Bs"d

Elliptic Curve Diffie-Hellman

Final Project in Data Security

Participants:

Nofar Duchan 322599424

Sivan Senvetu 207279845

Bs"d

Table of Contents

Introduction 3
The Problem That ECDH Solves 3
Cryptographic Assumptions 4
Mathematical Principles 4
Protocol Description 5
Advantages of ECDH 6
Challenges and Concerns in Using ECDH 6
Implementation of ECDH 7
Summary of ECDH Implementation Steps in the Code 9
Runtime Complexity of ECDH 10
Accuracy and Reliability 11
Key Challenges in ECDH Implementation 11
Comparison of ECDH and Classic Diffie-Hellman 12
Potential Pitfalls in Real-World ECDH Implementation 12
Examining the Impact on Future Standards 12
Bibliography 13

Introduction

Elliptic Curve Diffie-Hellman (ECDH) is an advanced cryptographic protocol for secure key exchange, based on elliptic curve principles. This protocol significantly improves upon the traditional Diffie-Hellman method, providing a higher level of security while using shorter keys. This advantage makes ECDH particularly efficient for resource-constrained systems, such as mobile devices and IoT applications.

ECDH is one of the most important algorithms in modern cryptography due to its unique combination of high security and computational efficiency. The algorithm enables two parties to securely generate a shared encryption key, even when communicating over an insecure channel, without exposing critical information to a third party.

One of the key characteristics of ECDH is its reliance on the Elliptic Curve Discrete Logarithm Problem (ECDLP). This mathematical problem is considered exceptionally difficult and serves as the foundation for ECDH's strong resistance against modern cryptanalytic attacks. Furthermore, the use of shorter keys not only improves efficiency but also reduces memory and processing requirements, making ECDH ideal for a wide range of applications.

The Problem That ECDH Solves

ECDH addresses the challenge of securely exchanging cryptographic keys over an insecure communication channel. It allows two parties, even without prior acquaintance, to generate a shared encryption key without revealing sensitive information to a third party.

When two parties, such as Alice and Bob, wish to communicate securely, they need a shared key for encrypting and decrypting their messages. Sending the key directly over an insecure channel exposes it to potential interception by a third party, which could enable them to decrypt messages and compromise communication confidentiality.

ECDH provides an elegant solution to this problem. By utilizing elliptic curves and advanced mathematical tools, the protocol enables Alice and Bob to establish a shared key without directly transmitting the key itself. This means that even if a third party intercepts the communication, they cannot deduce the shared key or decrypt the exchanged messages.

Thanks to this innovative approach, ECDH offers strong protection against interception and cryptographic attacks, making it an essential tool for securing communication in the digital age.

Cryptographic Assumptions

- **Elliptic Curve Discrete Logarithm Problem (ECDLP)** – This problem is considered extremely difficult, with no known efficient solution. It forms the basis of ECDH security and is the elliptic curve adaptation of the traditional discrete logarithm problem.
- **Irreversibility of Scalar Multiplication** – Given a point P and another point Q, where $Q = Pk \cdot k$, it is computationally infeasible to determine k. While scalar multiplication is easy to compute, reversing the operation is extremely difficult.
- **Secure Parameter Selection** – The security of ECDH depends on choosing appropriate elliptic curve parameters. Poorly chosen parameters can introduce vulnerabilities. Therefore, it is crucial to use standardized and widely accepted parameters defined by reputable organizations.

Mathematical Principles

- **Elliptic Curve** – An elliptic curve is an algebraic curve defined by the equation: $y^2 = x^3 + ax + b$ where a, b are constants belonging to a finite field.
  A fundamental requirement is that $4a^3 + 27b^2 \neq 0$ to ensure the curve is smooth.
- **Finite Field in ECDH** – ECDH operates over finite fields, which are finite sets of numbers where arithmetic operations are well-defined. The two most common types of finite fields used in ECDH are:
  - $GF(p)$ Prime Modular Field – Where p is a prime number.
  - $GF(2^m)$ Binary Field – Where m is a positive integer.
- **Base Point (G)** – A fixed point selected on the elliptic curve, serving as the starting point for scalar multiplication calculations.
- **Scalar Multiplication** – Scalar multiplication involves multiplying a point P on the elliptic curve by an integer k, yielding a new point Q as follows: $Q = k \cdot P$ This operation is fundamental for computing public and shared keys in ECDH.

Protocol Description

Public Parameter Definition:

- Selection of an elliptic curve and a finite field.
- Selection of a base point G on the curve.

Private and Public Key Generation:

- Each party (Alice and Bob) selects a secret number as their private key:
  - Ad for Alice
  - Bd for Bob
- Each party computes their public key using scalar multiplication:
  - $G \cdot Bd = BPG \cdot Ad = AP$

Shared Key Computation:

- Alice computes: $BP \cdot Ad = S$
- Bob computes: $AP \cdot Bd = S$
- Since $G \cdot Bd \cdot Ad = S$ Both parties derive the same shared key S.

Advantages of ECDH

One of the most significant advantages of ECDH is the high level of security it provides while using significantly shorter keys. This means that strong security can be achieved with lower storage and processing requirements, making ECDH highly efficient for resource-constrained devices.

Additionally, ECDH offers significantly improved performance. This is because scalar multiplication operations on elliptic curves are computationally simpler and faster. This advantage makes ECDH particularly suitable for devices with limited resources, such as smart cards, IoT devices, and mobile devices.

Another crucial advantage is the scalability of ECDH in common security protocols. ECDH is easily integrated into protocols such as TLS (versions 1.3 and above), SSH, and PGP, making it a popular choice for applications that require both high security and efficiency.

Due to these benefits, ECDH has become one of the most widely used and respected key exchange algorithms in modern cryptography. It provides an optimal combination of security, efficiency, and flexibility, making it an ideal choice for a wide range of applications, from securing internet communication to protecting data on mobile devices and IoT systems.

Challenges and Concerns in Using ECDH

Despite being a strong and efficient protocol, ECDH is not without challenges and concerns. One of the primary challenges is its vulnerability to timing attacks and side-channel leakage. These attacks exploit variations in the execution time of cryptographic operations or power consumption patterns to infer private key values. To mitigate this risk, various hardening techniques must be applied, such as constant-time operations and adding random noise to power consumption.

Another challenge is the need for secure parameter management. ECDH relies on public parameters such as the elliptic curve and base point. If these parameters are improperly chosen or sourced from an unreliable provider, they can introduce cryptographic vulnerabilities. Therefore, it is crucial to use standardized and widely recognized parameters selected by trusted organizations.

Finally, secure random number generation is critical for ECDH security. Private keys must be generated with complete randomness, as predictable keys can expose the shared secret. This necessitates the use of strong cryptographically secure pseudo-random number generators (CSPRNGs).

Despite these challenges, ECDH remains a robust, efficient, and secure key exchange protocol. It is a fundamental building block in modern cryptography, as elliptic curve cryptography enables high levels of security while using shorter keys compared to traditional methods. However, proper implementation and awareness of these challenges are essential to ensure its security.

Implementation of ECDH

Project Link on GitHub - https://github.com/nofarduchan/data_security.git

We used several functions from the cryptography library to perform encrypted key exchange using the ECDH (Elliptic Curve Diffie-Hellman) algorithm and to generate a shared key between a client and a server. The library includes the following functions:

- **ec.generate_private_key –** Creates a key pair (private and public) based on the SECP256R1 curve.
- **Serialization –** Allows the public key to be serialized into PEM format, which can be sent over the network.
- **HKDF (HMAC-based Key Derivation Function) –** Extracts a secret key from a shared secret to generate a one-time key for secure usage.
- **Hashes –** Uses the SHA-256 function for computing the digital signature of the keys and the shared secret.
- **Socket -** Key exchange in the context of ECDH occurs between two parties in the system. In our code, the two parties are the server and the client. We used sockets to simulate real network communication.

```
import socket
from cryptography.hazmat.primitives.asymmetric import ec
from cryptography.hazmat.primitives import serialization
from cryptography.hazmat.primitives.kdf.hkdf import HKDF
from cryptography.hazmat.primitives import hashes
import os
```

Explanation of Functions:

- **generate_key_pair** appears in both the server and client files.
  This is the fundamental function that generates a key pair – a private key and a public key. It uses the SECP256R1 elliptic curve, which is a standard and secure choice. In the ECDH process, each party (client and server) must generate its own key pair, where the private key is kept secret and the public key is sent to the other party. The SECP256R1 curve was chosen for its high-security level and broad system support.

```
def generate_key_pair():
    private_key = ec.generate_private_key(ec.SECP256R1())
    public_key = private_key.public_key()
    return private_key, public_key
```

- **derive_shared_secret** is the core of the ECDH protocol. It receives the private key of one party, the public key received from the other party, and the shared salt. The function performs the ECDH mathematical computation on the elliptic curve to generate a shared secret. Then, it uses HKDF (Key Derivation Function) to derive a symmetric key of 32 bytes from the shared secret. The use of HKDF ensures that the final key is uniformly distributed and suitable for symmetric encryption.

```
def derive_shared_secret(private_key, peer_public_key, salt):
    shared_secret = private_key.exchange(ec.ECDH(), peer_public_key)

    derived_key = HKDF(
        algorithm=hashes.SHA256(),
        length=32,
        salt=salt,
        info=b"ECDH Key Agreement"
    ).derive(shared_secret)

    return derived_key
```

- **serialize_public_key** is responsible for converting a public key into PEM format, which is suitable for network transmission. PEM is a base64-encoded textual format widely used in cryptography. This function is essential for ECDH as it allows the parties to exchange their public keys in a standard and structured format over the network.

```
def serialize_public_key(public_key):
    return public_key.public_bytes(
        encoding=serialization.Encoding.PEM,
        format=serialization.PublicFormat.SubjectPublicKeyInfo
    )
```

- **start_client** implements the client side of the ECDH protocol.
  It establishes a TCP connection to the server, generates a key pair for the client, receives the server's public key and salt, sends the client's public key to the server, and finally computes the shared key. All these steps are necessary to ensure that both the client and server securely derive the same shared key.

- **start_server** implements the server side of the ECDH protocol.
  It listens for incoming connections, generates a key pair for the server and a random salt, sends its public key and salt to the client, receives the client's public key, and finally computes the shared key. The server generates the salt to ensure that each key derivation is unique, adding an extra layer of security to the protocol.

Explanation of Functions in the tests.py file:

- **test_generate_key_pair –** Ensures that key pair generation works correctly.
- test_shared_secret – Verifies that the ECDH protocol generates the same shared key on both sides.
- **test_invalid_key –** Checks that the system detects and rejects invalid keys.
  These tests are crucial to ensure that the ECDH implementation functions correctly and is safe to use.
- **test_shared_secret_mismatch -** The function is designed to verify that the shared key generation process works correctly, and if there is a mismatch between the generated keys, it will be identified.

Summary of ECDH Implementation Steps in the Code

1. Both parties agree in advance on the elliptic curve parameters – in our code, SECP256R1.
2. The server generates a random salt to be used later.
   (A salt is a random sequence of bits or bytes added to the key derivation process to ensure uniqueness, even if the same shared secret is used.)
3. The server generates its key pair (private and public).
4. The client generates its key pair (private and public).
5. The server sends its public key and salt to the client.
6. The client sends its public key to the server.
7. The server uses its private key and the client's public key to compute the shared secret.
8. The client uses its private key and the server's public key to compute the shared secret.
9. Both parties use HKDF with the server-generated salt to derive the final symmetric key from the shared secret.
10. Both parties verify that they obtained the exact same shared key.
11. This key can then be used for symmetric encryption of their communication.

The major advantage of ECDH is that it allows two parties to agree on a shared key without any third party listening in on the communication being able to determine what the key is since the private keys never travel over the network.
Using elliptic curves (EC) instead of standard DH (Diffie-Hellman) achieves the same security level with shorter keys, making the protocol more efficient.

Runtime Complexity of ECDH

The ECDH algorithm is based on Elliptic Curve Cryptography (ECC), and its runtime depends on the key size and arithmetic operations on the elliptic curve.

1. Key Factors Affecting ECDH Runtime

The algorithm consists of the following steps:

- Generating private and public keys → Requires arithmetic computations on the elliptic curve.
- Computing the shared secret using ECDH → Involves elliptic curve point multiplication.
- Deriving a key using HKDF → Runs a hash function (e.g., SHA-256).

2. Computational Complexity of ECDH

- Elliptic curve point multiplication is the most computationally expensive step, involving a series of modular addition and multiplication operations. This is generally O(log(n)), where n is the order of the curve.
- HKDF (Key Derivation Function) performs SHA-256 hashing, which runs in O(n), where n is the input size in bits.

3. Overall Runtime Complexity of ECDH

- The overall runtime complexity of ECDH is O(log(n)), due to elliptic curve arithmetic operations.
- Standard key sizes (e.g., SECP256R1) directly impact runtime.
- HKDF adds a small overhead O(n) but does not significantly affect performance.

4. Practical Execution Time on a Standard Computer

- For a 256-bit key (SECP256R1), ECDH key exchange takes a few milliseconds on a modern CPU.
- With specialized hardware like TPMs or crypto accelerators, execution time can drop to microseconds.

Accuracy and Reliability

During the evaluation of ECDH, it was found that the algorithm performs secure key exchange while maintaining a high level of security with significantly shorter keys compared to traditional Diffie-Hellman. Verification of the computations confirmed that the shared key generated on both sides was identical in all experiments, reinforcing the correctness of the protocol.

Finally, the tests also included an analysis of how different parameters impact the algorithm's performance. It was found that using standardized and recommended elliptic curves, such as secp256r1, ensures the highest level of security. Additionally, key size should be chosen based on the required security level and available system resources.

Overall, the test results confirm that ECDH is a reliable, efficient, and secure key exchange algorithm. It provides high security with shorter keys compared to traditional methods, making it an ideal choice for a wide range of applications.

Key Challenges in ECDH Implementation

During the implementation of ECDH, we encountered several key challenges that required creative and tailored solutions.

First, choosing standardized and secure parameters was critical. We had to ensure the use of recognized and well-established elliptic curves, such as secp256r1, to prevent potential vulnerabilities that could compromise system security.

Second, maintaining the randomness of private keys was a significant challenge. It was essential to use reliable cryptographic random number generators (CSPRNGs) to ensure that private keys were completely unpredictable. Predictable private keys could expose the shared secret key, compromising communication security.

Comparison of ECDH and Classic Diffie-Hellman

Structure and Key Differences

In class, we covered the classic DH protocol, which led us to explore its variant, ECDH.
The most significant difference between ECDH and classic Diffie-Hellman (DH) is key size. ECDH provides the same level of security as classic Diffie-Hellman but with significantly shorter keys. For example, a 256-bit elliptic curve key offers a security level comparable to a 3072-bit key in classic Diffie-Hellman. This means that the same security level can be achieved with lower storage and processing requirements.

Another major difference is computational efficiency. Scalar multiplication on elliptic curves is computationally simpler and faster than modular exponentiation in classic DH. This makes ECDH more efficient, especially for resource-constrained systems, such as mobile devices, IoT devices, and smart cards.

Additionally, ECDH is more resistant to brute-force attacks due to the hardness of the Elliptic Curve Discrete Logarithm Problem (ECDLP). Furthermore, ECDH is less vulnerable to graph-based search attacks compared to classic DH.

Lastly, ECDH is widely used in modern protocols, including TLS 1.3, SSH, and Signal, whereas classic Diffie-Hellman is mainly used in older protocols, such as TLS 1.2. The reason for this transition is that ECDH provides higher security with shorter keys, making it more suitable for modern applications.

ECDH is considered a significant improvement over classic Diffie-Hellman, incorporating advanced mathematical principles and abstract algebra. Research highlights the importance of careful parameter selection to prevent attacks and avoid using insecure elliptic curves.

Potential Pitfalls in Real-World ECDH Implementation

- Weak elliptic curves – Using non-standard or weak curves can introduce cryptographic vulnerabilities.
- Insecure key management – Improper storage of private keys can expose them, compromising communication security.
- Timing attacks and power analysis – Countermeasures must be implemented to prevent side-channel information leakage.

Examining the Impact on Future Standards

ECDH is currently one of the leading standards in cryptography. However, advancements in quantum computing may eventually drive the transition to post-quantum cryptographic (PQC) systems. Until these technologies reach maturity, ECDH will remain a secure and efficient key exchange solution.

Bibliography

1. Haakegaard, R., & Lang, J. (2015). The Elliptic Curve Diffie-Hellman (ECDH). http://koclab.cs.ucsb.edu/teaching/ecc/project/2015Projects/Haakegaard+Lang.pdf

2. Maurer, U. M., & Wolf, S. (2000). The Diffie-Hellman Protocol. Designs, Codes and Cryptography, 19, 147–171. https://link.springer.com/article/10.1023/A:1008302122286

3. Dubal, M. J., Mahesh, T. R., & Ghosh, P. A. (2011). Design of New Security Algorithm Using Hybrid Cryptography Architecture. IEEE. https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=5941965