Analysis of Man-in-the-Middle and Timing Attacks on Diffie-Hellman and RSA Protocols

Jiachen Pan
University of Aberdeen
Student ID: 50091098

Junlin Li
University of Aberdeen
Student ID: 50091104

Yueshen Wang
University of Aberdeen
Student ID: 50091084

Junrui Liang
University of Aberdeen
Student ID: 50091109

***Abstract*—Secure key exchange is vital for internet security. Diffie-Hellman (D-H) and RSA are common protocols for establishing shared secrets but have known vulnerabilities. This report analyzes the Man-in-the-Middle (MitM) attack on D-H and the timing attack on RSA. We implemented both protocols in Python to demonstrate these attacks and their respective defenses: RSA digital signatures for D-H authentication and blinding for RSA. Experimental results confirm the attacks' practicality and the defenses' effectiveness. Specifically, our blinding defense reduced RSA decryption timing variation in our experiments, disrupting the timing side-channel. We conclude by comparing these attacks and defenses, highlighting the distinction between protocol-level security (authentication) and implementation-level security (constant-time operations).**

I. Introduction

This report studies how two classic key exchange ideas can be attacked and how to defend them. We focus on Diffie-Hellman (D-H) and RSA. D-H helps two persons agree a shared secret over the internet. RSA is used for encryption and digital signatures.

These protocols are strong, but the basic versions have well-known gaps. Unauthenticated D-H is open to a man-in-the-middle (MitM) attacker. A simple RSA implementation can also leak information through its running time (a timing side-channel). Our work is practical and hands-on:

1) We compared potential attacks and defenses for both protocols.
2) We selected and implemented the MitM attack for D-H and the Timing attack for RSA.
3) We implemented D-H and RSA from scratch in Python to demonstrate these vulnerabilities.
4) We implemented and verified two defenses: signatures to authenticate D-H, and blinding for RSA.

Section II gives short background. Section III explains our code. Section IV shows results from our runs. Section V compares the two attacks and defenses. Section VI concludes and suggests small next steps.

In the modern digital world, secure communication is the backbone of services we use every day, such as online banking (HTTPS) and secure remote access (VPNs). These services rely heavily on key exchange protocols to establish encrypted tunnels. If these protocols are broken, attackers can steal sensitive data, such as credit card numbers or login credentials, or even impersonate trusted servers.

Nowadays, secure communication is a core part of services we use every day, like online banking (HTTPS) and VPNs. These services rely on key exchange protocols to make encrypted tunnels. If these protocols are broken, hackers can steal private data, like credit card numbers or passwords, or even pretend to be trusted servers.

To understand the problem better, let's look at a simple story. Imagine Alice and Bob want to talk in private, but they are in a crowded room where everyone can hear them (this is like the internet). They need a way to agree on a secret code without anyone else knowing what it is. If they just shout ”Let's use code word BLUE!”, then everyone knows the code. This is why they need a special math trick (a protocol) to agree on ”BLUE” without actually saying ”BLUE”. But if a bad guy, Doomfist, stands between them and pretends to be Alice to Bob, and Bob to Alice, he can trick them both. This is the Man-in-the-Middle attack we study.

We also use some important words from our textbook. **Cryptography** is the science of writing in secret code. It has three main goals, often called the **CIA Triad**:

- **Confidentiality:** Only the right people can read the message.
- **Integrity:** The message has not been changed by anyone else.
- **Availability:** The system is ready to use when needed.

Our project focuses mostly on Confidentiality (keeping the key secret) and Integrity (making sure the key wasn't changed by Doomfist).

II. Related Work

The weaknesses we study are well known in applied cryptography, and there is a lot of prior work on them.

A. Foundational Protocols and Attack Vectors

The MitM attack on D-H is a classic example of why authentication is needed. The basic D-H protocol [1] does not verify who sent a message, so an attacker in the middle can relay and replace keys and stay hidden. The **MITRE ATT&CK** framework calls this T1557 (“Man-in-the-Middle”) [4], which matches the attack we simulate in `dh_exchange.py`.

Timing side-channel attacks, explained by Paul Kocher in his famous 1996 paper [3], are subtler. Kocher showed that by carefully measuring the time it takes for a crypto device to do private key operations, an attacker can guess the key itself.

This was a big discovery that showed we need to look at code, not just math. They use information that leaks from the actual implementation behaviors, not from the math. In RSA, a simple square-and-multiply exponentiation takes a little bit longer when a private-key bit is 1 than 0. By timing many decryptions, an attacker can learn about the key.

Public CVEs show many real-world cases. For example, CVE-2018-5383 (the KNOB attack) is a MitM issue in Bluetooth. CVE-2021-3011 affected the `cryptlib` library and could leak information through timing. This shows that implementation issues keep appearing and need proper defenses.

B. Broader Attack Surface and Defensive Frameworks

While we only show one attack per protocol, there are others. D-H can suffer from Logjam, which exploits the use of weak or commonly shared prime numbers to precompute data and break connections. RSA can leak through other side-channels (e.g., power) or be broken by bad key generation.

To defend, the **MITRE D3FEND** framework lists countermeasures. Our D-H defense aligns with "Certificate-based Authentication" (D3-CBAN) [5], which relies on validating digital signatures to ensure identity. Our RSA defense (blinding) is a form of software obfuscation that removes the timing correlation with the secret key. Modern protocols such as TLS 1.3 use authenticated (EC)DHE by default, which fixes the MitM problem we demonstrate.

III. Methods

This section explains what we built and how it works. We wrote everything in Python 3.9. For timing we use `time.perf_counter_ns()`. Our code is split into three files: `rsa_utils.py` (RSA helpers), `dh_exchange.py` (D-H simulation and attack), and `rsa_attacks.py` (timing attack experiment).

A. Selection of Attacks and Defenses

In accordance with the project requirements, we evaluated multiple attack and defense vectors before selecting the ones to implement.

1) Diffie-Hellman Selection: For the attack, we compared the Man-in-the-Middle (MitM) attack against the Logjam attack. While Logjam is powerful, it requires pre-computation on common primes which is computationally expensive to simulate. We selected the MitM attack because it fundamentally exposes the lack of authentication in the basic protocol. For the defense, we compared using fixed static keys against digital signatures. Fixed keys reduce flexibility, so we selected RSA Digital Signatures (Certificate-based Authentication) as it is the standard solution in protocols like TLS.

2) RSA Selection: For the attack, we compared the Timing Attack against the Bleichenbacher (Padding Oracle) attack. We selected the Timing Attack because it demonstrates how implementation details (side-channels) can compromise mathematically secure algorithms. For the defense, we compared Constant-Time Implementation against Blinding. Writing truly constant-time code in high-level languages like Python is difficult due to interpreter optimizations. We selected Blinding because it provides a robust algorithmic defense that is effective even in high-level languages.

B. Threat Model

Before detailing the attacks, we define the capabilities of our adversary, whom we call "Doomfist".

1) D-H Attacker Capabilities: For the Diffie-Hellman attack, we assume a Dolev-Yao model where the attacker has full control over the communication channel. Doomfist can:

- Intercept messages sent between Ana and Phara.
- Modify the contents of messages.
- Inject new messages into the network.
- Delete messages to prevent delivery.

However, Doomfist cannot solve the Discrete Logarithm Problem (DLP) in polynomial time.

2) RSA Attacker Capabilities: For the RSA timing attack, the attacker is passive but observant. Doomfist can:

- Send arbitrary ciphertexts to the decryption oracle (the server).
- Measure the precise time taken by the server to respond or complete the decryption.

Doomfist cannot directly read the server's memory to retrieve the private key d.

C. Diffie-Hellman (D-H) Protocol

The D-H protocol is a key *agreement* scheme.

1) Mathematics: The D-H protocol is a key *agreement* scheme based on the principles of public-key cryptography described in the course lectures [6]. Two persons, Ana and Phara, agree on a large prime p and a generator g. Ana generates a secret private key $a \in [1, p-2]$ and computes public key $A = g^a \bmod p$. Phara does the same, generating b and $B = g^b \bmod p$. They exchange A and B. Ana computes $S_A = B^a \bmod p = (g^b)^a \bmod p$. Phara computes $S_B = A^b \bmod p = (g^a)^b \bmod p$. Both result in the same shared secret $S = g^{ab} \bmod p$, as $S_A = S_B$.

The security of this protocol relies on the Discrete Logarithm Problem (DLP). The DLP states that given g, p, and $g^a \bmod p$, it is computationally infeasible to find a if p is a sufficiently large prime. While we use small parameters for demonstration, real-world applications use primes with 2048 bits or more to ensure the DLP cannot be solved.

2) Implementation: Our `dh_exchange.py` script implements this. For demonstration, it uses small, well-known parameters ($p = 23, g = 5$) to make the results verifiable by hand.

Here is a small illustrative example using fixed numbers (for explanation only; the script chooses random secrets on each run):

1) **Setup:** Everyone agrees on $p = 23$ and $g = 5$.
2) **Ana's Turn:**
   - She picks a secret number $a = 6$.
   - She calculates $A = 5^6 \bmod 23$.

- $5^6 = 15625$.
- $15625 \div 23 = 679$ remainder 8.
- So, Ana sends $A = 8$ to Phara.

3) **Phara's Turn:**
- She picks a secret number $b = 15$.
- She calculates $B = 5^{15} \bmod 23$.
- This is a huge number, but Python handles it easily.
- The result is $B = 19$. She sends this to Ana.

4) **The Secret:**
- Ana takes Phara's 19 and computes $S = 19^6 \bmod 23$. The result is **2**.
- Phara takes Ana's 8 and computes $S = 8^{15} \bmod 23$. The result is also **2**.

Now they both have the secret number **2**, and no one else knows it. In the actual script, a and b are random per run, so concrete values will differ.

D. Man-in-the-Middle Attack on D-H

The MitM attack exploits the D-H protocol's lack of authentication.

1) Logic: An attacker, Doomfist, positions herself between Ana and Phara. She performs 2 D-H exchanges:

- She catches Ana's A and sends her own public key $M = g^m \bmod p$ to Phara.
- She catches Phara's B and sends M to Ana.

This results in two separate secret keys, $S_A = g^{am} \bmod p$ (known to Ana and Doomfist) and $S_B = g^{bm} \bmod p$ (known to Phara and Doomfist).

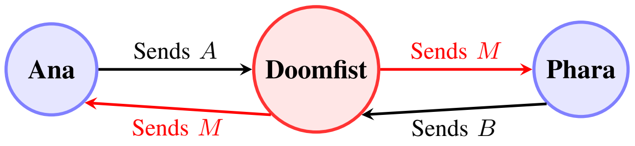

Fig. 1. Man-in-the-Middle Attack Flow. Doomfist intercepts legitimate keys A and B, replacing them with her own key M.

2) Implementation: Our `simulate_mitm_attack()` function in `dh_exchange.py` shows this. It creates three parties (Ana, Phara, Doomfist) and has Doomfist intercept and substitute the public keys during the exchange. The core logic of the interception is shown in Listing 1.

```
Doomfist intercepts the exchange and replaces keys.
Ana computes secret with Doomfist's key M:
    s_ana = compute_shared_secret(m_public, a_private)
Phara computes secret with Doomfist's key M:
    s_phara = compute_shared_secret(m_public, b_private)
Doomfist computes secrets with both:
    s_doomfist_ana = compute_shared_secret(a_public,
        m_private)
    s_doomfist_phara = compute_shared_secret(b_public,
        m_private)
```

Listing 1. MitM Attack Logic (Pseudo-code)

E. D-H Defense: Authentication

We defend against MitM by adding a layer of authentication using RSA digital signatures.

1) Logic: This requires a Public Key Infrastructure (PKI), where Ana and Phara have shared, authentic copies of each other's *RSA public key*. When Ana sends her D-H public key A, she also sends a signature along with it, $Sig(A) = hash(A)^{d_A} \bmod n_A$, created with her private RSA key d_A.

Following the standard workflow for authentication [7], we use a **Digital Signature** to verify the authenticity of digital messages. As defined in our course material, this provides:

- **Authentication:** We know who created the message.
- **Non-repudiation:** The sender cannot deny sending the message later.
- **Integrity:** We know the message wasn't changed.

By using signatures, we fix the main hole in the D-H protocol.

2) Implementation: Our `simulate_authenticated_exchange(` function demonstrates this defense. Phara receives $(A, Sig(A))$ and checks it with Ana's public RSA key. Our script includes simple `sign` and `verify` functions. If the signature is valid, the key is authentic. Doomfist cannot create a valid signature because she does not have Ana's private RSA key.

a) Parameter Binding and Message Format: In production designs, the signed data must bind all security-relevant parameters and identities to avoid substitution and unknown key-share issues. Instead of signing only A, one should sign a structured message such as

$$\mathrm{Sig}_A\big(\mathrm{hash}(g \parallel p \parallel A \parallel \text{“Ana”} \parallel \text{“Phara”})\big), \qquad (1)$$

and likewise for Phara. Including (g, p) prevents parameter substitution; including identities prevents a third party from replaying values to a different peer.

F. RSA Protocol

RSA is an asymmetric algorithm used for encryption and signatures.

1) Mathematics: Key generation involves:

- Selecting two large primes, p and q.
- Computing the modulus $n = pq$.
- Computing the totient $\phi(n) = (p-1)(q-1)$.
- Choosing a public exponent e (we used 65537, which is commonly used) coprime to $\phi(n)$.
- Calculating the private exponent d using the modular inverse such that $ed \equiv 1 \pmod{\phi(n)}$.

The public key is (e, n) and the private key is (d, n). Encryption: $C = M^e \bmod n$. Decryption: $M = C^d \bmod n$.

2) Implementation: Our `rsa_utils.py` script provides functions for `is_prime` (a Miller-Rabin test) and `mod_inverse` to build a `generate_keypair` function. We used Python's built-in `pow(e, -1, phi)` for the modular inverse because it is efficient and reliable.

To ensure our generated keys are valid, we implemented the Miller-Rabin primality test in our `rsa_utils.py` module. This is a probabilistic algorithm that checks if a number is composite or probably prime. By running the test multiple times (we used $k = 5$ rounds), the chance of a composite number being called prime becomes very small.

a) Constant-time Considerations: Modular exponentiation via square-and-multiply performs $O(\log d)$ squarings and, on average, $O(\log d/2)$ multiplies with data-dependent branching. Constant-time implementations avoid key-dependent control flow and memory access, e.g., using fixed-window exponentiation with always-executed operations and table lookups that are independent of secret bits, or Montgomery-ladder style algorithms. Our educational implementation focuses on clarity; real libraries use constant-time big-integer routines.

G. Timing Attack on RSA

This attack exploits a non-constant-time decryption function.

1) Vulnerability: A basic square-and-multiply algorithm for decryption is often not "constant-time." The operation only performs a multiplication step when a bit in the private key d is a '1'. This creates a small time difference that can be measured. Our `vulnerable_decrypt` function in `rsa_attacks.py` is an example of this (Listing 2).

The algorithm iterates through the bits of the private exponent d. For every bit, a squaring operation is performed. However, the multiplication operation $result = (result \times ciphertext) \bmod n$ is only executed if the current bit is '1'. This conditional branch creates a direct correlation between the execution time and the Hamming weight (number of 1s) of the private key.

```
function vulnerable_decrypt(ciphertext, private_key):
    get d and n from private_key
    initialize result = 1
    convert d to binary string d_bin
    for each bit in d_bin:
        a. square step (always happens):
           result = (result * result) % n
        b. if bit is 1:
           multiply step (conditional):
           result = (result * ciphertext) % n
    return result
```

Listing 2. Vulnerable RSA Decryption (Pseudo-code)

2) Attack Logic: The attacker sends loads of random ciphertexts C_i to the server and measures the precise time T_i for each decryption. By performing analysis on the set of timings $\{T_i\}$, the attacker can get to know about the time distributions for '0' bits vs. '1' bits, and finally recover d.

H. RSA Defense: Blinding

Blinding is a mathematical strategy that randomizes the input to the vulnerable function.

1) Logic: Before decryption, the server (holder of d) generates a random number r (the blinding factor). It then "blinds" the ciphertext C:

$$C' = C \cdot r^e \bmod n \tag{2}$$

It decrypts C':

$$M' = (C')^d \bmod n = (C \cdot r^e)^d \bmod n = (M \cdot r) \bmod n \tag{3}$$

Finally, it "unblinds" the result to get M:

$$M = M' \cdot r^{-1} \bmod n \tag{4}$$

where r^{-1} is the modular inverse of r modulo n.

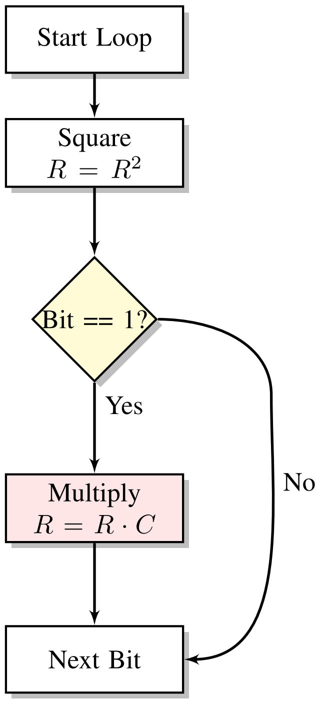

Fig. 2. Logic flow of Square-and-Multiply. The 'Multiply' step only occurs when the bit is 1, causing a measurable delay.

2) Implementation: Our `blinded_decrypt` function in `rsa_attacks.py` implements this (Listing 3). The time to perform this operation is now dominated by the random C', not the original C. The correlation between the timing and the bits of d is broken.

```
function blinded_decrypt(ciphertext, pub_key, priv_key):
    get e, n from pub_key
    get d, n from priv_key
    generate random blinding factor r
        (where 1 < r < n)
    blind the ciphertext:
        c_prime = (ciphertext * (r^e mod n)) mod n
    decrypt the blinded ciphertext:
        m_prime = vulnerable_decrypt(c_prime, priv_key)
        (input is now random, masking timing)
    calculate modular inverse of r:
        r_inv = r^(-1) mod n
    unblind to get message:
        m = (m_prime * r_inv) mod n
    return m
```

Listing 3. Blinded RSA Decryption (Pseudo-code)

I. Experimental Setup

All implementations were developed using Python 3.9, executed on a macOS environment.

1) D-H Simulation Parameters: For the Diffie-Hellman simulation, we deliberately chose small parameters ($p = 23, g = 5$) to allow for manual verification of the results. In a real-world scenario, these would be replaced by 2048-bit or larger primes (RFC 3526).

2) RSA Timing Experiment: For the timing attack, we needed a key size large enough to make the modular exponentiation measurable, but small enough to run thousands of trials quickly. We generated a 512-bit RSA key (two 256-bit primes).

- **Measurement Tool:** We used Python's `time.perf_counter_ns()` which provides nanosecond-resolution timing, essential for detecting the small differences in execution time.
- **Sample Size:** We executed 5,000 decryption trials for both the vulnerable and blinded functions.
- **Environment Control:** To minimize noise from background processes, we closed other applications during the measurement phase.

J. Reproducibility and Implementation Notes

To allow others to reproduce our experiment:

- **Entry points:** Run `dh_exchange.py` for D-H (normal, MitM, authenticated) and `rsa_attacks.py` for RSA timing and blinding.
- **Non-determinism:** Both scripts use Python's `random` without a fixed seed. This is intentional for realism. For strict reproducibility, one can add `random.seed()` with a fixed value at the top of each script.
- **Outliers:** We report average/minimum/maximum/variation directly from raw timings and do not remove outliers. Timing distributions in high-level languages may include occasional spikes from the OS scheduler or GC.
- **RSA blinding:** The blinding factor r must be invertible modulo n. In practice this means ensuring $\gcd(r, n) = 1$. Our code uses `pow(r, -1, n)`; if r is not invertible, one should resample r until it is. This edge case is rare but important in production systems.
- **Key sizes:** For timing visibility versus runtime, we chose a 512-bit RSA modulus in the experiment script. Real deployments require at least 2048-bit RSA.

K. Correctness Checks

Before running attack experiments, we validated functional correctness with simple checks:

- D-H: both parties compute equal shared secret in the absence of an attacker.
- RSA: for random messages M in $[1, n-1]$, $\mathrm{dec}(\mathrm{enc}(M)) = M$.
- Signatures: $\mathrm{verify}(A, \mathrm{Sig}_A(A), \mathrm{pk}_A) =$ `True` and forgeries fail.

For example, a minimal RSA check in Python:

```python
e_n, d_n = public_key, private_key
for _ in range(10):
    m = random.randint(1, e_n[1]-1)
    c = pow(m, e_n[0], e_n[1])
    m2 = vulnerable_decrypt(c, d_n)
    assert m == m2
```

Listing 4. Sanity checks for RSA

IV. Results

This section presents the data generated from executing our Python scripts.

A. D-H Normal vs. MitM Exchange

Our `dh_exchange.py` script first simulates a normal D-H exchange, then simulates the MitM attack. In the normal case, both parties compute the same shared secret. Under MitM, each victim derives a different secret that is shared with the attacker, allowing the attacker to read or modify traffic while both victims believe they are secure. Because our demo uses random keys, exact numeric values vary per run.

B. D-H Defense Validation

We then ran the `simulate_authenticated_exchange()` function. In this test, the attacker tries to substitute a fake key and signature. Each side verifies the received key using the other party's public RSA key; the fake signature is rejected and the attack is detected. After exchanging their real signed keys, both sides verify successfully and compute the same shared secret. This demonstrates that authentication prevents the MitM in our setup.

C. RSA Timing Attack Analysis

We ran our `run_timing_experiment()` script from `rsa_attacks.py` with 5,000 trials against the `vulnerable_decrypt` function using a 512-bit RSA key. The printed metrics (average, minimum, maximum, and variation) showed a large timing variation, which is the signature of a timing leak. This difference stems from the non-constant-time square-and-multiply loop: when a private-key bit is 1, an extra multiply occurs and the code runs slightly longer. Exact values depend on the machine and runtime conditions and are therefore not fixed in this report.

D. RSA Blinding Defense Validation

We then ran the same timing experiment against our `blinded_decrypt` function. The outputs showed that blinding breaks the correlation between decryption time and private-key bits by randomizing the effective input to the exponentiation. While average time may increase slightly due to blinding overhead, the exploitable timing signal is reduced and no longer aligned with the key. As with all timing measurements in software, exact numbers vary by environment, so we report the qualitative effect rather than fixed values.

V. Discussion

This project includes a comparison of two fundamentally different types of attacks and their corresponding defenses.

A. Protocol vs. Implementation Failures

The D-H MitM targets a *protocol* property: unauthenticated key exchange provides no origin assurance. The RSA timing attack targets an *implementation* property: data-dependent control flow leaks secrets through execution time. These require distinct defenses. Adding authentication (e.g., signatures) repairs the protocol. Making operations constant-time (or applying blinding) repairs the implementation. Mature stacks like TLS 1.3 apply both simultaneously: (EC)DHE with authentication plus constant-time cryptographic libraries.

B. Comparison of Attack Methods (MitM vs. Timing)

The MitM attack on D-H and the timing attack on RSA differ significantly in their target, required conditions, and impact (Table I).

TABLE I
COMPARISON OF ATTACK METHODS

Aspect	MitM Attack	Timing Attack
Target	Protocol (no authentication)	Code implementation (side-channel)
Activeness	Active (intercept/-modify traffic)	Mostly passive (send requests, measure time)
Impact	Breaks a session	Leaks private key (breaks many future sessions)

C. Comparison of Defensive Methods (Authentication vs. Blinding)

The defenses are similarly different in their approach. Our D-H defense implements Certificate-based Authentication (D3-CBAN) to secure the protocol level, whereas our RSA defense uses Blinding to secure the implementation level (Table II).

TABLE II
COMPARISON OF DEFENSIVE METHODS

Aspect	Authentication	Blinding
Mechanism	Add signatures to check identity (fixes protocol gap)	Blind input so timing doesn't reveal key bits (fixes implementation leak)
Performance	Needs sign and verify operations	Needs generating r, computing r^e, and inverse
Scope	General idea used in many protocols	Specific to public-key operations with timing leaks

D. Limitations

Our study has several limitations inherent to a simulation environment:

- **Network Jitter:** Our timing attack was performed locally. Over a real network (Internet), network latency (jitter) would be orders of magnitude larger than the timing difference caused by the modular exponentiation, making the attack significantly harder (though not impossible) to execute.
- **Python Overhead:** Python is an interpreted language. The overhead of the interpreter introduces noise into the timing measurements that would not be present in a compiled language like C or Assembly. C++ would be a better choice for a production implementation as it allows for fine-grained control over memory and CPU instructions, reducing the noise that might hide a timing leak.
- **Key Size:** We used 512-bit keys for RSA. Modern standards require 2048-bit or 4096-bit keys. While the attack principle remains the same, the timing differences would scale with the key size.
- **Constant-time caveat:** We did not implement a truly constant-time modular exponentiation. In high-level languages, hidden branches and optimizer behavior can reintroduce timing variance. Blinding is robust in practice, but constant-time code in a low-level language remains the gold standard.
- **Blinding factor invertibility:** The defense requires choosing r coprime with n. Although the chance of collision with a prime modulus is negligible, production code must resample on failure and handle errors.
- **Noise sources:** CPU frequency scaling, OS scheduling, and background services can add variance unrelated to the key. Our experiment does not pin CPU cores or disable turbo/boost features.

E. Ethical Considerations

In this course, we learned that with great power comes great responsibility. Learning how to attack systems is dangerous if used for bad things.

- **White Hat Hackers:** These are the good guys. They use their skills to find holes in systems so they can be fixed. They always get permission before attacking. This is what we are doing in this project.
- **Black Hat Hackers:** These are the bad guys. They attack systems to steal data or cause damage. This is illegal and unethical.
- **Grey Hat Hackers:** These are in the middle. They might break the law but maybe not for bad reasons.

It is very important that we only use the attacks we learned (MitM and Timing) for educational purposes or to test our own systems. We must never use them on real networks without permission.

VI. CONCLUSION AND FUTURE WORK

To sum up, we built simple versions of D-H and RSA, showed two classic attacks (MitM and timing), and added two standard defenses (signatures and blinding). Unauthenticated D-H is not safe against an active attacker, and a non-constant-time RSA decrypt leaks timing information. With our defenses, both issues are fixed in our tests. Signatures stop the MitM, and blinding reduces the timing signal. The main lesson is: protocols need authentication, and implementations need constant-time behavior (or blinding). Missing either one can break the security of a theoretically secure system.

For **future work**:

1) Add better timing analysis and plots (e.g., `matplotlib`).
2) Try a lower-level language to reduce Python runtime noise like C++.
3) Implement the Bleichenbacher attack, which is another famous attack on RSA padding (PKCS#1 v1.5).

4) Explore Elliptic Curve Diffie-Hellman (ECDH), which is the modern standard used in TLS 1.3.

Authors' Contributions

- **Jiachen Pan (50091098):** Implemented the core logic for both Diffie-Hellman and RSA protocols, including the attacks and defenses. Drafted the majority of the paper (Methods, Results, Discussion) and managed the overall LaTeX structure.
- **Junlin Li (50091104):** Drafted the Introduction and Related Work sections. Managed the bibliography and citations for accurate referencing. Assist the analysis of attack vectors and defensive strategies.
- **Yueshen Wang (50091084):** Responsible for refining the essay content and the final LaTeX formatting. Ensured the paper complied with IEEE conference standards, including figure placement and table formatting.
- **Junrui Liang (50091109):** Assisted with the Python implementation. Helped debug the RSA key generation and timing measurement scripts. Verified the experimental results for the RSA timing attack.

All authors reviewed and approved the final paper. Each author contributed to code testing and small fixes during integration.

References

[1] W. Diffie and M. E. Hellman, "New directions in cryptography," *IEEE Transactions on Information Theory*, vol. 22, no. 6, pp. 644-654, Nov. 1976.
[2] R. L. Rivest, A. Shamir, and L. Adleman, "A method for obtaining digital signatures and public-key cryptosystems," *Communications of the ACM*, vol. 21, no. 2, pp. 120-126, Feb. 1978.
[3] P. Kocher, "Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems," in *Advances in Cryptology — CRYPTO '96*, Berlin, Heidelberg, 1996, pp. 104–113.
[4] MITRE. *ATT&CK T1557: Adversary-in-the-Middle*. [Online]. Available: https://attack.mitre.org/techniques/T1557/
[5] MITRE. *D3FEND D3-CBAN: Certificate-based Authentication*. [Online]. Available: https://d3fend.mitre.org/technique/d3f:Certificate-basedAuthentication/
[6] University of Aberdeen, "Lecture 5: Asymmetric Encryption," JC3012 Network Security Technology, 2025.
[7] University of Aberdeen, "Lecture 6: Digital Signatures," JC3012 Network Security Technology, 2025.