE667E F9DCBBAC 55A06295 CE870B07 029BFCDB 2DCE28D9 59F2815B 16F81798, y=483ADA77 26A3C465 5DA4FBFC 0E1108A8 FD17B448 A6855419 9C47D08F FB10D4B8)
	n = FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFE BAAEDCE6 AF48A03B BFD25E8C D0364141
	h = 1
secp384r1 (wielkość 384 bity)	$p=2^{384}-2^{128}-2^{96}+2^{32}-1$
	a = FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFE FFFFFFFF 00000000 00000000 FFFFFFFC
	b = B3312FA7 E23EE7E4 988E056B E3F82D19 181D9C6E FE814112 0314088F 5013875A C656398D 8A2ED19D 2A85C8ED D3EC2AEF
	G = (x=AA87CA22 BE8B0537 8EB1C71E F320AD74 6E1D3B62 8BA79B98 59F741E0 82542A38 5502F25D BF55296C 3A545E38 72760AB7, y=3617DE4A 96262C6F 5D9E98BF 9292DC29 F8F41DBD 289A147C E9DA3113 B5F0B8C0 0A60B1CE 1D7E819D 7A431D7C 90EA0E5F)
	n = FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF C7634D81 F4372DDF 581A0DB2 48B0A77A ECEC196A CCC52973
	h = 1
secp521r1 (wielkość 521 bitów)	$p=2^{521}-1$
	a = 01FF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFC
	b = 0051 953EB961 8E1C9A1F 929A21A0 B68540EE A2DA725B 99B315F3 B8B48991 8EF109E1 56193951 EC7E937B 1652C0BD 3BB1BF07 3573DF88 3D2C34F1 EF451FD4 6B503F00
	G = (x=00C6858E 06B70404 E9CD9E3E CB662395 B4429C64 8139053F B521F828 AF606B4D 3DBAA14B 5E77EFE7 5928FE1D C127A2FF A8DE3348 B3C1856A 429BF97E 7E31C2E5 BD66, y=0118 39296A78 9A3BC004 5C8A5FB4 2C7D1BD9 98F54449 579B4468 17AFBD17 273E662C 97EE7299 5EF42640 C550B901 3FAD0761 353C7086 A272C240 88BE9476 9FD16650)
	n = 01FF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFA 51868783 BF2F966B 7FCC0148 F709A5D0 3BB5C9B8

	899C47AE BB6FB71E 91386409
	h = 1

3.5. Przykładowa implementacja ECDH w języku Java

Poniżej omawiam fragmenty prostej, autorskiej implementacji protokołu ECDH, napisaną w języku Java, bez użycia bibliotek zewnętrznych. Nie sprawdzi się ona oczywiście w realnym wykorzystaniu z uwagi na wiele uproszczeń tj.:

- Oparcie obliczeń na 64 bitowym typie *long*.
- Pomimo, że przykład opiera się na standardowym, 256-bitowym zestawie parametrów domenowych: **secp256k1**, zmniejszyłem rozmiar *p* oraz dobrałem nowy, mniejszy punkt bazowy co za tym idzie, zmiejszył się również rząd *n*.
- Brak sprawdzenia poprawności parametrów na początku oraz w trakcie komunikacji, co jest zalecane nawet przy użyciu standardowych krzywych

Kompletny kod wraz z instrukcją uruchomienia można znaleźć w moim publicznym repozytorium: https://github.com/kijko/elliptic-curve-diffie-hellman-ke-example

3.5.1. Parametry domenowe

Klasa: pl.edu.prz.kijko.ECDHExample.DomainParameters

```
public static class DomainParameters {
     // secp256k1
     public static final long a = 0;
     public static final long b = 7;
     public static final long p = 17;
     public static final Point G = new Point(6, 11);
     public static final long orderG = 18;
     public static final long countPoints = 18;
     public static final int cofactor = (int)
     (countPoints/orderG);
}
```

3.5.2. Główna logika komunikacji

Klasa: pl.edu.prz.kijko.ECDHExample

```
public void run() {
	// Tutaj powinno nastąpić sprawdzenie poprawności
	parametrów domenowych

	// Losowanie klucza prywatnego dA Alicji. Liczba całkowita
	z przedziału <1, n - 1>
	long alicePrivateKey = alicePrivKeyGen.get(1L,
	DomainParameters.orderG - 1);

	// Obliczenie klucza publicznego Qa Alicji dA x G
	Point alicePublicKey = multiplyPoint(alicePrivateKey,
	DomainParameters.G);

	// Losowanie klucza prywatnego dB Boba. Liczba całkowita z
	przedziału <1, n - 1>
	long bobPrivateKey = bobPrivKeyGen.get(1,
	DomainParameters.orderG - 1);

	// Obliczenie klucza publicznego Qb Boba dB x G
	Point bobPublicKey = multiplyPoint(bobPrivateKey,
	DomainParameters.G);

	// Alicja otrzymuje klucz publiczny Qb od Boba
	// Przed użyciem powinna go sprawdzić (np. czy punkt ten w
	ogóle leży na krzywej)
	// Oblicza sekretny punkt S = dA x Qb
	Point aliceSecretPoint = multiplyPoint(alicePrivateKey,
	bobPublicKey);

	// Możemy otrzymać punkt w nieskończoności. Powinniśmy
	wtedy rozpocząć komunikację na nowo
	if (aliceSecretPoint.isInfinity()) {
		System.out.println("Niepomyślne parametry. " +
			"Sekretny punkt obliczony przez Alice jest
		punktem w nieskończoności. " +
			"Spróbuj ponownie");

		return;
	}
```

```
    // Otrzymany sekret nie spełnia warunków dobrego klucza
    szyfrującego
    // Używamy więc KDF - Key Derivation Function
    String aliceSecret = kdf(aliceSecretPoint.x);


    // Bob otrzymuje klucz publiczny Qa od Alicji
    // Przed użyciem powinien go sprawdzić (np. czy punkt ten
    leży na krzywej)
    // Oblicza sekretny punkt S = dB x Qa
    Point bobSecretPoint = multiplyPoint(bobPrivateKey,
    alicePublicKey);

    // Możemy otrzymać punkt w nieskończoności. Powinniśmy
    wtedy rozpocząć komunikację na nowo
    if (bobSecretPoint.isInfinity()) {
         System.out.println("Niepomyślne parametry. " +
              "Sekretny punkt obliczony przez Boba jest punktem
         w nieskończoności. " +
              "Spróbuj ponownie");
         return;
    }

    // Otrzymany sekret nie spełnia warunków dobrego klucza
    szyfrującego
    // Używamy więc KDF - Key Derivation Function
    String bobSecret = kdf(bobSecretPoint.x);

    if (!aliceSecret.equals(bobSecret))  {
         throw new RuntimeException("Wymiana kluczy nie
         powiodła się. " +
              "Sekret Alicji=" + aliceSecret + " != Sekret
              Boba=" + bobSecret + "\n" +
              "Najprawdopodobniej jest to błąd w implementacji
              lub niepoprawne parametry domenowe");
    } else {
         System.out.println("Sekret Alicji=" + aliceSecret + "
         == " + "Sekret Boba=" + bobSecret);
         System.out.println("OK!");
    }
}
```

3.5.3. Metoda dodająca punkt do siebie x razy

Klasa: pl.edu.prz.kijko.ECDHExample

```
private Point multiplyPoint(long multiplier, Point point) {
	if (point.isInfinity()) {
		return Point.INFINITY;
	}

	Point result = addPoints(point, point);

	for (long i = 1; i <= (multiplier - 2); i++) {
		result = addPoints(result, point);
	}

	return result;
}
```

3.5.4. Metoda dodająca dwa punkty

Klasa: pl.edu.prz.kijko.ECDHExample

```
private Point addPoints(Point P, Point Q) {
	if (P.isInfinity()) {
		return Q;
	}

	if (Q.isInfinity()) {
		return P;
	}

	long mod = DomainParameters.p;
	if (P.equals(Q)) { // Operacja podwojenia punktu

		Point negP = new Point(P);
		negP.y = adjustToMod(negP.y * (-1));

		// Rząd podwajanego punktu = 2, więc P + P = INF
		if (negP.equals(P)) {
			return Point.INFINITY;
		}

		long mNumerator = adjustToMod(3 * (P.x * P.x) +
		DomainParameters.a);
		long mDenominator = adjustToMod(2 * P.y);
```

```java
            if (mDenominator == 0) {
                throw new RuntimeException("Mianownik lambdy = 0.
                Nie powinno się to zdarzyć. Sprawdź kod");
            }

            long mDenominatorInverse =
            calcModInverse(mDenominator);
            long lambda = adjustToMod(mNumerator *
            mDenominatorInverse);

            return calcPointsSum(lambda, P, Q);
        } else { // Dodawanie punktów
             // punkty na tej samej prostej pionowej
            if (P.x == Q.x) {
                return Point.INFINITY;
            }

            long mNumerator = adjustToMod(Q.y - P.y);
            long mDenominator = adjustToMod(Q.x - P.x);

            long mDenominatorInverse =
            calcModInverse(mDenominator);
            long lambda = adjustToMod(mNumerator *
            mDenominatorInverse);

            return calcPointsSum(lambda, P, Q);
        }
    }
```

3.5.5. Metoda pomocnicza “redukująca” daną liczbę do przedziału <*0, p – 1*>

Klasa: pl.edu.prz.kijko.ECDHExample

```java
    private long adjustToMod(long num) {
        long result = num % DomainParameters.p;

        if (result < 0) {
            result += DomainParameters.p;
        }

        return result;
    }
```

3.5.6. Metoda licząca odwrotność modularną modulo *p*

Klasa: pl.edu.prz.kijko.ECDHExample

```
// Metoda zakłada, że "num" i "DomainParameters.p" są względnie
pierwsze
private long calcModInverse(long num) {
     long totient = totientForPrime(DomainParameters.p);

     long inverse = adjustToMod(num * num);
     for (long i = 1; i <= (totient - 1) - 2; i++) {
          inverse = adjustToMod(inverse * num);
     }

     return inverse;
}
```

3.5.7. Metoda licząca wartość funkcji Fi Eulera (działa jedynie dla liczb pierwszych)

Klasa: pl.edu.prz.kijko.ECDHExample

```
// Metoda zakłada "num" jako liczbę pierwszą
private long totientForPrime(long num) {
     return num - 1;
}
```

3.5.8. Metoda pomocnicza licząca współrzędne punktu wynikowego z dodawania dwóch punktów

Klasa: pl.edu.prz.kijko.ECDHExample

```
private Point calcPointsSum(long lambda, Point P, Point Q) {
     long Rx = adjustToMod((long) (lambda * lambda - P.x -
     Q.x));
     long Ry = adjustToMod((long) (lambda * (P.x - Rx) - P.y));

     return new Point(Rx, Ry);
}
```

3.5.9. Metoda imitująca KDF (Key Derivation Function)

Metoda imituje KDF (Key Derivation Function) do pozyskania klucza szyfrującego z sekretnej liczby uzgodnionej przez obie strony poprzez ECDH. W rzeczywistej implementacji kryptosystemu możnaby się tutaj spodziewać np. implementacji HKDF (HMAC-based KDF) rekomendowanej w RFC5869

Klasa: pl.edu.prz.kijko.ECDHExample

```
     private String kdf(long sharedPointX) {
          return "kdf(" + sharedPointX + ")";
     }
```

3.5.10. Klasa reprezentująca punkt na krzywej

Klasa: pl.edu.prz.kijko.Point

```
public class Point {
    public static final Point INFINITY = new Point();

    public long x;
    public long y;
     private boolean infinity;

    public Point() {
        this.x = 0;
        this.y = 0;
          this.infinity = true;
    }

    public Point(Point point) {
        this.x = point.x;
        this.y = point.y;
        this.infinity = point.isInfinity();
    }

    public Point(long x, long y) {
        this.x = x;
        this.y = y;
        this.infinity = false;
    }

     public boolean isInfinity() {
```

```java
        return this.infinity;
    }


    public String toString() {
        return "(" + x + ", " + y + ")";
    }

    // dla uproszczenia pominąłem equals i hashCode

}
```

4. Wnioski

Podsumowując, ECDH jest bardziej efektywną i przyszłościową alternatywą dla klasycznego DH, szczególnie w kontekście rosnących wymagań bezpieczeństwa i potrzeb wydajnościowych. Jego elastyczność i wysoka wydajność sprawiają, że jest niezastąpionym elementem współczesnych systemów kryptograficznych.

5. Źródła

https://planetcalc.com/3311/

https://neilmadden.blog/2017/05/17/so-how-do-you-validate-nist-ecdh-public-keys/

https://mareknarozniak.com/2020/11/30/ecdh/

https://andrea.corbellini.name/ecc/interactive/reals-add.html

https://curves.xargs.org/

https://graui.de/code/elliptic2/

https://en.wikipedia.org/wiki/Diffie%E2%80%93Hellman_key_exchange

https://en.wikipedia.org/wiki/Diffie%E2%80%93Hellman_problem

https://en.wikipedia.org/wiki/Discrete_logarithm

https://en.wikipedia.org/wiki/Elliptic-curve_Diffie%E2%80%93Hellman

https://pl.wikipedia.org/wiki/Kryptografia_krzywych_eliptycznych

https://en.wikipedia.org/wiki/National_Institute_of_Standards_and_Technology

Standards for Efficient Cryptography, SEC 2: Recommended Elliptic Curve Domain Parameters https://www.secg.org/sec2-v2.pdf