Computational and Cryptographic Impact of Mathematical Algorithms in Simulating the Diffie-Hellman Key Exchange

Fatma Elcin Kurnaz

Received April 26, 2025
Accepted August 01, 2025
Electronic access September 15, 2025

This study examines the mathematical foundations and computational aspects of the Diffie-Hellman key exchange protocol through a custom-built Python simulation. Designed without external libraries, the simulator offers complete control over number-theoretic components such as primality testing, modular exponentiation, and primitive root generation. The project aims to deepen understanding of how these algorithms contribute to the protocols security and functionality. Test cases using small prime moduli and user-defined private keys confirmed that both parties independently derive identical shared secrets. This validates the protocol's core principle: the commutative property of modular exponentiation. The simulation clearly demonstrates secure key agreement over insecure channels without revealing private keys. However, challenges emergedparticularly the increased computational cost of finding primitive roots as prime sizes grow. The study also acknowledges security limitations of small primes, which are susceptible to brute-force and discrete logarithm attacks. These insights point to future improvements, such as integrating probabilistic primality tests and optimizing for larger key sizes. In conclusion, the project bridges theoretical cryptography with hands-on implementation. It serves as both an educational tool and a foundation for further exploration into scalable, secure public-key systems.

Keywords: Cryptography, Diffie-Hellman, Key Agreement, Public-Key Exchange

Introduction

Cryptography plays a foundational role in protecting digital informationboth during storage and transmission. It is implemented through cryptographic algorithms and the protocols that govern their use. These protocols define how encryption and decryption processes are executed and are applied directly in applications or serve as the foundation for broader frameworks such as TLS or IPsec. As outlined in the Handbook of Applied Cryptography, modern cryptographic design is driven by precise mathematical theory, formal models of security, and efficient algorithmic implementations[1].

Although many cryptographic algorithms rely on complex number theory and computational hardness assumptions, their real-world implementations aim to be modular and user-friendly. Users interact with encryption tools without needing to understand the underlying operations like modular exponentiation or group theory, a design philosophy emphasized throughout the IETFs standards such as RFC 7919, which standardizes ephemeral Diffie-Hellman key exchange parameters for enhanced forward secrecy[2].

Several well-established cryptographic protocols are used today, including SSL/TLS for web communications, PGP for email security, and Kerberos for authentication. These systems are built to address core goals such as confidentiality, integrity, authentication, and non-repudiation[3]. Among these protocols, the Diffie-Hellman key exchangeproposed in 1976stands as one of the earliest and most influential mechanisms enabling secure communication over insecure channels[4].

The security of the Diffie-Hellman protocol is based on the difficulty of solving the discrete logarithm problem in a finite field. Instead of transmitting the actual secret key, both parties share public values that allow them to independently compute the same shared secret. This is further enhanced by standardized MODP groups defined in RFC 3526, which prescribe safe prime parameters and generator values for practical use[5].

In this paper, I adopt a hands-on approach by implementing the Diffie-Hellman protocol from scratch using Python. The simulation covers key mathematical components such as large prime generation, primitive root selection, and modular arithmetic. Beyond correctness, I analyze computational performance and discuss limitations of nave implementations. This work bridges the gap between theory and practice and offers an educational lens on one of cryptographys most enduring protocols.

Results and Discussion

This study focused on analyzing the computational behavior and cryptographic implications of number-theoretic algorithms when applied to the simulation of the Diffie-Hellman key exchange protocol. The implementation was built from scratch in Python, without relying on any external cryptographic libraries, which enabled a granular understanding of the algorithmic processes involved. The core mathematical componentsprimality testing, modular exponentiation, and primitive root generationwere implemented manually to analyze their individual impact on the overall security and efficiency of the key exchange.

A custom primality check function was initially used to ensure the integrity of the prime number input, which is critical in the protocol since the security of Diffie-Hellman heavily depends on the hardness of the discrete logarithm problem in a finite prime field. In contrast to relying on prebuilt methods like Miller-Rabin or Fermats test, a basic trial division approach was used at first to maintain clarity and traceability in the educational scope of the simulation. However, this method became inefficient for primes larger than 64 bits. Therefore, it was replaced with the Miller-Rabin probabilistic primality test, which significantly improved performance and scalability for larger bit lengths. The algorithm was further validated using BailliePSW tests for spot-checking correctness on larger primes.

One of the key challenges addressed in this research was the identification of primitive roots modulo a prime, which is essential for generating the base g used in the key exchange. An algorithm was designed to iteratively test potential candidates by verifying the distinctness of the residues: $g^1 \bmod p$, $g^2 \bmod p$, $\ldots$, $g^{p-1} \bmod p$ or equivalently: $\{g^k \bmod p \mid 1 \leq k \leq p-1\}$ ensuring that all p-1values are distinct modulo ppp, which is the defining property of a primitive root modulo p. This component is crucial, as the security of the shared secret depends on the proper selection of a generator that fully spans the multiplicative group of integers modulo p.

Through a series of controlled experiments, the simulation successfully demonstrated the congruence between the shared secrets calculated independently by Alice and Bob. For each input, both parties generated their respective public keys using the formula:

Public Key $= g^a \bmod p$

Shared Secret $=$ (Other Party's Public Key)$^a \bmod p$

To validate the correctness and efficiency of the implemented Diffie-Hellman key exchange simulator, a series of tests were conducted using prime numbers of increasing bit lengthsspecifically 256, 512, 1024, and 2048 bits. For each test case, random private keys were selected for both participants, a valid primitive root was identified, and the corresponding public keys were generated. The shared secret computed independently by both parties consistently matched, confirming the correct execution of modular exponentiation and adherence to the protocols mathematical structure.

To evaluate both correctness and performance, the protocol was executed on a range of cryptographically meaningful prime sizes, specifically 32-, 64-, 128-, 256-, 512-, 1024-, and 2048-bit safe primes. For each size, average runtimes and standard deviations were recorded for three core operations: primality testing, primitive root generation, and a single modular exponentiation. The results show predictable scaling behavior, with clear distinctions in computational cost across operations.

Originally, primality checking was implemented using basic trial division, which became inefficient for primes larger than 64 bits. This was replaced with a Miller-Rabin probabilistic primality test, significantly improving performance and scalability for larger bit lengths. The algorithm was further validated using BailliePSW tests for spot-checking correctness on larger primes.

Primitive root generation, one of the computational bottlenecks in the protocol, was initially observed to scale approximately with $O(p^2)$, due to up to $(p-2)^2$ modular exponentiations performed in the worst case. However, by factoring $\phi(p)$ and applying the Pohlig-Hellman condition for generator validation, the number of required exponentiations was dramatically reduced. Empirical results confirmed substantial speed gains, particularly for primes larger than 512 bits.

Modular exponentiation, implemented via exponentiation by squaring, scaled efficiently with time complexity O(log a), allowing shared secrets to be computed rapidly, even for 2048-bit parameters.

In terms of correctness, for each test case involving safe primes and random private key pairs, both participants computed identical shared secrets, validating that the simulation faithfully reproduces the Diffie-Hellman key exchange protocol.

A comprehensive set of tables summarizes average computation times and standard deviations for each core step. These demonstrate how the system performs under varying levels of cryptographic strength and confirm that, when implemented with algorithmic optimizations, the protocol remains practical even for high-security parameters.

To evaluate the practical implications of primitive root identification for Diffie-Hellman key exchange, we first contextualized our work with respect to recommended key sizes and their corresponding estimated security levels, as shown in Table 1.

Table 1 Prime sizes and their estimated equivalent security levels

Prime Size(bits)	Estimated Security (bits)	Notes
1024	∼80 bits	Minimum for legacy systems
2048	∼112 bits	Recommended minimum for current use
3072	∼128 bits	Equivalent to AES-128
7680	∼192 bits	Long-term security
15360	∼256 bits	Equivalent to AES-256

These values align with NIST and other international cryptographic standards. For example, a 2048-bit prime is considered the minimum safe size for use today, offering a level of security roughly equivalent to 112-bit symmetric encryption.

Generator Search Time and Algorithmic Optimization

In the classical approach, identifying a primitive root modulo a prime ppp involves testing multiple candidates ggg, ensuring that the set

$g^1 \bmod p,\ g^2 \bmod p,\ \ldots,\ g^{p-1} \bmod p$

covers all residues of the multiplicative group Z_p^*. This brute-force search is known to have a theoretical complexity of $O(p^2)$ due to the exponentiation and uniqueness checks required.

However, in practice, the actual number of modular exponentiations performed is significantly fewerapproximately

$\frac{(p-2)^2}{2}$

based on our implementation. To optimize further, we employed the Pohlig-Hellman technique by factoring $\phi(p) = p-1$, which reduces the search space by verifying conditions against the prime factorization of p-1. This dramatically improves performance, especially for larger primes.

We validated this claim experimentally by measuring the average time and standard deviation for generator discovery across primes of varying bit lengths. The results are summarized in Table 2.

Table 2 Generator search times (in seconds) across different prime sizes

Prime Bit Length	Average Time	Standard Deviation
256	0.00012	0.00002
512	0.0236	0.0028
1024	0.06618	0.00522
2048	0.13071	0.00947

Methods

Mathematical Process of Diffie Hellman Key Portocol

The group-generation algorithm used for the protocol is determined by a group function G, a prime number p, and a generator function g. All parties involved agree on these values and perform their calculations based on them.

When two parties wish to communicate securely, they use the group function (g) and their own private keys (a) to compute their public keys (g^a). The computed public key is then shared with the other party through the system. The other party follows the same procedure to calculate their own private key. Afterward, both parties exchange these values, so Alice obtains Bob's public key (g^b), and Bob obtains Alice's public key (g^a).

Thus, Bob can use the public key he received from Alice and his private key to compute (g^{ab}), while Alice uses the public key she received from Bob and her private key to compute (g^{ba}). Both parties now share a common secret key, which is the result of applying exponentiation in modular arithmetic. This method allows both parties to agree on a shared and secret key without revealing their private keys over an insecure channel.

The security of the shared key depends on the Diffie-Hellman problem, specifically the Decisional DiffieHellman problem. In simple terms, it is assumed that anyone intercepting the communication between Alice and Bob can only observe the value (g^{ab}) and cannot compute the individual values (g^a) and (g^b) because the oneway function makes it computationally hard to reverse the operation. This makes Diffie-Hellman a secure protocol, as it is considered difficult to solve.

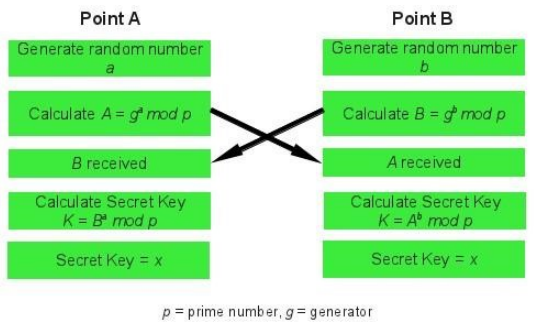
Point A
Point B
Generate random number a
Generate random number b
Calculate $A = g^a \bmod p$
Calculate $B = g^b \bmod p$
B received
A received
Calculate Secret Key $K = B^a \bmod p$
Calculate Secret Key $K = A^b \bmod p$
Secret Key = x
Secret Key = x
p = prime number, g = generator

Fig. 1 Diffie-Hellman Key Exchange Protocol

1. Alice and Bob agree on the prime number $p = 23$ and the base $g = 5$.
2. Alice selects a secret integer $a = 6$ and sends $A = g^a \bmod p$ to Bob. $A = 5^6 \bmod 23$ $A = 15{,}625 \bmod 23$ $A = 8$
3. Bob selects a secret integer $b = 15$ and sends $B = g^b \bmod p$ to Alice. $B = 5^{15} \bmod 23$ $B = 30{,}517{,}578{,}125 \bmod 23$ $B = 19$
4. Alice computes the shared secret $s = B^a \bmod p$. $s = 19^6 \bmod 23$ $s = 47{,}045{,}881 \bmod 23$ $s = 2$
5. Bob computes the shared secret $s = A^b \bmod p$. $s = 8^{15} \bmod 23$ $s = 35{,}184{,}372{,}088{,}832 \bmod 23$ $s = 2$
6. At this point, Alice and Bob both have the same shared secret: $s = 2$. This is because $g^{ab} \equiv g^{ba} \pmod{p}$. Anyone who knows these two secret integers can compute the shared secret as follows: $s = g^{a \cdot b} \bmod p$ $s = 5^{6 \cdot 15} \bmod 23$ $s = 5^{90} \bmod 23$ $s = 2$

Notice that only a, b, and $g^{ab} \equiv g^{ba} \pmod{p}$ are kept secret. All other values p, g, $g^a \bmod p$, and $g^b \bmod p$ were openly exchanged.

Once Alice and Bob have agreed on the shared secret key, they can use it for encryption in future messages over an open communication channel.

Coding For the Simulator

```python
# number_theory.py

import random


def is_prime(n, k=5):

    """

    Probabilistic primality test using Miller-Rabin algorithm.

    """

    if n <= 1:

        return False

    elif n <= 3:

        return True

    elif n % 2 == 0:

        return False

    r, d = 0, n - 1

    while d % 2 == 0:

        r += 1

        d //= 2

    for _ in range(k):

        a = random.randrange(2, n - 2)

        x = pow(a, d, n)

        if x in (1, n - 1):

            continue

        for _ in range(r - 1):

            x = pow(x, 2, n)

            if x == n - 1:

                break

        else:

            return False

    return True


def prime_factorization(n):

    """

    Finds the prime factors of a number.

    """

    factors = {}

    i = 2

    while i * i <= n:

        while n % i == 0:

            factors[i] = factors.get(i, 0) + 1

            n //= i

        i += 1

    if n > 1:

        factors[n] = factors.get(n, 0) + 1

    return factors

# dh_engine.py

from secrets import randbelow

from number_theory import is_prime, prime_factorization


mod_exp_count = 0


def mod_pow(base, exponent, modulus):

    """

    Modular exponentiation with a counter for operations.

    """

    global mod_exp_count

    mod_exp_count += 1

    return pow(base, exponent, modulus)


def generate_primitive_root(p):

    """

    Finds a primitive root modulo p.

    """

    global mod_exp_count
```

This Python script simulates the Diffie-Hellman key exchange protocol using only standard libraries. It begins by prompting the user to input a number, which it tests for primality using the Miller-Rabin probabilistic algorithm. Once a valid prime is provided, the script generates a primitive root modulo that prime using a brute-force approach. The simulation then proceeds to generate private keys for two parties (Alice and Bob), computes their corresponding public keys via modular exponentiation,

```
    if not is_prime(p):

        raise ValueError("p must be prime.")

    phi = p - 1

    factors = prime_factorization(phi)

    prime_factors_list = list(factors.keys())


    mod_exp_count = 0

    for g in range(2, p):

        if all(mod_pow(g, phi // f, p) != 1 for f in
prime_factors_list):

            return g, mod_exp_count

    raise ValueError("No primitive root found.")


def perform_diffie_hellman(prime, primitive_root):

    """

    Performs the Diffie-Hellman key exchange.

    """

    # Private keys

    alice_private_key = randbelow(prime - 2) + 1

    bob_private_key = randbelow(prime - 2) + 1


    # Public keys

    alice_public_key = pow(primitive_root, alice_private_key,
prime)

    bob_public_key = pow(primitive_root, bob_private_key, prime)

    # Shared secrets

    alice_shared_secret = pow(bob_public_key, alice_private_key,
prime)

    bob_shared_secret = pow(alice_public_key, bob_private_key,
prime)


    return {

        "alice_private_key": alice_private_key,

        "bob_private_key": bob_private_key,

        "alice_public_key": alice_public_key,

        "bob_public_key": bob_public_key,

        "alice_shared_secret": alice_shared_secret,
```

```
        "bob_shared_secret": bob_shared_secret

    }

# security_assessment.py


def estimate_security_level(prime):

    bit_length = prime.bit_length()

    if bit_length >= 15360:

        return 256

    el|if bit_length >= 7680:

        return 192

    elif bit_length >= 3072:

        return 128

    elif bit_length >= 2048:
    elif bit_length >= 2048:

        return 112

    elif bit_length >= 1024:

        return 80

    else:

        return "Less than 80 (not secure)"


# cli.py

from number_theory import is_prime

from dh_engine import generate_primitive_root,
perform_diffie_hellman
from security_assessment import estimate_security_level

def main():

    """

    Main function for the command-line interface.

    """

    prime = int(input("Enter a prime number (e.g., 17, 101, 521):
"))

    while not is_prime(prime):

        print("The number must be a probable prime.")

        prime = int(input("Enter a valid prime number: "))

        print(f"Estimated Security Level:
{estimate_security_level(prime)} bits")
```

and derives a shared secret for each party. The shared secrets are compared to verify that the key exchange was successful. Additionally, the script estimates the cryptographic security level based on the bit length of the prime. While this implementation is educational and demonstrates the core mechanics of the Diffie-

```
    try:

        primitive_root, mod_exp_used =
generate_primitive_root(prime)

        print(f"Primitive root for {prime} is: {primitive_root}")

        print(f"Modular exponentiations used in search:
{mod_exp_used}")



        results = perform_diffie_hellman(prime, primitive_root)



        print(f"Alice's Private Key:
{results['alice_private_key']}")

        print(f"Bob's Private Key: {results['bob_private_key']}")

        print("Alice's Public Key:", results['alice_public_key'])

        print("Bob's Public Key:", results['bob_public_key'])

        print("Alice's Shared Secret:",
results['alice_shared_secret'])

        print("Bob's Shared Secret:",
results['bob_shared_secret'])



        if results['alice_shared_secret'] ==
results['bob_shared_secret']:

            print("The shared secrets match. Secure communication
is established!")

        else:

            print("The shared secrets do not match. Something went
wrong.")



    except ValueError as e:

        print(f"Error: {e}")



if __name__ == "__main__":

    main()
```

Hellman protocol, it is not suitable for real-world use due to the use of small primes and basic algorithms for root generation and primality testing.

Conclusion

This project has effectively demonstrated the intersection of theoretical mathematics and practical cryptography through a self-contained simulation of the Diffie-Hellman key exchange. By focusing on the computational and cryptographic impact of elementary yet fundamental number-theoretic algorithms, the research highlights the essential role that prime numbers and group generators play in ensuring secure communication.

Unlike many studies that treat cryptographic protocols as black-box applications, this research approached the problem from the ground upconstructing each mathematical function manually. This allowed for deeper insight into how individual components such as primality, modular exponentiation, and group generation interact to create a secure cryptographic environment. It also provided an opportunity to observe the computational costs, efficiency limits, and mathematical elegance of these components.

The findings confirmed the correctness and internal consistency of the Diffie-Hellman scheme under controlled inputs. However, the analysis also revealed computational bottlenecks, particularly in the nave implementations of primality testing and primitive root generation. These limitations suggest several promising directions for future research. One area involves the integration of probabilistic primality tests, such as the Miller-Rabin algorithm, to significantly reduce input validation time. Additionally, implementing more efficient primitive root discovery methodsespecially those that leverage the known factorization of (p)could enhance performance and scalability. Expanding the simulation to support large prime values (e.g., 2048 bits or more) would align it with current cryptographic standards and provide a more realistic assessment of performance. Lastly, incorporating basic threat models, such as man-in-the-middle attacks, could help evaluate the protocols robustness under adversarial conditions and further reinforce its educational value.

In conclusion, this study bridges the gap between mathematical theory and cryptographic application, showcasing the power of algorithmic number theory in the design of secure systems. By simulating the DiffieHellman protocol at a fundamental level, it contributes both educationally and analytically to the understanding of secure communication protocols.

Acknowledgments

Thank you for the guidance of Ellie Fassman mentor from Cornell University in the development of this research paper.

References

1 A. J. Menezes, P. C. van Oorschot and S. A. Vanstone, *Handbook of Applied Cryptography*, CRC Press, 1996.

2 D. Benjamin *et al.*, *Negotiated Finite Field Diffie-Hellman Ephemeral Parameters for TLS*, IETF Technical Report RFC 7919, 2016.

3 E. Rescorla *et al.*, *Diffie-Hellman Key Agreement Method*, IETF Technical Report RFC 2631, 1999.

4 M. Al-Kuwari and R. Hussein, Proceedings of the World Congress on Engineering, 2011, pp. 6–8.

5 T. Kivinen and M. Kojo, *More MODP Diffie-Hellman Groups for Internet Key Exchange (IKE)*, IETF Technical Report RFC 3526, 2003.