College of Computing

Key exchange protocols and Public Key Infrastructure

A Study on efficient key exchange algorithms, their weaknesses and solutions to mitigate them

Report Submitted by:
Arnaoui Basma
El-Amrani Soufiane
Maftah Mahmoud

Course:
Cryptography

Supervisor:
Abdulsalam Yunusa

Date:
January 7, 2025

Contents

1 Introduction **2**

2 Primality Testing **3**
2.1 Fermat's Primality Test . . . 3
2.2 Miller-Rabin Primality Test . . . 3
2.3 AKS Primality Test . . . 3
2.4 Comparison of Primality Tests . . . 3

3 The discrete log problem **3**
3.1 Shanks' Baby-Step Giant-Step Algorithm . . . 3

4 Large Prime Generation with Miller-Rabin **4**
4.1 Introduction . . . 4
4.2 Miller-Rabin–Based Prime Generation . . . 4

5 Benchmarking Prime Generation **5**
5.1 Key-Size Benchmark . . . 5

6 Diffie-Hellman Key Exchange **7**
6.1 Motivation and Role in Cryptography . . . 7
6.2 Protocol Description . . . 7
6.3 Security Considerations and Vulnerabilities . . . 7
6.4 Variants of Diffie-Hellman . . . 7
6.4.1 Elliptic Curve Diffie-Hellman (ECDH) . . . 7
6.4.2 Authenticated Key Exchange (e.g., *Station-to-Station*, SIGMA, etc.) . . . 7

7 Security Analysis and key considerations **8**
7.1 Strengths of Diffie-Hellman . . . 8
7.2 Known Vulnerabilities . . . 8
7.3 Best Practices . . . 8
7.4 Conclusion . . . 8

8 Certificate Authorities and Trust Models **9**
8.1 Trust Visualization Tool . . . 9
8.1.1 Strict Hierarchy Model . . . 9
8.1.2 Web Browser Model . . . 9
8.1.3 Pretty Good Privacy Web of Trust . . . 10

1 Introduction

Secure communication over untrusted networks is at the heart of modern information exchange, whether for e-commerce, secure messaging, or government communications. Cryptographic protocols enable two or more parties to establish secure channels, ensuring confidentiality, authenticity, and integrity of messages. Among these protocols, *key exchange* stands out as a critical operation: it allows participants to agree on secret keys even if they have never communicated or shared any secret information before.

This report explores classical and modern **key exchange techniques**, analyzing their theoretical foundations, potential weaknesses, and practical attacks. We also consider **mitigation strategies** against these vulnerabilities to reinforce the security of cryptographic systems.

Scope and Objectives:

- **Mathematical Foundations.** We review the necessary mathematical tools, including *primality testing*, *integer factorization* algorithms, and related number-theoretic concepts. These topics are crucial to understanding why certain public-key techniques (such as RSA, Diffie–Hellman) are considered secure under hardness assumptions.
- **Public Key Cryptography & Infrastructure.** We explain how *public key* systems work in practice, the concept of certificates, certificate authorities, and the broader *Public Key Infrastructure* (PKI).
- **Comparisons of Algorithms.** We compare various algorithms (both for testing primality and for factoring integers) to illustrate how different cryptographic schemes rely on the presumed intractability of specific number-theoretic problems.
- **Practical Demonstration.** The report culminates in a *simulation application* that demonstrates the functioning of various trust models, including the Strict Hierarchy, Web Browser, and PGP Web of Trust. Users can interact with the application to visualize trust relationships, query trust paths, and explore the practical implications of each model in real-world scenarios.

By examining key exchange protocols and public-key mechanisms under attack scenarios, we aim to provide a comprehensive perspective on both their theoretical underpinnings and real-world implementations. The ultimate goal is to emphasize the vital importance of cryptographic agility, where system designers and practitioners must remain vigilant against emerging cryptanalytic methods and advanced attackers.

2 Primality Testing

Primality testing is a fundamental problem in computational number theory and has significant applications in cryptography, particularly in key generation for public-key systems like RSA. This section explores various algorithms for primality testing, categorized into deterministic and probabilistic methods, detailing their working principles, complexity, and use cases.
Code snippets for all primality checking algorithms are to be found on our github repo.

2.1 Fermat's Primality Test

Fermat's primality test is based on Fermat's Little Theorem, which states that if n is prime and a is an integer such that $1 \leq a < n$, then $a^{n-1} \equiv 1 \mod n$.
Type: Probabilistic
Complexity: $O(k \cdot \log^2 n)$, where k is the number of iterations.

2.2 Miller-Rabin Primality Test

The Miller-Rabin test improves upon Fermat's test by repeatedly testing the compositeness of n using modular arithmetic.
Type: Probabilistic
Complexity: $O(k \cdot \log^3 n)$, where k is the number of iterations.

2.3 AKS Primality Test

The AKS primality test is a deterministic, polynomial-time algorithm that verifies the primality of a number without probabilistic assumptions.
Type: Deterministic
Complexity: $O(\log^6 n)$

2.4 Comparison of Primality Tests

We present here a more comprehensive list of primality testing algorithms for curious readers.

Algorithm	**Type**	**Complexity**	**Best Use Case**
Trial Division	Deterministic	$O(\sqrt{n})$	Small numbers
Fermat's Test	Probabilistic	$O(k \cdot \log^2 n)$	Quick checks, non-critical tasks
Miller-Rabin	Probabilistic	$O(k \cdot \log^3 n)$	Cryptographic applications
AKS Test	Deterministic	$O(\log^6 n)$	Theoretical guarantees of primality
Lucas-Lehmer	Deterministic	$O(p^2)$	Mersenne primes

Table 1: Comparison of Primality Testing Algorithms

3 The discrete log problem

3.1 Shanks' Baby-Step Giant-Step Algorithm

Shanks' algorithm, also known as the Baby-Step Giant-Step algorithm, is an efficient method to solve the discrete logarithm problem in groups of finite order. It has a time complexity of $O(\sqrt{n})$, where n is the order of the group (more details about the algorithm implementation can be found in our github repo feel free to read it).

4 Large Prime Generation with Miller-Rabin

4.1 Introduction

Generating large prime numbers efficiently is a foundational task in modern cryptography. One popular approach is to use **probabilistic primality tests** like *Miller-Rabin*, which provide a high level of confidence about a candidate being prime, yet execute in polynomial time with respect to the number of bits. In this document, we explore how to implement prime generation using Miller-Rabin and propose a few alternative strategies (like the Sieve of Eratosthenes and a hybrid Fermat+Miller-Rabin approach).

4.2 Miller-Rabin–Based Prime Generation

In this section we will reuse the Miller-Rabin probabilistic primality test for arbitrary numbers,

```
import random

def is_prime(n, k=10):
    """
    Miller-Rabin primality test to check if a number is prime.
    :param n: The number to check for primality.
    :param k: Number of iterations for accuracy.
    :return: True if n is probably prime, False otherwise.
    """
    # Handle simple cases
    if n <= 1:
        return False
    if n <= 3:
        return True
    if n % 2 == 0:
        return False

    # Write n-1 as 2^r * d
    r, d = 0, n - 1
    while d % 2 == 0:
        r += 1
        d //= 2

    # Miller-Rabin test
    for _ in range(k):
        a = random.randint(2, n - 2)
        x = pow(a, d, n)  # Compute a^d % n
        if x == 1 or x == n - 1:
            continue
        for _ in range(r - 1):
            x = pow(x, 2, n)
            if x == n - 1:
                break
        else:
            return False
    return True
```

Explanation

- **Purpose:** This function performs the *Miller-Rabin test*, a fast *probabilistic* algorithm, to check if an integer n is prime.
- **Key Idea:** We express $n-1$ as $2^r \times d$. Then, for several random bases a, we test whether $a^d \equiv 1 \pmod{n}$ or $a^{2^j d} \equiv -1 \pmod{n}$ for some j in $\{0, \dots, r-1\}$. Failing these conditions for all chosen bases indicates compositeness.

- **Complexity:** Each round of Miller-Rabin uses fast modular exponentiation, typically $O(\log n)$ multiplications per exponentiation. Repeated k times, the test remains efficient even for large n.

Now we will use the function above to test primality efficiently for random number of the specified size.

```python
def generate_large_prime(bits):
    """
    Generate a large prime number with the specified number of bits.
    :param bits: The bit-length of the prime number.
    :return: A large prime number.
    """
    while True:
        # Generate random candidate of 'bits' bits
        # Ensure the highest bit (bits-1) and the lowest bit (0) are set to 1
        candidate = random.getrandbits(bits) | (1 << bits - 1) | 1

        # Check primality with is_prime
        if is_prime(candidate):
            return candidate
```

Explanation

- **Purpose:** This function uses `is_prime` (Miller-Rabin) to find a random large prime of exactly `bits` bit-length.
- **Bit Guarantee:** `(1 << bits - 1)` ensures the most significant bit is set to 1, and `| 1` ensures the least significant bit is 1 (i.e., the number is odd).
- **Efficiency:** The probability that a random `bits`-bit integer is prime is about $1/\ln(2^{\texttt{bits}}) \approx 1/(\texttt{bits} \times \ln(2))$. Thus, on average, we do not need many candidates before finding a prime.

5 Benchmarking Prime Generation

In this section, we present two benchmarks evaluating the performance of our prime-generation method, specifically how it scales with different key sizes. We base our benchmarks on the Miller-Rabin primality test and measure both (i) the overall number of primes generated over time and (ii) the average time required to generate a single prime.

5.1 Key-Size Benchmark

We compare the generation of 256-bit, 512-bit, and 1024-bit primes using the Miller-Rabin test. Specifically, we run each configuration (key size) for a fixed duration (e.g., 20 s) and repeatedly generate primes as quickly as possible. The pseudocode below outlines the procedure:

1. **Initialize:** Set the key sizes to test, such as {256, 512, 1024} bits.
2. **For each key size:**
   (a) Record the start time.
   (b) Generate primes until the fixed duration expires.
   (c) Keep track of how many primes were produced.
   (d) Log the number of generated primes at regular intervals (e.g., every 5 s).

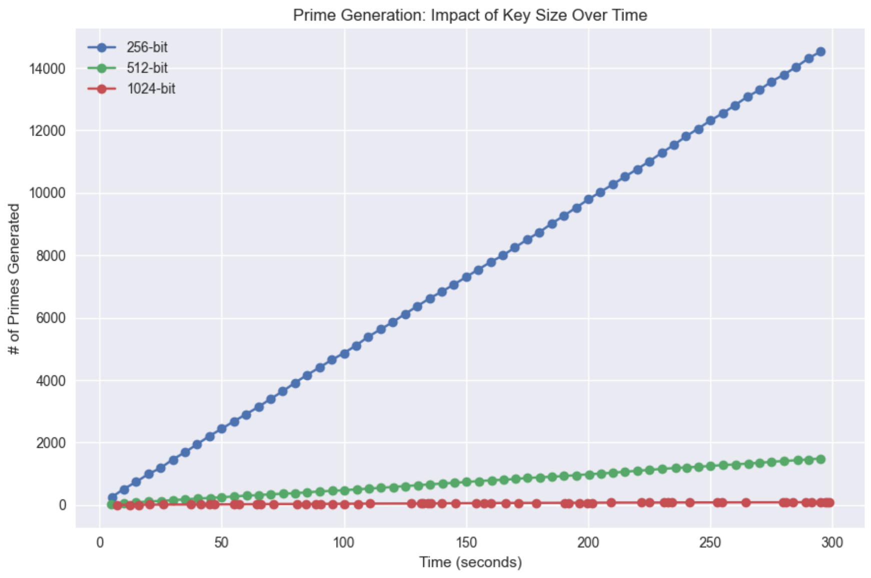

Figure 1: Number of primes generated vs. time for 256-bit, 512-bit, and 1024-bit key sizes. Note how the slope of each curve decreases for larger key sizes, as prime generation takes longer.

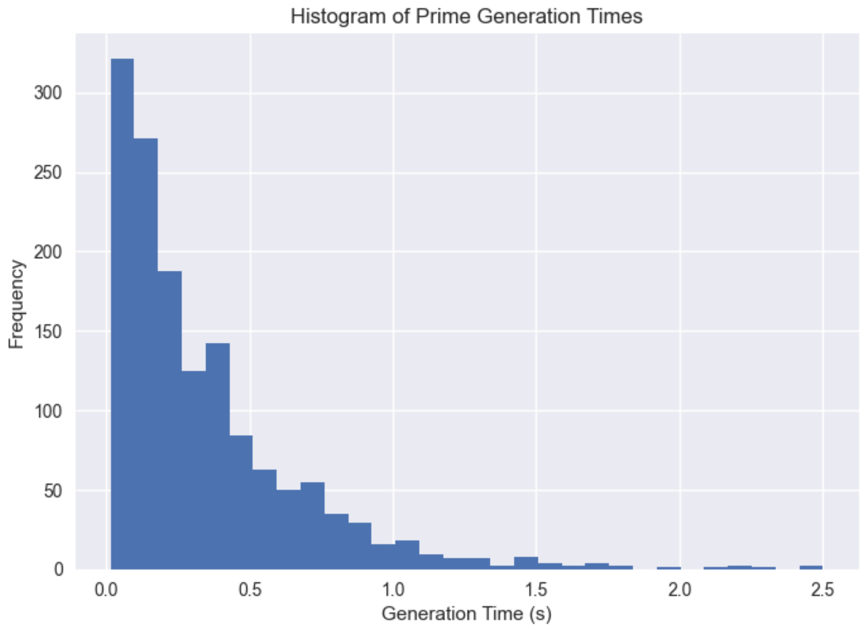

Figure 2: Average time (seconds) needed to generate a single prime for 256-bit, 512-bit, and 1024-bit key sizes. Larger key sizes require more computational effort per prime.

Figure 1 reveals that 256-bit primes are generated most rapidly; meanwhile, 1024-bit primes accumulate at a noticeably slower rate, demonstrating the added computational cost. Figure 2 further quantifies this difference, as the average generation time per prime increases with key size. These results confirm the expected scaling properties of prime generation using the Miller-Rabin test with different key lengths.

6 Diffie-Hellman Key Exchange

The **Diffie-Hellman (DH)** key exchange protocol is a seminal method in public-key cryptography that allows two parties (traditionally called **Alice** and **Bob**) to establish a shared secret key over an insecure communication channel. It was invented by Whitfield Diffie and Martin Hellman in 1976 and represents one of the first practical implementations of public key exchange.

6.1 Motivation and Role in Cryptography

Prior to Diffie-Hellman, key exchange relied on *pre-shared* secrets or trusted couriers to convey symmetric keys. This was impractical for large-scale communications (e.g., the internet). With the advent of **Diffie-Hellman**, two parties who have *never met* (and share no prior secrets) can securely agree on a *symmetric key* used to encrypt subsequent communications.

6.2 Protocol Description

Let p be a large prime and g be a generator (or primitive root modulo p). Both p and g are *publicly known.*

1. **Alice picks a secret** a (uniformly at random) in the range 1 to $p - 1$.
2. **Alice computes and sends** $A = g^a \bmod p$ *to Bob.*
3. **Bob picks a secret** b (also uniform in 1 to $p - 1$).
4. **Bob computes and sends** $B = g^b \bmod p$ *to Alice.*
5. **Alice computes** the *shared secret* as $s = B^a \bmod p = (g^b)^a = g^{ab} \bmod p$.
6. **Bob computes** the *shared secret* as $s = A^b \bmod p = (g^a)^b = g^{ab} \bmod p$.

Since $g^{ab} \bmod p$ is the same for both parties, Alice and Bob have established a *shared secret key s.* An eavesdropper who sees A and B but does not know a or b faces the *Discrete Logarithm Problem (DLP)* to recover a or b.

6.3 Security Considerations and Vulnerabilities

Discrete Logarithm Problem (DLP): The security of Diffie-Hellman primarily depends on the hardness of the DLP. If one can efficiently solve $g^x \bmod p$ for x, they can recover a or b and break the protocol.

Man-in-the-Middle (MITM): The basic DH protocol *does not* authenticate the participants. An attacker can impersonate Alice to Bob, and Bob to Alice, establishing two keys: one shared with Alice and another with Bob. *Both* parties think they are talking to each other but are, in fact, talking to the attacker. **Solution:** Use authenticated protocols (*e.g.*, sign the DH parameters, use certificates, etc.).

6.4 Variants of Diffie-Hellman

6.4.1 Elliptic Curve Diffie-Hellman (ECDH)

Instead of working in the multiplicative group modulo p, **ECDH** uses points on an elliptic curve. This typically offers *shorter key lengths* and faster computations for comparable security levels (e.g., a 256-bit elliptic-curve group vs. a 3072-bit traditional modulo prime group).

6.4.2 Authenticated Key Exchange (e.g., *Station-to-Station*, SIGMA, etc.)

By incorporating digital signatures or other authentication mechanisms, the *man-in-the-middle* vulnerability is mitigated. In practice, protocols like TLS use ephemeral Diffie-Hellman with X.509 certificates (RSA or ECDSA) to authenticate the server (and optionally the client).

7 Security Analysis and key considerations

7.1 Strengths of Diffie-Hellman

The Diffie-Hellman (DH) key exchange protocol derives its security from the *Discrete Logarithm Problem (DLP)* in large finite fields, which is computationally hard to solve. Using a large prime p and a generator g, the value $g^a \bmod p$ does not reveal a.

When paired with *ephemeral keys* (keys generated per session), DH ensures *forward secrecy*, meaning past session keys cannot be decrypted even if a long-term private key is compromised.

7.2 Known Vulnerabilities

Man-in-the-Middle (MITM) Attack. Unauthenticated DH is vulnerable to MITM attacks, where an adversary intercepts and replaces public keys during exchange.

Mitigation: Use digital signatures or certificates to authenticate public keys.

Small Subgroup Attacks. Poorly chosen primes p or generators g may allow an attacker to force the exchange into a smaller subgroup, making DLP easier to solve.

Mitigation: Use safe primes (e.g., RFC 3526) to ensure large subgroup orders.

Logjam Attack. Reusing small or weak primes (e.g., 512-bit) enables attackers to break exchanges using precomputation.

Mitigation: Use at least 2048-bit primes and reject weak parameters.

Side-Channel Attacks. Timing or power analysis during modular exponentiation can leak private key information.

Mitigation: Implement constant-time algorithms and blinding techniques.

7.3 Best Practices

- Use large, standardized primes ($\geq$ 2048-bit) from sources like RFC 3526.
- Employ digital signatures or certificates for authentication to prevent MITM attacks.
- Enable forward secrecy with ephemeral keys (DHE).
- Use constant-time algorithms to resist side-channel attacks.

7.4 Conclusion

Diffie-Hellman remains a robust protocol when properly parameterized with secure primes, authenticated exchanges, and ephemeral keys. By adhering to best practices, the protocol can withstand modern adversarial threats.

8 Certificate Authorities and Trust Models

Certificate Authorities (CAs) are trusted entities that issue digital certificates, enabling secure communication and authentication across the internet. Trust relies on a hierarchical chain where root CAs serve as trust anchors pre-installed in operating systems and web browsers.

8.1 Trust Visualization Tool

We have developed an interactive visualization tool for exploring trust relationships between entities across different trust models. The tool allows users to query and verify trust relationships between any two entities, providing a clear visual representation of trust paths and relationships. Try our implementation at: `https://certificateauthority.pythonanywhere.com/strict/`

8.1.1 Strict Hierarchy Model

A traditional tree-like structure where trust flows downward from a root CA. Users can query if two entities trust each other by tracing paths through their common trusted root. As shown in Figure 3, the strict hierarchy model is visualized for clarity.

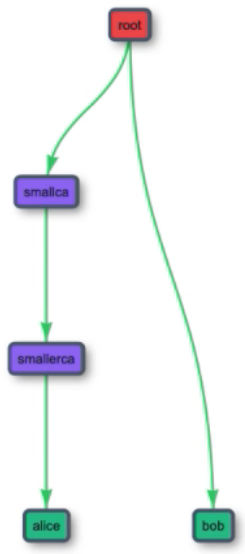

Figure 3: Strict Hierarchy Model Implementation

8.1.2 Web Browser Model

Reflecting modern browsers' approach with multiple root CAs, the web browser trust model allows cross-certification paths. Figure 4 illustrates how trust relationships can be verified between different entities under distinct root authorities.

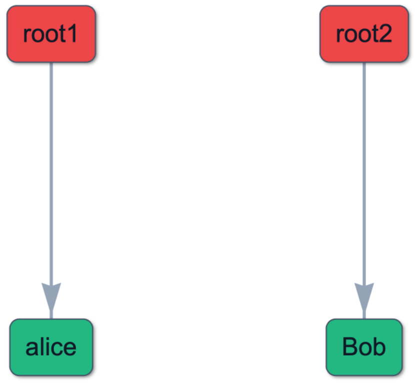

Figure 4: Web Browser Trust Model Implementation

8.1.3 Pretty Good Privacy Web of Trust

The PGP Web of Trust is a decentralized approach, shown in Figure 5. Users can visualize and query complex peer-to-peer trust relationships, including complete, marginal, and indirect trust paths between any two entities.

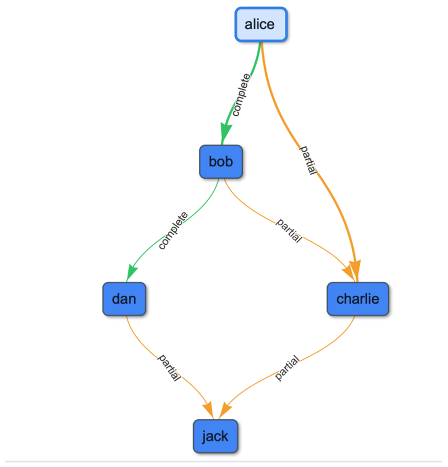

Figure 5: PGP Web of Trust Implementation

Model	Advantages	Challenges
Strict Hierarchy	Clear trust paths	Single point of failure
Web Browser	Wide compatibility	Complex decisions
PGP Web of Trust	Decentralized	Bootstrap complexity

Table 2: Comparison of Trust Models