Analysis of Brute Forcing Diffie-Hellman Key Exchange On Different Hardware Platforms

Alex Jasper, Deepika Dasaroju, Venkatesh Pulibandla
CSC-487
24 April 2024

1) Introduction

The Diffie-Hellman key exchange is a cryptographic protocol that enables two parties to securely share a secret key over an unsecured channel, relying on the computational difficulty of the discrete logarithm problem for security.

In practice, each party selects a private key, raises the base g to the power of their private key(typically denoted a & b) modulo p, and sends the result to the other party. Upon receiving each other's results, they raise the received value to the power of their private key, resulting in the same final value for both parties, known as the shared secret. This shared secret can then be used as a key for symmetric encryption.

In the Diffie-Hellman key exchange, each party's private key is typically a very large number. This large size is crucial because it increases the security of the exchange, making it significantly more difficult for an unauthorized party to compute the private key from the public information (the discrete logarithm problem). Generally, the private key should be randomly chosen and securely stored, as it forms the basis for generating the shared secret used for secure communication.

Diffie-Hellman is exclusively a key exchange protocol used to establish a shared secret between two parties, relying on the difficulty of the discrete logarithm problem(finding the exponent in the equation $g^x \bmod p$). RSA is an encryption and digital signature algorithm that uses a pair of keys (public and private) for encryption/decryption and signing/verification, based on the difficulty of factoring large prime numbers. While both use large prime numbers and modular arithmetic, Diffie-Hellman doesn't encrypt messages by itself, whereas RSA does both encryption and signing.

Diffie-Hellman is used to securely generate a shared secret key between two parties over an insecure channel. This key can then be used for symmetric encryption, where the same key is used for both encryption and decryption of messages. This allows secure communication using symmetric key algorithms, which are generally faster than asymmetric ones and are well-suited for encrypting large amounts of data. [1]

This study seeks to empirically evaluate the computational cost, specifically in terms of time, required to compromise the Diffie-Hellman protocol through a brute-force attack on its finite field arithmetic. For this evaluation, the protocol is implemented using a small prime number p and a corresponding primitive root g. The choice of a smaller prime number enables the completion of the brute-force attack within a feasible timeframe for academic study.

To ensure the integrity of the experiment, initial steps involve the verification of p as a prime number and g as a primitive root modulo p. Following verification, a brute-force attack is conducted to deduce a secret number integral to the key exchange process. The experiment records the duration of this attack, averaging it over multiple trials to mitigate anomalies. The crux of the research lies in contrasting the computational costs across three distinct types of computer hardware. This comparison aims to illustrate the relative efficacy of the hardware platforms in performing cryptographic operations, providing insight into the practical security considerations of the Diffie-Hellman key exchange in real-world scenarios.

2 Execution Screenshots, Graphs, and Explanations

2.1: Program Execution and Screenshots

To conduct this experiment we wrote a Python script which employed four functions that each carried out a specific task. Initially, the `generate_prime(bits)` function is deployed, utilizing a cryptographic library(pycryptodome) to produce a prime number of a specified bit length. This prime serves as the modulus in the cryptographic operations that follow.

Subsequently, the `find_primitive_root(prime)` function is engaged. Its purpose is to determine a primitive root modulo the prime number generated in the previous step. This is a crucial component in cryptographic protocols that depend on the properties of primitive roots. If the function is provided with a non-prime input, it is designed to terminate with an error, ensuring the validity of subsequent computations.

```
# Generate a prime number with a given number of bits.
1 usage   Alex Jasper
def generate_prime(bits):
    return getPrime(bits)


# Find a primitive root for a given prime number.
1 usage   Alex Jasper
def find_primitive_root(prime):
    if not isprime(prime):
        raise ValueError("Number must be prime.")

    p_minus_1 = prime - 1
    factors = factorint(p_minus_1)  # Factor p-1 to find primitive roots.

    for g in range(2, prime):  # Test potential primitive roots g.
        if all(pow(g, p_minus_1 // factor, prime) != 1 for factor in factors):
            return g  # Return g if it is a primitive root.
    return None  # Return None if no primitive root is found.
```

Figure 1: generate_prime() and find_primitive_root() source code

Upon establishing the prime and its primitive root, the `diffie_hellman_key_exchange(p, g)` function executes a single iteration of the Diffie-Hellman key exchange. It does so by randomly generating a private key, denoted 'a', and then calculating the public key 'A' via the expression g^a mod p, in accordance with the protocol's specifications.

```
# Alice creates 'a'.
1 usage   Alex Jasper
def diffie_hellman_key_exchange(p, g):
    a = random.randint( a: 1, p - 1)  # Select a private key a randomly.
    A = pow(g, a, p)  # Calculate public key A as g^a mod p.
    return a, A
```

Figure 2

To assess the vulnerability of the key exchange to brute-force attacks, the `brute_force_dh(p, g, A)` function is employed. This function aims to reconstruct the private key 'a' by iterating through potential candidates until it discovers the correct one that satisfies the equation g^a mod p = A. Concurrently, the function measures and records the duration of this brute-force process.

```
# Charlie's brute force attempt to figure out 'a'.
1 usage   Alex Jasper
def brute_force_dh(p, g, A):
    k = 1
    start_time = time.perf_counter_ns()  # Start timing.
    while pow(g, k, p) != A:
        k += 1  # Increment k until g^k mod p equals A.
    end_time = time.perf_counter_ns()  # Stop timing.
    return k, end_time - start_time  # Return the discovered key and time taken.


# Loop through specified bit sizes and perform tests.
for bits in range(10, 31, 5):
    total_time = 0
    valid_trials = 0
    for trial in range(10):
        p = generate_prime(bits)
        g = find_primitive_root(p)
        if g is None:
            continue  # Skip if no primitive root is found.
        _, A = diffie_hellman_key_exchange(p, g)
        _, time_taken_ns = brute_force_dh(p, g, A)
        total_time += time_taken_ns
        valid_trials += 1

    if valid_trials > 0:
        average_time_ns = total_time / valid_trials
        print(
            f"{bits}-bit prime, Average time taken for brute force over {valid_trials} trials: {average_time_ns:,.0f} nanoseco
    else:
        print(f"No valid trials for {bits}-bit primes due to lack of primitive roots.")
```

Figure 3

The experimental protocol iteratively increases the bit size of prime numbers from 10 to 30 bits in increments of 5 bits. For each increment, a set of 10 trials is conducted to ascertain the average time associated with the brute-force attack for each prime number. The process entails generating a prime, identifying its primitive root, executing the key exchange, and finally, determining the time required to brute-force the private key. The results are aggregated to compute the average time per successful brute-force attack for each bit size. Notably, the framework also captures and reports on trials that are aborted due to the inability to find a suitable primitive root.

2.2 Graphs and Time Outputs

```python
import pandas as pd
import matplotlib.pyplot as plt



# Create a DataFrame with the data
data = {
    'Bit Size': [10, 15, 20, 25, 30],
    'MacBook Pro M1 (nanoseconds)': [181221, 7398713, 446755263, 15536839833, 680854204121],
    'CyberPower PC (nanoseconds)': [111540, 7402340, 288394150, 10667809000, 462293863250],
    'Windows Server (nanoseconds)': [221750, 8990850, 373064570, 22977646390, 720745697924]
}

df = pd.DataFrame(data)

# Plotting the transformed data
plt.figure(figsize=(12, 10))  # Changing figure size for better visibility
plt.plot( *args: df['Bit Size'], df['MacBook Pro M1 (nanoseconds)'], marker='o', label='MacBook Pro M1', linewidth=2)
plt.plot( *args: df['Bit Size'], df['CyberPower PC (nanoseconds)'], marker='o', label='CyberPower PC', linestyle='--', linewidt
plt.plot( *args: df['Bit Size'], df['Windows Server (nanoseconds)'], marker='o', label='Windows Server', linestyle='-.', linewi

plt.xlabel('Bit Size of Prime')
plt.ylabel('Average Time(nanoseconds)')
plt.title('Performance Comparison for Prime Brute Force')
plt.legend()
plt.grid(True)
plt.yscale('log')  # Set the y-axis to logarithmic scale

plt.tight_layout()

# Show plot
plt.show()
```

Figure 4(Graph Generation Code)

This Python script uses the pandas and matplotlib libraries to analyze and visualize the average computation times required for brute-force attacks on the Diffie-Hellman key exchange across different bit sizes of prime numbers on three distinct computing systems. The data, detailing nanosecond durations for attacks on 10 to 30-bit primes, is structured into a pandas Data Frame for easy manipulation.

The script then generates a line plot, with each system's performance distinctly marked and a logarithmic scale applied to the y-axis to clearly demonstrate the exponential growth in computation time as prime sizes increase. This visual representation is crucial to understanding the relationship between prime number size and computational difficulty.

Finally, with descriptive axis labels, a grid for readability, and a legend for identification, the plot is displayed, providing insights into the performance variations between the hardware configurations in a cryptographic setting. Admittedly what is attached here is the code for the adjusted y-axis with the updated logarithmic scale for more easily viewable differences in lines at lower values as found in Figure 6.

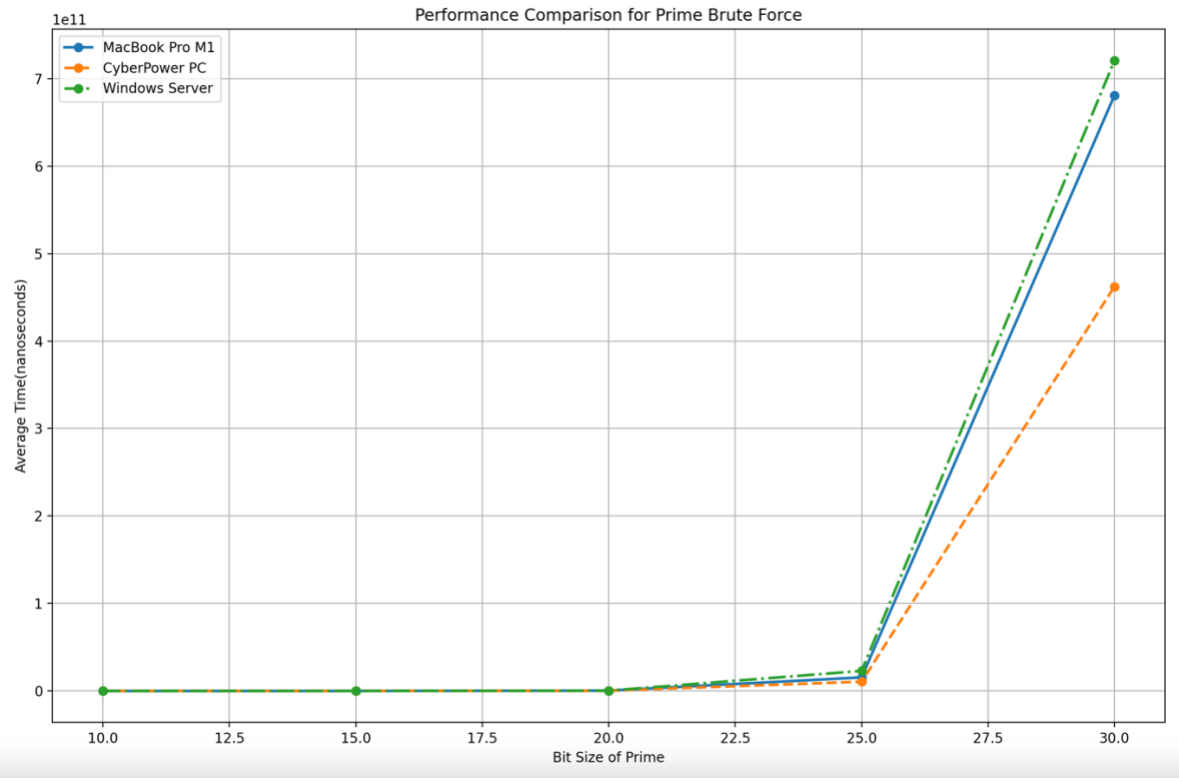

Figure 5(non-adjusted y-axis graph)

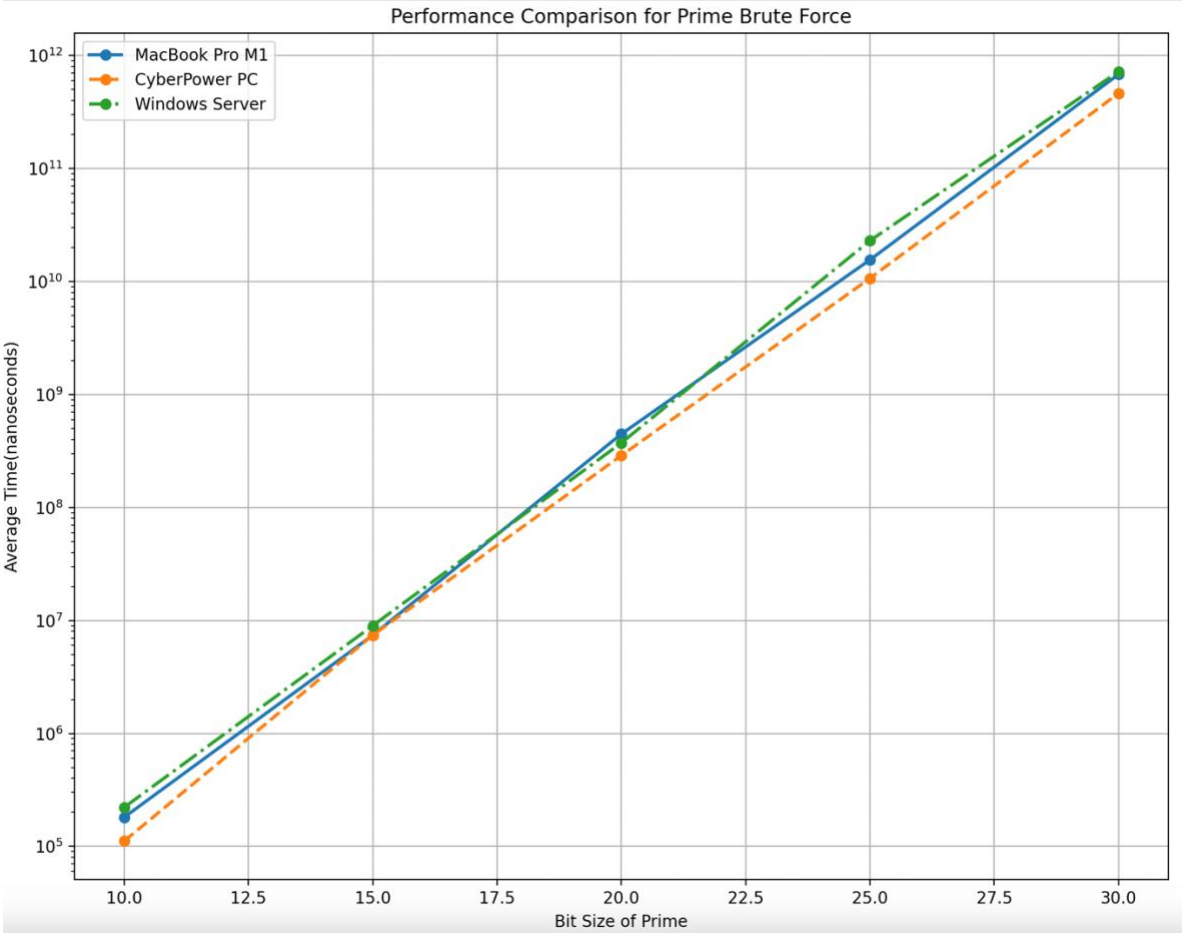

Figure 6(y-axis adjusted with default matplotlib base 10 logarithm)

2.3 Discussion/Explanation

The collected data indicates that the bit size of the private key in the Diffie-Hellman key exchange greatly affects security, showing exponential increases in the time needed for brute-force attacks. On the MacBook Pro with an M1 chip, time taken rises sharply from 181 microseconds for a 10-bit prime to over 680 seconds for a 30-bit prime, illustrating the growing complexity of the discrete logarithm problem.

The CyberPower PC, featuring an AMD Ryzen 5 5600G, outperforms the MacBook Pro, completing a 30-bit brute-force attack in approximately 462 seconds. This suggests architectural and operating system efficiencies play a significant role in cryptographic computations. Surprisingly, the Windows Server, despite its powerful AMD EPYC 64-Core Processor, does not consistently surpass the performance of other systems. For larger prime sizes, it takes longer, hinting at the impact of factors such as network latency and virtualization overhead in cloud environments on its effectiveness.

The study's limited scope of averaging ten trials per bit size and the randomness of prime number generation highlights the necessity for larger sample sizes to ascertain average computation times more accurately. Such variations in brute-force attack performance illustrate that the factors affecting cryptographic computation are diverse, from processor architecture to software environments. Processor specifications significantly influence outcomes, as the study's Python code primarily uses a single core. The M1 chip, optimized for single-threaded tasks, may hold advantages over multicore processors where cloud virtualization can negate multicore benefits.

Operating system and Python environment architecture crucially affect resource management. For instance, a Windows server might run more background processes than macOS, potentially reducing resources for brute-forcing. Python's implementation, whether CPython, PyPy, or others, can optimize or underperform depending on hardware capabilities. The sequential nature of the applied brute-force algorithm does not take advantage of multicore processors' parallel processing capabilities. Also, the variability of prime numbers' properties can lead to differences in computational effort.

3 Conclusion

The investigation into the computational efficiency of brute-force attacks on the Diffie-Hellman key exchange across various hardware systems reveals important insights into the interplay between cryptographic security and computational power. The results, as visualized through Python’s data processing and graphing capabilities, underline a fundamental cryptographic principle: the security of encryption protocols like Diffie-Hellman scales exponentially with key size. This is exemplified by the dramatic increase in computation times on all systems as the prime number bit size grows.

The data further emphasizes the importance of optimizing both hardware and software configurations to achieve efficient cryptographic operations. Systems equipped with processors that are optimized for single-threaded tasks, like the M1 chip in the MacBook Pro, can offer substantial performance benefits, which are crucial in time-sensitive encryption tasks. However, this study also acknowledges the limitations posed by

small sample sizes and the inherent randomness in prime number generation, suggesting that more extensive testing is needed to draw definitive conclusions.

While the study's ten-trial average provides only a glimpse into performance differences, the data confirms the expected exponential increase in computation time with larger prime numbers. A more comprehensive trial range would better represent performance nuances. Nonetheless, the trend is clear: larger key sizes significantly enhance the security of cryptographic exchanges.

References

1. https://www.sciencedirect.com/topics/computer-science/hellman-algorithm