جامعة الامير سلطان
PRINCE SULTAN UNIVERSITY

College of Computer and Information Sciences (CCIS)

CS285: Discrete Mathematics for Computing

Implementation and Analysis of the Diffie–Hellman Key Exchange Algorithm in Java

Course Instructor: Dr. Jalila Zouhair

Section #: 963

Project Contributors:

SHOUG FAWAZ ABDULLAH ALOMRAN 223410392

ALJOHARA WALEED A ALBAWARDI 223410346

YARA MUTLAQ MOHAMMED ALZAMEL 223410834

FAI MOHAMMAD BIN KHANJAR 223410071

جامعة الامير سلطان
PRINCE SULTAN UNIVERSITY

Table Of Contents

1. Introduction

In today's digital world, secure communication is the need of the hour to protect sensitive information being exchanged over public networks. The Diffie-Hellman key exchange algorithm allows two parties to agree on a shared secret key without actually exchanging it, and forms a basis for many of today's encryption schemes.

The following Java code for the Diffie–Hellman implementation is provided for CS285, Discrete Mathematics for Computing. The developed system incorporates some of the very fundamental mathematical concepts regarding modular arithmetic and exponentiation. The system simulates secure key generation and exchange between two entities using the Diffie–Hellman key exchange method. A simple encryption and decryption mechanism has also been provided to illustrate how the derived shared key can be used to secure messages

We divide the program into the following classes: Parameters handles all public values, KeyExchange generates and computes keys, Encryptor encrypts messages, and Validator, Helpers, and Utils ensure correct input handling and output formatting. The Main class integrates all components of the system and allows users to test both a predefined numerical example and a live interactive mode.

The project ties together theoretical mathematics and programming in practice, showing how secure key exchange and encryption work in real-world systems.

1.1 Purpose of the Diffie–Hellman Key Exchange and Its Applications

The Diffie-Hellman key exchange is a cryptography technique whose main purpose is to allow two parties to exchange a key that can be used for encrypting messages.This shared key

can be used in public channels that ensures no one unauthorized can read it. The key concept was developed by Ralph Merkle but was published by Whitfield Diffie and Martin Hellman in 1976, which is why it's named after them. The Diffie-Hellman key exchange is used in many ways in real life for example its use in Transport Layer Security (TLS). TLS uses a set of protocols to establish a secure tunnel for web traffic. During the initial "handshake" process, the browser authenticates the server's TLS certificate to verify its identity. Once both parties are authenticated, they use a Diffie-Hellman (DH) variant to exchange public values and generate a unique session key. This key encrypts all data transmitted throughout the browsing session, ensuring that sensitive information such as passwords and credit card numbers remains protected (Ambatipudi, 2024). Another use is Secure Shell commonly known as SSH. This network protocol is used for securely logging into and controlling remote computers, as well as transferring files. SSH uses the DH algorithm early in the connection process to establish a secret key between the client and the server, ensuring that all subsequent commands, outputs, and file transfers are encrypted and protected from interception (Ambatipudi, 2024).

1.2 Algorithm Explanation

The Diffie–Hellman key exchange algorithm allows two people known as Alice and Bob to create a secret key and send it over insecure channels such as the Internet. To generate matching secret keys, the Diffie–Hellman method uses modular arithmetic and exponentiation, in which both users perform calculations using a shared set of public numbers. Without knowing the private numbers, this process is almost impossible to reverse, but it is simple to compute in one direction. First, a primitive root α and a large prime number q are publicly agreed upon by both Alice and Bob. Everyone can know these two values; they are not hidden. After that, Alice chose

an integer Xa, and Bob chose an integer Xb; both integers should be less than q and kept private. The third step is by generating their public keys as Ya = α^{Xa} mod q and Yb = α^{Xb} mod q. Both Alice and Bob can now exchange their public keys using the network. Now, using modular arithmetic once more, each user raises the received public key to the power of their own private key in order to obtain the shared secret. Alice calculates $k = Yb^{Xa}$ mod q, while Bob calculates $k = Ya^{Xb}$ mod q; this will result in the same secret key. Even though the values q, α, Ya, and Yb are publicly available, it is nearly impossible to deduce the private keys Xa or Xb from them, which is what gives the Diffie–Hellman algorithm its security. This complexity is due to the fact that it is very challenging to solve the discrete logarithm problem, which is the task of determining the exponent used in modular arithmetic, for large numbers (Diffie & Hellman, 1976; Roy, Datta & Mitchell, 2007).

During a royal convoy, vehicles and the control center must communicate on location and security instructions. Each vehicle and the control center first reach an agreement on some public integers. Then they create their own private keys and calculate the public keys from them. After sending the public keys to each other, both sides use the Diffie–Hellman method, specifically the Elliptic Curve version (ECDH), to arrive at the same secret key. This key helps maintain the security of the messages. Even if someone is able to intercept the data, they won't be able to decode the messages or determine exactly which route taken by the convoy. Section 1.3 provides an easy explanation of how this is implemented

1.3 Numerical Example

To give an example, both 2 cars and the control center need a safe way to communicate during the royal convoy.

1. Everyone agrees on two public numbers: q = 23 and α = 5.
2. The first car chooses a private number Xa = 6 and calculates its public key $Ya = a^{Xa}$ mod $q = 5^{6}$ mod 23= 8.
3. Car 2 chooses a private number Xb= 15 and calculates its public key $Yb = a^{Xb}$ mod $q = 5^{15}$ mod 23= 19.
4. They exchange their public keys Ya and Yb.
5. Each car calculates the shared secret key:
   - Car 1 computes Yb^Xa mod q = 19^{6} mod 23 = 2.
   - Car 2 computes Ya^{Xb} mod q = 8^{15} mod 23 = 2.
6. Both cars get the same secret key 2, which keeps their messages safe so no one can read them.

Numerical Example (Section 1.3 demo - fixed values)

```
--- Numerical Example (Section 1.3 - Fixed) ---
To give an example, both 2 cars and the control center need a safe way to communicate.
Public Parameters:
Prime number (q): 23
Primitive root (alpha): 5
Car 1 chooses Xa = 6 → Ya = 5^6 mod 23 = 8
Car 2 chooses Xb = 15 → Yb = 5^15 mod 23 = 19
Each car computes the shared key:
Car 1: 19^6 mod 23 = 2
Car 2: 8^15 mod 23 = 2
Both cars share the same key: 2

Example message: Royal convoy message remains secure through shared key exchange.
Encrypted Message = ia7NqGzfh+I5LTLc5lVhclaXOtXeTEEkKh+E60QkvOWus9HpdJeW4iI8NYW1UGVzQJJ925tHBCwzFYL5CjC8qA==
Decrypted Message = Royal convoy message remains secure through shared key exchange.
Decryption OK = true
```

Numerical Example (auto-generated values)

```
--- Numerical Example (Auto-generated values) ---
Automatically generated example parameters and results:
Public Parameters:
Prime number (q): 197
Primitive root (alpha): 54
Car 1 (Xa): 2 --> Ya = 158
Car 2 (Xb): 99 --> Yb = 54
Shared key for Car 1: 158
Shared key for Car 2: 158
Keys match: true
```

1.4 Advantages & Limitations

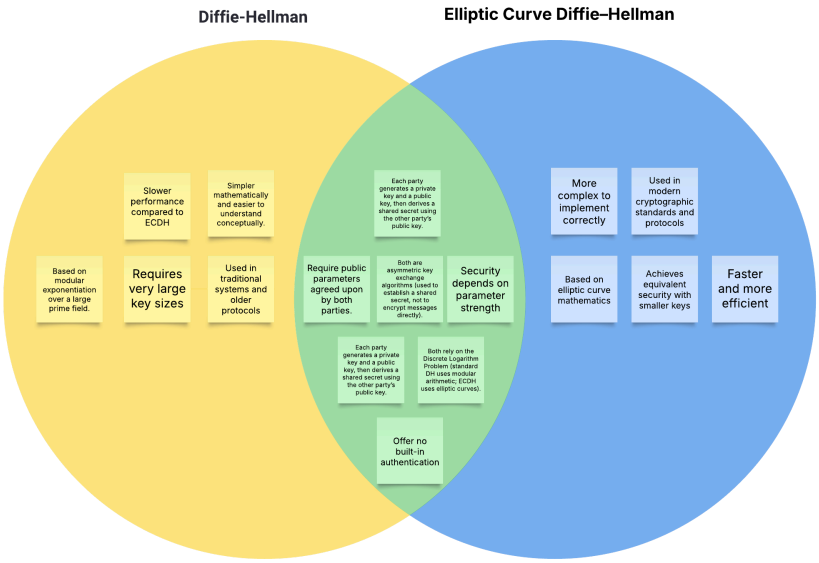
The Diffie-Hellman (DH) key exchange is a quite ancient public-key cryptography scheme, in which two individuals desiring to derive a mutual secret key over an insecure line of communication agree. Its security relies upon the solvability of the discrete logarithmic problem (DLP). Mathematical problem whose solution becomes effectively impossible to achieve

employing very large prime numbers (RubinStein-Salzedo 101). One of the reasons that Diffie-Hellman is most useful is that it offers secure key generation without needing an advanced key exchange. Both sides create a public and private value, and even if the attacker gets the public information, they cannot calculate the private key because of the hardness of DLP (RubinStein-Salzedo 104). This makes Diffie-Hellman a foundation for secure communication, especially where advance trust or physical key exchange is not feasible. Another key advantage is that DH can be designed to provide forward secrecy in case of correct usage. Forward secrecy ensures that even if a long-term private key is broken, previous messages encrypted with it are still secure. Adrian et al. refer to this element as the one that works towards limiting the effects of security violations and ensuring long-term confidentiality of computer systems (Adrian et al. 41). Diffie-Hellman is also great in theory and easy to implement, and this has set the standard for other secure protocols. Odlyzko records that application of modular arithmetic and exponentiation in the algorithm are well-chosen parameters to provide theoretical security (Odlyzko 3).

Although being robust, the Diffie-Hellman algorithm has several frailties that must be addressed to deploy it safely. Its vulnerability to man-in-the-middle (MitM) attacks is the first among them. As the original protocol is not authenticating the user, there is an opportunity for the attacker to intercept messages, impersonating both parties, and generate two different shared keys, which enables them to read or alter all the communications (Rubinstein-Salzedo 109). Because authentication has not been applied, the system assumes both parties' public key is legitimate, which can be tampered with. Parameter reuse and incorrect configuration are another vulnerability. Adrian et al. have learned that the majority of real-world systems reuse one set of primes on several servers, making it simpler to attack them if one set gets into the wrong hands

(Adrian et al. 42). Reusing weak or common primes diminishes general security because an adversary can precompute solutions for those specific values. Finally, DH is slower computationally than symmetric-key algorithms. Odlyzko explains that big primes require significant processing power, especially on small machines or low-end machines (Odlyzko 7). There must be a proper balance of the performance/security trade-off in order to realize practicability in implementations.

1.5 Comparison



The original Diffie-Hellman (DH) scheme rests on the pillars of arithmetic and the discrete logarithm problem, which together hand it a mathematically sound foundation for swapping keys securely. Still as we pointed out earlier its robustness leans heavily on the development of primes and the careful tuning of its parameters. Toss in primes or recycle the parameters over and over and the whole construction becomes open to sub-exponential and pre-computation attacks - adversaries can tease out the private keys by playing with the publicly shared values. Beyond that Diffie-Hellman can fall prey to man-in-the-middle attacks when authentication is omitted. Such shortcomings curb its efficiency and make it hard to scale for systems that demand both speed and robust security.

Elliptic Curve Diffie-Hellman (ECDH) avoids many of the shortcomings by relying on elliptic-curve mathematics instead of modular arithmetic (Boneh & Shparlinski, 2001;

Haakegaard & Lang, 2015). Since the elliptic-curve discrete logarithm problem (ECDLP) is far tougher to solve, ECDH can provide security with smaller keys - for example, a 256-bit ECDH key offers roughly the same strength as a 3072-bit DH key. Unlike Diffe-Hellman, which can be weakened when many users share the parameters, elliptic-curve Diffie-Hellman (ECDH) lets each party pick its own curve settings, thereby slashing the threat of large-scale pre-computation attacks. It also brings secrecy and unpredictability to the bits - knowing a slice of the key offers no edge to an attacker (Boneh & Shparlinski, 2001). Though ECDH adds some complexity, it has been widely embraced in cryptographic protocols such as TLS 1.3 and SSH, striking a sweet spot between performance and strong security.

In summary, both DH and elliptic-curve share a common foundation yet differ significantly in design and efficiency. ECDH rests on the discrete-logarithm problem, it's practical and is secure. It preserves DH's strengths while delivering faster performance, smaller keys, and stronger resistance to known attacks.

2. Code Description

2.1 How the Program Works

For our program, we implemented a main program which tests all of our implementations, supported by six classes. Each class had a distinct rule, the parameters class stores and displayed our public parameters which included the prime number and primitive root. The key exchange class generated private and public keys and computed a shared secret key. We also had an encryptor class which handled encrypting and decrypting messages that used the shared key. Additionally, a validator class which performs for prime checks and making sure our message length is the right length. A utilities class was implemented with the main goal of making our

program run better, it had helper methods such as random generation and formatted output. Finally, we have the helpers class which manages all our user inputs and ensures correctness such as message validation loops.

2.2 Implementation of Each Part

Key Generation

```
private void generateKeys() {
        this.privateKey = new BigInteger(getQ().bitLength(),
random).mod(getQ().subtract(BigInteger.TWO))
                .add(BigInteger.ONE);
        // 1 <= x <= q-2

        this.publicKey = getAlpha().modPow(privateKey, getQ());
        // Y = α^x mod q
    }
```

To generate a private key, we would choose a random number from 1 to the prime number (q) minus 2. Then we find the public key (y) using the formula Y = α^x mod q.

Encryption

```
    String encrypt(String msg, BigInteger sharedKey) throws Exception {
        byte[] msgByte = msg.getBytes(StandardCharsets.UTF_8); // converts the text
into bytes
        byte[] keyByte = Key(sharedKey); // uses the key function to store the data
        byte[] cipherMsgByte = xor(msgByte, keyByte); // ciphers the text
        return Base64.getEncoder().encodeToString(cipherMsgByte);
    }
```

To encrypt the message, we convert the message into Byte using the getBytes function, storing each byte in the array. Next, we use the Key method to convert the shared key to a byte then hashing the key, converting it into SHA-256, making it more difficult to decrypt. Lastly we

cipher the message using the xor function which decrypts the bytes using the public key formula. This then can be printed in the main giving us multiple random letters that aren't readable.

Decryption

```
String decrypt(String base64cipherMsg, BigInteger sharedKey) throws Exception {
        byte[] keyByte = Key(sharedKey); // stores data
        byte[] cipherMsgByte = Base64.getDecoder().decode(base64cipherMsg); //
decodes the data using a method
        byte[] msgByte = xor(cipherMsgByte, keyByte); // uses the same equation to
return it to a readable text
        return new String(msgByte, StandardCharsets.UTF_8);
    }
```

For decryption, it's similar to encryption with a few differences. It starts with the Key method, hashing and organizing the bytes first then decoding the message using the getDecoder() function. Lastly it uses the xor again to return it into readable texts.

2.3 Challenges Faced and How We Overcame Them

In the process of testing and development, a lot of problems were encountered, particularly in input checking, class integration, and modular arithmetic operations. The most frequent problem was with initial testing of the validator and Main classes, where invalid user input, such as non-prime input numbers or message lengths less than 20 characters, at one point caused the program to crash. This was done by adding relentless loops of verification that will repeatedly ask the user instead of terminating execution, ensuring the program will be executed successfully regardless of wrong inputs.

The second issue occurred while combining different team members' classes. Because every class (like Parameters, KeyExchange, and Encryptor) was programmed separately, variable names and method types needed to be standardized in a way that the Main class would compile

and execute as needed. To avoid this, the team standardized method names such as generatePublicKey() and computeSharedKey() and inserted temporary placeholders in Main until they were finished writing all classes so it would be simple to insert later.

Besides, preliminary tests were producing incorrect values in mathematical calculations for the public and common keys. This was due to the absence of modular exponentiation logic. Using Java's native modPow() method into the keyExchange class fixed this issue, delivering correct keys that matched both vehicles exactly.

Finally, it was even attempted that encryption was exposing contradictions between the encrypted and decrypted messages due to discrepancies in how byte arrays were being handled. Altering the Encryptor class to use StandardCharsets.UTF_8 and Base64 encoding ensured all messages decrypted flawlessly into its original text.

By overall systematic testing of all components, addition of good validation, and standardized naming conventions coordination, the team managed to get beyond integration as well as logic flaws and produce a functional implementation of the Diffie-Hellman key exchange system.

3. Results and Test Cases

Successful run with matching shared keys and valid messages

Autogenerated keys

```
--- Live Mode ---
You can either enter your own values for q and alpha, or let the program generate them.

Choose parameter mode:
a) Auto-generate q and alpha
b) Enter manually
Your choice (a/b): a
Automatically generated parameters:
q = 173, alpha = 128

Public Parameters:
Prime number (q): 173
Primitive root (alpha): 128

Choose private-key mode:
a) Auto-generate private keys
b) Enter private keys manually
Your choice (a/b): a
Auto private keys generated: Xa = 114, Xb = 96

Enter a message (> 20 characters): CS285 is really really fun!

--- RESULTS ---
q = 173
alpha = 128
Xa = 114
Xb = 96
Ya = 151
Yb = 106
Shared key = 60
Original Message = CS285 is really really fun!
Encrypted Message = me4Ix0O/bpgJFyV/3EXubsnGIRaRe5ILrjhv
Decrypted Message = CS285 is really really fun!
Decryption OK = true
Secure session established between convoy vehicles and control center.
```

Manual Keys

```
--- Live Mode ---
You can either enter your own values for q and alpha, or let the program generate them.

Choose parameter mode:
a) Auto-generate q and alpha
b) Enter manually
Your choice (a/b): b
Enter a prime q (≥ 3): 17
Enter alpha (1 < alpha < q): 14
Public Parameters:
Prime number (q): 17
Primitive root (alpha): 14

Choose private-key mode:
a) Auto-generate private keys
b) Enter private keys manually
Your choice (a/b): b

Private keys must be integers between 1 and q-2 (inclusive).
Enter private key for Car 1: 15

Private keys must be integers between 1 and q-2 (inclusive).
Enter private key for Car 2: 13
Enter a message (> 20 characters): This message is directed to members of CS285.

--- RESULTS ---
q = 17
alpha = 14
Xa = 15
Xb = 13
Ya = 11
Yb = 5
Shared key = 7
Original Message = This message is directed to members of CS285.
Encrypted Message = nl3uK9a/Gx+HMxX2WR7UaJnhS3jVDau+XbPQP207hRuvR/R4mbReL6dgSqZX
Decrypted Message = This message is directed to members of CS285.
Decryption OK = true
Secure session established between convoy vehicles and control center.
```

Invalid prime q & alpha

Manually Keys

```
--- Live Mode ---
You can either enter your own values for q and alpha, or let the program generate them.

Choose parameter mode:
a) Auto-generate q and alpha
b) Enter manually
Your choice (a/b): b
Enter a prime q (≥ 3): 27
That's not a prime number. Try again.

Enter a prime q (≥ 3): 18
That's not a prime number. Try again.

Enter a prime q (≥ 3): 17
Enter alpha (1 < alpha < q):
```

```
Enter alpha (1 < alpha < q): 18
Alpha must be > 1 and < q. Try again.

Enter alpha (1 < alpha < q): 0
Alpha must be > 1 and < q. Try again.

Enter alpha (1 < alpha < q): 13
Public Parameters:
Prime number (q): 17
Primitive root (alpha): 13
```

Short message (< 20 chars) validation error

Autogenerated Keys

```
--- Live Mode ---
You can either enter your own values for q and alpha, or let the program generate them.

Choose parameter mode:
a) Auto-generate q and alpha
b) Enter manually
Your choice (a/b): a
Automatically generated parameters:
q = 233, alpha = 106

Public Parameters:
Prime number (q): 233
Primitive root (alpha): 106

Choose private-key mode:
a) Auto-generate private keys
b) Enter private keys manually
Your choice (a/b): a
Auto private keys generated: Xa = 222, Xb = 219

Enter a message (> 20 characters): Short Message
Message too short. Please try again (must be > 20 characters).

Enter a message (> 20 characters):
```

4. Team Contribution

It was a collaborative project that required everyone to coordinate, communicate, and work together. All members of the group collaborated on both the Java implementation and the report, allowing for simultaneous technical and theoretical progress.

Yara Alzamel established the foundation of the program by declaring the public parameters and helper methods upon which the rest of the system depended. She was responsible for designing the classes that are responsible for the generation of the public values, i.e., the prime number and primitive root, and adding helper functions for normalizing random number generation and formatted print. She was responsible for the foundation on which the rest of the program was constructed and also to describe the program's initialization and overall flow in the report.

Fai Bin Khanjar took care of the underlying mathematics of implementing the Diffie–Hellman algorithm. She implemented key generation with private keys on both sides, calculated their corresponding public keys, and made sure that the resulting shared secret key generated was identical on both sides. She created a numerical example based on the report to explain step-by-step algorithm calculation. Her attentiveness during modular arithmetic verification and accuracy result checks helped to ensure the program's correctness.

Aljohara Albawardi developed the encryption and decryption components of the project. She implemented a lightweight encryption mechanism that used the shared secret key derived from Fai's module. Her focus was on converting theoretical key exchange into practical message security, ensuring that messages longer than twenty characters could be securely encrypted and successfully decrypted. She also conducted test runs, captured the output by taking screenshots, and mentioned security advantages and restrictions of the algorithm in the report.

Shoug Alomran executed validation, integration, and assembly of the program last. She used input validation to ensure that users enter a valid prime number and that the size of the message complies with project requirements. She applied all modules—key generation, key exchange, encryption, and output formatting—inside the driver program, tested the whole program running, and managed exceptions so that the program ran flawlessly end to end. Apart from her technical contribution, she also integrated the group's written elements into the final report, maintaining the structure, tone, and format intact.

The team achieved coordination primarily by using GitHub as a version manager and code sharer, allowing each member to push and sign off on changes. Regular group meetings were held to facilitate communication and share progress updates and integration concerns. Each team member examined other members' work and participated in final test sessions to confirm that the system produced accurate results and that all project criteria were met. This enabled the team to efficiently divide the burden, balance theory and practice, and deliver a well-documented, unified project.

5. Conclusion

This project provided a nice insight into how secure communication systems operate using modular arithmetic and key exchange concepts. Implementing the Diffie-Hellman algorithm was an example in demonstration how two parties can safely come to agreement on a shared secret key even across an insecure medium using concepts like modular exponentiation and prime number verification. Walking through the code reinforced our admiration for how public and private keys mathematically interact to maintain confidentiality.

Apart from the technical details, the project also highlighted the importance of coordination and teamwork among members. Each class such as Parameters, KeyExchange and Encryptor

depended on the others, thus the need for frequent communication and sequential testing to achieve flawless integration. Debugging also taught patience and precision, especially in identifying errors of logic or validation affecting program flow.

For further growth, the project can be expanded by using stronger methods of encryption, such as AES for message encryption, or by modifying the KeyExchange class to accommodate the real elliptic-curve parameters for optimal ECDH performance.

6. References and Resources

Ambatipudi, S. (2024, June 6). *Cryptographic advancements enabled by Diffie–Hellman.* ISACA Journal, Volume 3.
https://www.isaca.org/resources/isaca-journal/issues/2024/volume-3/cryptographic-advancements-enabled-by-diffie-hellman

Boneh, D., & Shparlinski, I. E. (2001). *On the unpredictability of bits of the Elliptic Curve Diffie–Hellman scheme*. In J. Kilian (Ed.), *Advances in Cryptology – CRYPTO 2001* (pp. 201–212). Springer.

https://doi.org/10.1007/3-540-44647-8_12

Diffie, W., & Hellman, M. E. (1976). *New directions in cryptography. IEEE Transactions on Information Theory, 22*(6), 644–654.
https://www-ee.stanford.edu/~hellman/publications/24.pdf

Haakegaard, R., & Lang, J. (2015). *The Elliptic Curve Diffie–Hellman (ECDH)*. University of California, Santa Barbara. Retrieved from
http://koclab.cs.ucsb.edu/teaching/ecc/project/2015Projects/Haakegaard+Lang.pdf

IEEE, Institution of Electrical and Electronics Engineers. "Automated Analysis of Diffie-Hellman Protocols and Advanced Security Properties | IEEE Conference Publication | IEEE Xplore." *Automated Analysis of Diffie-Hellman Protocols and Advanced Security Properties*, 2012, ieeexplore.ieee.org/document/6266153/.

Krawczyk, Hugo. "HMQV: A High-Performance Secure Diffie-Hellman Protocol." *SpringerLink*, Springer Berlin Heidelberg, 1 Jan. 2005, link.springer.com/chapter/10.1007/11535218_33.

Odlyzko, A. M. (2014). *Discrete logarithms in finite fields and their cryptographic significance.*

www-users.cse.umn.edu/~odlyzko/doc/arch/discrete.logs.pdf.

Roy, A., Datta, A., & Mitchell, J. C. (2007). *Formal proofs of cryptographic security of Diffie–Hellman–based protocols*. Stanford University. https://theory.stanford.edu/people/jcm/papers/rdm-tgc-07.pdf

Shoug Alomran, Yara, Fai, & Aljohara. (2025). *CS285 – Secure Key Exchange Project: A Java implementation of the Diffie–Hellman algorithm for the CS285 course* [Source code]. GitHub. https://github.com/Shoug-Alomran/CS285-CryptographyProject

Alomran, S. F., Khanjar, F. M., Albawardi, A. W., & Alzamel, Y. M. (2025). CS285 – Secure Key Exchange Project: Interactive documentation site. Prince Sultan University.https://shoug-alomran.github.io/CS285-CryptographyProject/

Appendix – Full Source Code

Main Class

```
import java.math.BigInteger;
import java.security.SecureRandom;
import java.util.Scanner;

public class Main {

   private static final SecureRandom random = new SecureRandom();

   public static void main(String[] args) {
       Scanner input = new Scanner(System.in);

       System.out.println("--- Royal Convoy - Secure Communication (Diffie-Hellman / ECDH Demo) ---");
       System.out.println("This program demonstrates how the control center and vehicles exchange keys securely.\n");

       while (true) {
           System.out.println("MAIN MENU:");
           System.out.println("1) Numerical Example (Section 1.3 demo - fixed values)");
           System.out.println("2) Numerical Example (auto-generated values)");
           System.out.println("3) Live Mode (manual / auto parameters and private keys)");
           System.out.println("0) Exit");
           System.out.print("Choose an option: ");

           String choice = input.nextLine().trim();

           switch (choice) {
               case "1":
                   runFixedExample();
                   break;
               case "2":
                   runRandomExample(input);
                   break;
               case "3":
                   runLiveMode(input);
                   break;
               case "0":
                   System.out.println("Goodbye!");
```

```
                    input.close();
                    return;
                default:
                    System.out.println("Invalid choice. Please enter 1, 2, 3, or
0.\n");
                    break;
            }
        }
    }

    // ———————————— Numerical Example - Fixed (as in report) ————————————
    private static void runFixedExample() {
        System.out.println("\n--- Numerical Example (Section 1.3 - Fixed) ---");

        BigInteger q = BigInteger.valueOf(23);
        BigInteger alpha = BigInteger.valueOf(5);
        Parameters storage = new Parameters(q, alpha);

        BigInteger Xa = BigInteger.valueOf(6);
        BigInteger Xb = BigInteger.valueOf(15);

        BigInteger Ya = alpha.modPow(Xa, q);
        BigInteger Yb = alpha.modPow(Xb, q);

        BigInteger kA = Yb.modPow(Xa, q);
        BigInteger kB = Ya.modPow(Xb, q);

        System.out.println("To give an example, both 2 cars and the control center
need a safe way to communicate.");
        System.out.println(storage.toString());
        System.out.println("Car 1 chooses Xa = 6 → Ya = 5^6 mod 23 = " + Ya);
        System.out.println("Car 2 chooses Xb = 15 → Yb = 5^15 mod 23 = " + Yb);
        System.out.println("Each car computes the shared key:");
        System.out.println("Car 1: 19^6 mod 23 = " + kA);
        System.out.println("Car 2: 8^15 mod 23 = " + kB);
        System.out.println("Both cars share the same key: " + kA + "\n");

        String message = "Royal convoy message remains secure through shared key
exchange.";
        System.out.println("Example message: " + message);
```

```
        try {
            Encryptor enc = new Encryptor();
            String cipher = enc.encrypt(message, kA);
            String plain = enc.decrypt(cipher, kB);
            System.out.println("Encrypted Message = " + cipher);
            System.out.println("Decrypted Message = " + plain);
            System.out.println("Decryption OK = " + plain.equals(message) + "\n");
        } catch (Exception e) {
            System.out.println("Encryption/Decryption failed: " + e.getMessage());
        }
    }

    // ———————————— Numerical Example - Random ————————————
    private static void runRandomExample(Scanner input) {
        System.out.println("\n--- Numerical Example (Auto-generated values) ---");

        BigInteger q = BigInteger.probablePrime(8, random);
        BigInteger alpha = BigInteger.valueOf(random.nextInt(3, q.intValue() - 1));
        Parameters storage = new Parameters(q, alpha);

        BigInteger Xa = BigInteger.valueOf(random.nextInt(2, q.intValue() - 2));
        BigInteger Xb = BigInteger.valueOf(random.nextInt(2, q.intValue() - 2));

        BigInteger Ya = alpha.modPow(Xa, q);
        BigInteger Yb = alpha.modPow(Xb, q);
        BigInteger kA = Yb.modPow(Xa, q);
        BigInteger kB = Ya.modPow(Xb, q);

        System.out.println("Automatically generated example parameters and
results:");
        System.out.println(storage.toString());
        System.out.println("Car 1 (Xa): " + Xa + " --> Ya = " + Ya);
        System.out.println("Car 2 (Xb): " + Xb + " --> Yb = " + Yb);
        System.out.println("Shared key for Car 1: " + kA);
        System.out.println("Shared key for Car 2: " + kB);
        System.out.println("Keys match: " + kA.equals(kB) + "\n");

        // Allow user to enter a message for encryption even in auto mode
        String message = Helpers.promptMessage(input);
        try {
```

```
            Encryptor enc = new Encryptor();
            String cipher = enc.encrypt(message, kA);
            String plain = enc.decrypt(cipher, kB);
            System.out.println("--- Encryption Test ---");
            System.out.println("Original Message  = " + message);
            System.out.println("Encrypted Message = " + cipher);
            System.out.println("Decrypted Message = " + plain);
            System.out.println("Decryption OK = " + plain.equals(message) + "\n");
        } catch (Exception e) {
            System.out.println("Encryption/Decryption failed: " + e.getMessage());
        }
    }

    // ———————————— Live Mode (manual or auto q, alpha, and keys) ————————————
    private static void runLiveMode(Scanner input) {
        System.out.println("\n--- Live Mode ---");
        System.out.println("You can either enter your own values for q and alpha, or
let the program generate them.\n");

        System.out.println("Choose parameter mode:");
        System.out.println("a) Auto-generate q and alpha");
        System.out.println("b) Enter manually");
        System.out.print("Your choice (a/b): ");
        String paramChoice = input.nextLine().trim().toLowerCase();

        BigInteger q, alpha;

        if ("b".equals(paramChoice)) {
            q = Helpers.promptPrime(input);
            alpha = Helpers.promptAlpha(input, q);
        } else {
            q = BigInteger.probablePrime(8, random);
            alpha = BigInteger.valueOf(random.nextInt(3, q.intValue() - 1));
            System.out.println("Automatically generated parameters:");
            System.out.println("q = " + q + ", alpha = " + alpha + "\n");
        }

        Parameters storage = new Parameters(q, alpha);
        System.out.println(storage + "\n");
```

```
        System.out.println("Choose private-key mode:");
        System.out.println("a) Auto-generate private keys");
        System.out.println("b) Enter private keys manually");
        System.out.print("Your choice (a/b): ");
        String keyChoice = input.nextLine().trim().toLowerCase();

        BigInteger Xa, Xb;
        if ("b".equals(keyChoice)) {
            Xa = Helpers.promptPrivateKey(input, q, "Enter private key for Car 1: ");
            Xb = Helpers.promptPrivateKey(input, q, "Enter private key for Car 2: ");
        } else {
            Xa = BigInteger.valueOf(random.nextInt(2, q.intValue() - 2));
            Xb = BigInteger.valueOf(random.nextInt(2, q.intValue() - 2));
            System.out.println("Auto private keys generated: Xa = " + Xa + ", Xb = "
+ Xb + "\n");
        }

        BigInteger Ya = alpha.modPow(Xa, q);
        BigInteger Yb = alpha.modPow(Xb, q);
        BigInteger kA = Yb.modPow(Xa, q);
        BigInteger kB = Ya.modPow(Xb, q);

        if (!kA.equals(kB)) {
            System.out.println("Shared keys do not match. Please try again.\n");
            return;
        }

        // Always allow message input regardless of parameter mode
        String message = Helpers.promptMessage(input);
        try {
            Encryptor enc = new Encryptor();
            String cipher = enc.encrypt(message, kA);
            String plain = enc.decrypt(cipher, kB);

            System.out.println("\n--- RESULTS ---");
            System.out.println("q = " + q);
            System.out.println("alpha = " + alpha);
            System.out.println("Xa = " + Xa);
            System.out.println("Xb = " + Xb);
            System.out.println("Ya = " + Ya);
```

```
            System.out.println("Yb = " + Yb);
            System.out.println("Shared key = " + kA);
            System.out.println("Original Message = " + message);
            System.out.println("Encrypted Message = " + cipher);
            System.out.println("Decrypted Message = " + plain);
            System.out.println("Decryption OK = " + plain.equals(message));
            System.out.println("Secure session established between convoy vehicles
and control center.\n");

        } catch (Exception e) {
            System.out.println("Encryption/Decryption failed: " + e.getMessage());
        }
    }
}
```

Parameters Class

```
import java.math.BigInteger;

public class Parameters {

    private BigInteger q;
    private BigInteger alpha;

    // Constructor that sets q and alpha
    public Parameters(BigInteger q, BigInteger alpha) {
        this.q = q;
        this.alpha = alpha;
    }

    public BigInteger getQ() {
        return q;
    }

    public BigInteger getAlpha() {
        return alpha;
```

```
    }

    @Override
    public String toString() {
        return "Public Parameters:\n" +
                "Prime number (q): " + q + "\n" +
                "Primitive root (alpha): " + alpha;
    }
}
```

Utils Class

```
import java.math.BigInteger;
import java.security.SecureRandom;

public class Utils {
    // Create a static SecureRandom object for use across classes
    private static final SecureRandom secureRandom = new SecureRandom();

    // - Return random BigInteger in range [1, upperLimit - 1]
    public static BigInteger getRandom(BigInteger upperLimit) {
        BigInteger result;
        do {
            result = new BigInteger(upperLimit.bitLength(), secureRandom);
        } while (result.compareTo(BigInteger.ONE) < 0 || result.compareTo(upperLimit)
>= 0);
        return result;
    }
    // - Format console output for screenshots (e.g., "User A public key: 8")
    public static void printLine(String label, Object value) {
        System.out.println(label + ": " + value);
    }
    // - Remove leading zero byte if present in key.toByteArray()
    public static byte[] normalizeKeyBytes(BigInteger key) {
        byte[] bytes = key.toByteArray();
        if (bytes.length > 1 && bytes[0] == 0) {
            byte[] normalized = new byte[bytes.length - 1];
            System.arraycopy(bytes, 1, normalized, 0, normalized.length);
```

```
            return normalized;
        }
        return bytes;
    }
}
```

Validator Class

```
import java.math.BigInteger;

public class Validator {

    // true if q is a prime number ≥ 3
    public static boolean isPrime(BigInteger q) {
        if (q == null)
            return false;
        if (q.compareTo(BigInteger.valueOf(3)) < 0)
            return false; // reject < 3
        return q.isProbablePrime(100); // Miller-Rabin (built into BigInteger)
    }

    // true if 1 < alpha < q
    public static boolean isAlphaInRange(BigInteger alpha, BigInteger q) {
        return alpha != null && q != null
                && alpha.compareTo(BigInteger.ONE) > 0
                && alpha.compareTo(q) < 0;
    }

    // true if message has more than 20 non-space characters
    public static boolean isValidMessage(String message) {
        return message != null && message.trim().length() > 20;
    }
}
```

Key Exchange Class

```java
import java.math.BigInteger;
import java.security.SecureRandom;

public class KeyExchange extends Parameters {

   private SecureRandom random = new SecureRandom();

   // Attributes to store keys
   private BigInteger privateKey;
   private BigInteger publicKey;

   public KeyExchange(BigInteger q, BigInteger alpha) {
       super(q, alpha); // call Parameters class constructor
       generateKeys(); // generate public and private keys when object is created
   }

   // Generate a random private key and compute corresponding public key
   private void generateKeys() {
       this.privateKey = new BigInteger(getQ().bitLength(),
random).mod(getQ().subtract(BigInteger.TWO))
               .add(BigInteger.ONE);
       // 1 <= x <= q-2

       this.publicKey = getAlpha().modPow(privateKey, getQ());
       // Y = α^x mod q
   }

   // Compute shared key using peer's public key

   public BigInteger computeSharedKey(BigInteger otherPublic) {
       return otherPublic.modPow(privateKey, getQ());
   }

   // Getters for the keys
   public BigInteger getPrivateKey() {
       return privateKey;
   }

   public BigInteger getPublicKey() {
       return publicKey;
```

```
    }
}
```

Encryptor Class

```
import java.math.BigInteger;
import java.util.Base64;
import java.security.MessageDigest;
import java.nio.charset.StandardCharsets;

public class Encryptor {
    // Convert shared BigInteger key into a byte array (keyByte)
    // uses bigInteger for huge values
    private byte[] Key(BigInteger key) throws Exception {
        MessageDigest msgDigest = MessageDigest.getInstance("SHA-256"); // SHA for
Secure Hash Algorithm, it outputs 256 outputs
        byte[] ArrayByte = key.toByteArray(); // converts each letter to a byte to
help in ciphering the texts letter by letter
        return msgDigest.digest(ArrayByte); // stores the bytes in the program
    }

    // xor function
    private byte[] xor(byte[] data, byte[] key) {
        byte[] xorArray = new byte[data.length]; // creates a new byte array to store
the data
        for (int i = 0; i < data.length; i++) {
            xorArray[i] = (byte) (data[i] ^ key[i % key.length]); // equation to
cipher msg
        }
        return xorArray;
    }

    // Implement encrypt(message, key)
    String encrypt(String msg, BigInteger sharedKey) throws Exception {
        byte[] msgByte = msg.getBytes(StandardCharsets.UTF_8); // converts the text
into bytes
        byte[] keyByte = Key(sharedKey); // uses the key function to store the data
```

```
        byte[] cipherMsgByte = xor(msgByte, keyByte); // ciphers the text
        return Base64.getEncoder().encodeToString(cipherMsgByte);
    }

    // Implement decrypt(ciphermsg, key)
    String decrypt(String base64cipherMsg, BigInteger sharedKey) throws Exception {
        byte[] keyByte = Key(sharedKey); // stores data
        byte[] cipherMsgByte = Base64.getDecoder().decode(base64cipherMsg); //
decodes the data using a method
        byte[] msgByte = xor(cipherMsgByte, keyByte); // uses the same equation to
return it to a readable text
        return new String(msgByte, StandardCharsets.UTF_8);
    }

    // Check messages before encryption
    public static void validateMsg(String msg) {
        if (msg == null) {
            throw new IllegalArgumentException("Message cannot be empty");
        }
    }

    // Check messages before decryption
    public static void validateCipherMsg(String cipherMsg) {
        if (cipherMsg == null) {
            throw new IllegalArgumentException("Cipher message cannot be empty");
        }
    }
}
```

Helpers Class

```
import java.math.BigInteger;
import java.security.SecureRandom;
import java.util.Scanner;

public class Helpers {
```

```
    // 1. Prompt for a prime q (≥ 3). Re-prompts until valid.
    public static BigInteger promptPrime(Scanner input) {
        while (true) {
            try {
                System.out.print("Enter a prime q (≥ 3): ");
                BigInteger q = new BigInteger(input.nextLine().trim());
                if (Validator.isPrime(q))
                    return q;
                System.out.println("That's not a prime number. Try again.\n");
            } catch (Exception e) {
                System.out.println("Invalid input. Please type an integer.\n");
            }
        }
    }

    // 2. Prompt for alpha with range check 1 < alpha < q. Re-prompts until valid.
    public static BigInteger promptAlpha(Scanner in, BigInteger q) {
        while (true) {
            try {
                BigInteger alpha = new BigInteger(in.nextLine().trim());
                if (Validator.isAlphaInRange(alpha, q))
                    return alpha;
                System.out.println("Alpha must be > 1 and < q. Try again.\n");
            } catch (Exception e) {
                System.out.println("Invalid input. Please type an integer.\n");
            }
        }
    }

    // 3. Prompt for private keys in the inclusive range [1 .. q-2]. Re-prompts
    // until valid.
    public static BigInteger promptPrivateKey(Scanner input, BigInteger q, String
message) {
        System.out.println("\nPrivate keys must be integers between 1 and q-2
(inclusive).");
        BigInteger min = BigInteger.ONE;
        BigInteger max = q.subtract(BigInteger.TWO); // q-2
        while (true) {
            try {
                System.out.print(message);
```

```
                BigInteger privateKey = new BigInteger(input.nextLine().trim()); //
Read input
                if (privateKey.compareTo(min) >= 0 && privateKey.compareTo(max) <= 0)
                    return privateKey;
                System.out.println("Invalid range. Please enter a value between 1 and
q-2.\n");
            } catch (Exception e) {
                System.out.println("Invalid input. Please type an integer.\n");
            }
        }
    }

    // 4. Generate a random private key in [1 .. q-2].
    public static BigInteger randomPrivateKey(BigInteger q, SecureRandom randomNo) {
        BigInteger privateKey; // Placeholder for the generated key
        BigInteger min = BigInteger.ONE;
        BigInteger max = q.subtract(BigInteger.TWO); // q - 2

        while (true) {
            // Picks a random integer smaller than q
            privateKey = new BigInteger(q.toString()).abs(); // Generate random
BigInteger (the abs() ensures
                                                             // non-negative)

            // If it's between 1 and q-2, we accept it
            if (privateKey.compareTo(min) >= 0 && privateKey.compareTo(max) <= 0) {
                return privateKey;
            }
            // Otherwise, try again
        }
    }

    // 5. Prompt for a message (> 20 characters). Re-prompts until valid.
    public static String promptMessage(Scanner input) {
        System.out.println("\nStep 3: Enter the message to be securely sent between
vehicles.");
        while (true) {
            System.out.print("Enter a message (> 20 characters): ");
            String message = input.nextLine();
            if (Validator.isValidMessage(message))
```

```
                return message;
            System.out.println("Message too short. Please try again (must be > 20
characters).\n");
        }
    }
}
```