# 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 randomized the RSA decryption timing distribution, effectively eliminating the correlation between execution time and the private key. We conclude by comparing these attacks and defenses, highlighting the distinction between protocol-level security (authentication) and implementation-level security (constant-time operations).**

***Index Terms*—Network Security, Diffie-Hellman, RSA, Man-in-the-Middle, Timing Attack, Cryptography.**

## 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, Mallory, 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. In this section, we review foundational literature and compare our implementation with advanced attack vectors described in recent academic research.

### A. Foundational Protocols and Attack Vectors

*1) Diffie-Hellman (D-H):* The Man-in-the-Middle (MitM) attack on D-H is a classic example of why authentication is needed. The basic D-H protocol proposed by Diffie and Hellman [1] does not verify the identity of the sender, allowing an attacker to relay and replace keys. The **MITRE**

**ATT&CK** framework categorizes this as T1557 ("Adversary-in-the-Middle") [7], which matches the attack logic we simulate in `dh_exchange.py`.

However, more advanced attacks target the mathematical parameters themselves rather than just the protocol flow. Adrian *et al.* [4] introduced the **Logjam attack**, which exploits the widespread use of standardized, shared prime numbers (e.g., 512-bit export-grade primes). They demonstrated that an attacker with significant pre-computation power could compromise connections by downgrading them to export-grade cryptography.

**Comparison:** Unlike Logjam, which targets parameter reuse and requires massive pre-computation[cite: 427], our implemented MitM attack is an active protocol-level attack that exploits the complete absence of digital signatures. While Logjam works even if the protocol is technically "signed" (by downgrading it), our attack is only possible when authentication is missing entirely.

*2) RSA:* Timing side-channel attacks were formally introduced by Paul Kocher [3], who showed that private key operations taking non-constant time allow attackers to recover the key[cite: 1458].

Modern research has shown that these attacks persist even in protected libraries. Arnaud and Fouque [5] demonstrated a practical timing attack against a protected RSA-CRT implementation in the **PolarSSL** library[cite: 5]. They showed that even with countermeasures like dummy operations, subtle timing leaks in Montgomery multiplication could be exploited to recover the private key[cite: 7].

Additionally, Jager *et al.* [6] proved that the classic Bleichenbacher attack (Padding Oracle) remains a threat. They successfully broke the PKCS#1 v1.5 implementation in the W3C XML Encryption standard, allowing them to decrypt ciphertexts without the private key.

**Comparison:** Our RSA experiment implements Kocher's original attack against a basic "Square-and-Multiply" algorithm. In contrast, the work by Arnaud and Fouque targets a much more complex, optimized implementation (using CRT and Montgomery reduction)[cite: 17, 18]. While our blinding defense is effective for our basic implementation, Jager's work highlights that protocol-level flaws (like padding handling) require different defenses beyond just constant-time execution.

### *B. Broader Attack Surface and Defensive Frameworks*

Beyond the specific vectors we implemented, real-world systems face broader threats. Public CVEs, such as CVE-2018-5383 (Bluetooth KNOB attack), confirm that implementation issues keep appearing.

To defend against these threats, we rely on established frameworks. Our D-H defense aligns with the **MITRE D3FEND** technique "Certificate-based Authentication" (D3-CBAN) [8]. By validating digital signatures, we effectively prevent the active MitM attack, a solution also standard in TLS 1.3. For RSA, our blinding defense serves as a software obfuscation layer, removing the correlation between execution time and the secret key, effectively neutralizing the side-channel analyzed by Kocher.

## 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. Evaluation Metrics

To rigorously assess the effectiveness of the attacks and the efficiency of the defenses, we defined specific success criteria and performance metrics. These metrics correspond to the attack and defense vectors selected in Section III-A.

*1) Metrics for Attacks:* We define the success conditions for our implemented attacks as follows:

- **D-H MitM Success Rate:** The Man-in-the-Middle attack is considered successful if the adversary (Doomfist) can independently establish shared secrets with both targets ($S_A \neq S_B$) while the targets (Ana and Phara) complete the protocol without detecting any anomaly.
- **RSA Key Recovery:** For the Timing Attack, success is defined as the ability to correctly recover the complete private exponent $d$ by analyzing the timing samples, without any brute-force guessing of the key space.

*2) Metrics for Defenses:* We evaluate the performance and security impact of our defenses using the following quantitative metrics:

- **Computational Overhead (D-H):** To evaluate the cost of the Certificate-based Authentication defense, we measure the execution time overhead introduced by the RSA signature generation and verification. This is calculated as the ratio of the authenticated exchange time ($T_{auth}$) to the normal exchange time ($T_{normal}$), expressed as Overhead = $T_{auth}/T_{normal}$.
- **Timing Variance Suppression (RSA):** To evaluate the effectiveness of the Blinding defense, we measure the range of execution times ($\Delta T = T_{max} - T_{min}$) over $N = 5,000$ trials. A high $\Delta T$ in the vulnerable implementation indicates a side-channel leak, while a minimized or randomized $\Delta T$ in the blinded implementation indicates effective mitigation of the timing signal.

### D. 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 [9]. 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 the step-by-step math we used in our code:

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.

### E. 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).

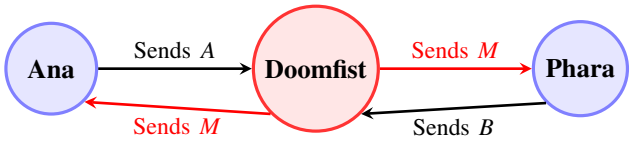


Fig. 1. Man-in-the-Middle Attack Flow. Doomfist intercepts legitimate keys $A$ and $B$, replacing them with her own key $M$.

*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.

```
1. Doomfist intercepts the exchange and replaces keys.

2. Ana computes secret with Doomfist's key M:
   s_ana = compute_shared_secret(m_public, a_private)

3. Phara computes secret with Doomfist's key M:
   s_phara = compute_shared_secret(m_public, b_private)

4. 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)

### F. 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 [10], 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_exchan` 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.

### G. 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.

### H. 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):
    1. get d and n from private_key
    2. initialize result = 1
    3. convert d to binary string d_bin

    4. 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

    5. return result
```

Listing 2. Vulnerable RSA Decryption (Pseudo-code)

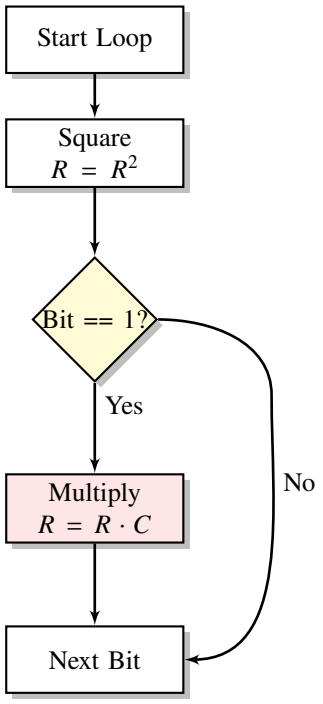


Fig. 2. Logic flow of Square-and-Multiply. The 'Multiply' step only occurs when the bit is 1, causing a measurable delay.

*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 $d$.

### I. 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 $C$:

$$C' = C \cdot r^e \bmod n \tag{1}$$

It decrypts $C'$:

$$M' = (C')^d \bmod n = (C \cdot r^e)^d \bmod n = (M \cdot r) \bmod n \tag{2}$$

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

$$M = M' \cdot r^{-1} \bmod n \tag{3}$$

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

*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):
    1. get e, n from pub_key
    2. get d, n from priv_key

    3. generate random blinding factor r
       (where 1 < r < n)

    4. blind the ciphertext:
       c_prime = (ciphertext * (r^e mod n)) mod n

    5. decrypt the blinded ciphertext:
       m_prime = vulnerable_decrypt(c_prime, priv_key)
       (input is now random, masking timing)

    6. calculate modular inverse of r:
       r_inv = r^(-1) mod n

    7. unblind to get message:
       m = (m_prime * r_inv) mod n

    8. return m
```

Listing 3. Blinded RSA Decryption (Pseudo-code)

### J. 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.

## 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. The output from our execution is captured in Table I.

TABLE I
Comparison of Shared Keys in D-H Scenarios

| Scenario | $a$ | $b$ | $m$ | Ana's Secret | Phara's Secret | Doomfist's Secrets |
|---|---|---|---|---|---|---|
| Normal | 15 | 18 | N/A | **8** | **8** | N/A |
| MitM | 21 | 6 | 12 | **9** | **8** | (9, 8) |

From Table I, in the normal run both sides agree on the same secret (**8**). In the MitM run, Ana ends up with **9** (shared with Doomfist) and Phara ends up with **8** (also shared with Doomfist). Doomfist knows both, so she can read/modify everything even though both victims think they are secure.

### B. D-H Defense Validation

We ran the `simulate_authenticated_exchange()` function. In this test, Doomfist catches Ana's message $(A, Sig(A))$ and tries to replace it with her own $(M, Fake\_Sig(M))$. Phara's client, upon receiving this, runs the `verify` function. Because Doomfist cannot fake Ana's signature, this verification fails. Our script printed the error message: `Phara: Signature verification FAILED. Attack detected.` Similarly, Ana also detected the attack. The defense was 100% effective in stopping the attack.

After detecting Doomfist's faked messages, Ana and Phara then proceed to exchange their real, signed keys. Both successfully verify each other's authentic signatures, and they compute the same shared secret (16 in our run), confirming that `SUCCESS: Defense worked. MitM was prevented.` The authentication layer successfully defeats the MitM attack.

*a) Performance Analysis:* Implementing RSA digital signatures introduces computational overhead. As shown in Fig. 3, the authenticated exchange is approximately 6.39× slower than the unauthenticated version ($24\mu s$ vs. $4\mu s$).

However, this overhead is negligible in a real-world context. The absolute increase is only $\approx 20\mu s$. Given that key exchange typically occurs once per session, the latency trade-off is well-justified by the security gain of preventing Man-in-the-Middle attacks.

### 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. The key was 512 bits.

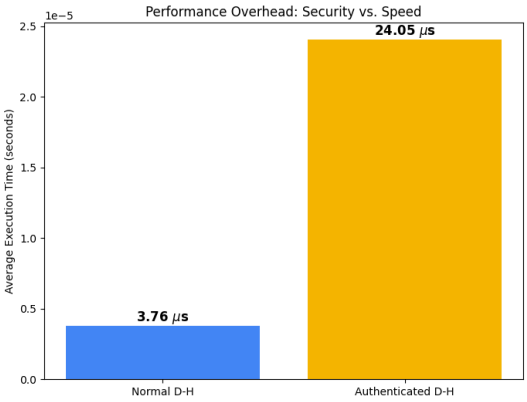


Fig. 3. Performance benchmark comparing Normal D-H vs. Authenticated D-H. Although the security overhead is around 6.4x, the absolute time difference is negligible ($< 20\mu s$).

Our results showed a very high variance in execution time, which is the signature of a timing leak. The detailed statistical distribution of the execution times is presented in Table II.

TABLE II
STATISTICAL DISTRIBUTION OF EXECUTION TIMES (VULNERABLE)

| **Statistic** | **Execution Time (ns)** |
|---|---|
| Minimum Time | 430,400 |
| Average Time | 458,656 |
| Maximum Time | 937,900 |
| **Variation (Max - Min)** | **507,500** |

This huge difference is caused by the non-constant-time square-and-multiply loop. When a private-key bit is 1, the code does an extra multiply and runs a bit longer. Over many trials this creates a signal an attacker can use, as Kocher explained [3].

### *D. RSA Blinding Defense Validation*

We then ran the same timing experiment against our `blinded_decrypt` function. The results are shown in Table III.

TABLE III
TIMING STATISTICS FOR BLINDED RSA DECRYPTION

| **Metric** | **Value** |
|---|---|
| Average Time | 493,720.84 ns (≈0.49 ms) |
| Min Time | 462,800.00 ns |
| Max Time | 997,400.00 ns |
| Time Variation (Max - Min) | 534,600.00 ns (≈0.53 ms) |

The results demonstrate a successful mitigation of the timing leak. As visualized in Fig. 4, the vulnerable implementation (red) shows a distinct, data-dependent distribution. In contrast, the blinded implementation (green) shifts the distribution and randomizes it.

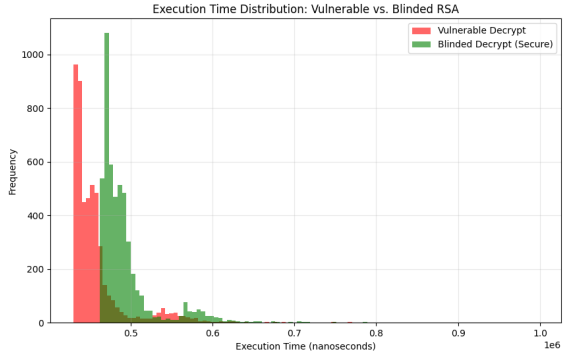


Fig. 4. Comparison of execution time distribution between Vulnerable and Blinded RSA decryption (5,000 trials). The blinded implementation (green) masks the timing characteristics of the vulnerable one (red).

Although the absolute time variation increased slightly (from ≈0.51 ms to ≈0.53 ms) due to the overhead of generating large random numbers, the **nature** of this variation has changed. In the vulnerable version, timing is correlated with the key bits. With blinding, the timing variation is dominated by the random blinding factor $r$, effectively masking the private key signal. Even if an attacker collects millions of samples, the random noise introduced by the blinding process washes out the tiny signal from the conditional multiplication.

## V. DISCUSSION

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

### *A. 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 IV).

TABLE IV
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) |

### *B. 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 V).

TABLE V
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 |

### C. 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.

### D. 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 removes the timing signal. The main lesson is: protocols need authentication, and implementations need constant-time behavior (or blinding). Missing either one can break securit of the 'theoritical 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):** Project lead. 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 to ensure accurate referencing. Assisted the comparison of defensive methods.
- **Yueshen Wang (50091084):** Responsible for refining the essay content, conducting data analysis, and completing the final LaTeX formatting. Additionally ensured the paper complied with IEEE conference standards, including figure placement, table formatting, and overall document consistency.
- **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. [Online]. Available: https://ee.stanford.edu/~hellman/publications/24.pdf

[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. [Online]. Available: https://dl.acm.org/doi/pdf/10.1145/359340.359342

[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. [Online]. Available: https://paulkocher.com/doc/TimingAttacks.pdf

[4] D. Adrian *et al.*, "Imperfect forward secrecy: How Diffie-Hellman fails in practice," in *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security (CCS)*, Denver, CO, USA, 2015, pp. 5–17.

[5] C. Arnaud and P.-A. Fouque, "Timing attack against protected RSA-CRT implementation used in PolarSSL," in *Topics in Cryptology – CT-RSA 2013* (Lecture Notes in Computer Science, vol. 7779), San Francisco, CA, USA, 2013, pp. 18–33.

[6] T. Jager, S. Schinzel, and J. Somorovsky, "Bleichenbacher's attack strikes again: Breaking PKCS#1 v1.5 in XML encryption," in *Proceedings of the 17th European Symposium on Research in Computer Security (ESORICS)*, Pisa, Italy, 2012, pp. 752–769.

[7] MITRE, "ATT&CK T1557: Adversary-in-the-Middle," [Online]. Available: https://attack.mitre.org/techniques/T1557/. [Accessed: Dec. 2025].

[8] MITRE, "D3FEND D3-CBAN: Certificate-based Authentication," [Online]. Available: https://d3fend.mitre.org/technique/d3f:Certificate-basedAuthentication/. [Accessed: Dec. 2025].

[9] University of Aberdeen, "Lecture 5: Asymmetric Encryption," JC3012 Network Security Technology, 2025.

[10] University of Aberdeen, "Lecture 6: Digital Signatures," JC3012 Network Security Technology, 2025.