JNEST EISSN: 2961-8738 ● PISSN: 2961-8916 ● DOI: 10.56741/jnest.v4i03.1275
Article Received 19 July 2025, Revised 02 November 2025

Integration of ECDHE Curve25519, RSASSA-PSS, and AES-256 for Enhanced PrivateDH Key Exchange Protocol in End-to-End Communication

Ardi Saputra*, Ronsen Purba

***Abstract*—The growing demand for secure digital communication calls for cryptographic protocols that are not only efficient but also capable of ensuring message confidentiality, integrity, and authenticity. PrivateDH is one such protocol that combines Diffie-Hellman, RSA, and AES; however, it still exhibits key weaknesses, including the absence of user authentication and reliance on classical Diffie-Hellman algorithms, which are computationally intensive and do not support forward secrecy. This study proposes an enhanced version of the PrivateDH protocol by integrating ECDHE Curve25519 as a replacement for classic DH, and RSASSA-PSS as a robust digital signature mechanism for user authentication. The methodology involves implementing and testing the proposed protocol within a peer-to-peer communication scenario, with performance evaluations based on handshake duration, CPU and memory usage, as well as security assessments including digital signature validation and forward secrecy. The results demonstrate that the enhanced protocol effectively accelerates key exchange, maintains resource efficiency, and provides reliable user authentication. In conclusion, this protocol contributes meaningfully to the advancement of more secure and efficient end-to-end communication systems, aligning with the demands of modern digital environments.**

***Index Terms*—Secure Communication, End-to-End Encryption, ECDHE Curve25519, RSASSA-PSS, AES-256**

I. INTRODUCTION

DIGITAL communication serves as a fundamental pillar in today's technological ecosystem, utilized across various domains—including financial transactions, professional collaboration, and social interaction [1,2]. Alongside the rapid adoption of information technology, there has been a growing demand for privacy protection and data security [3]. Users now expect communication systems that are not only fast and reliable, but also capable of ensuring data confidentiality and integrity [4]. However, despite its widespread adoption, digital communication also introduces notable security vulnerabilities [3], such as the risk of data interception or manipulation by unauthorized parties during transmission [5].

Dusane et al. [6] stated that protocols such as HTTPS are widely used to encrypt connections between users and servers. However, this approach only ensures security during transmission between the client and the server, without providing true end-to-end encryption between the sender and the receiver. In this model, the server still has access to the message contents, as decryption occurs before the data is forwarded to the recipient. This creates a dependency on the trustworthiness of the service provider. In cases of data breaches—whether due to hacking, external pressure from government authorities, or intervention by third-party organizations—user privacy may be compromised. As a result, new approaches such as End-to-End Encryption (E2EE) are becoming increasingly essential, as they ensure that only the sender and intended recipient can access the message content, excluding even the service provider itself [7,8].

The Diffie–Hellman (DH) key exchange protocol has long been used as a method for establishing a shared secret key between two parties over an insecure communication channel [9,10]. However, DH is vulnerable to various attacks, including Man-in-the-Middle (MITM) attacks, due to the absence of a mutual authentication mechanism [10,11]. One enhancement of DH is the PrivateDH protocol introduced by Patgiri [12], which integrates RSA and AES to strengthen DH. Nevertheless, PrivateDH still has some weaknesses. First, the receiver cannot distinguish between a legitimate user and an attacker when receiving two identical ciphertexts from different sources, which may lead to failure in its implementation. Although PrivateDH provides an anomaly detection mechanism by identifying the reception of two identical ciphertexts from different sources, this mechanism is reactive and does not prevent compromises that may have already occurred. Second, PrivateDH incurs computational overhead due to the relatively heavy encryption and decryption processes, particularly related to DH parameters. In its implementation, the sender must perform one encryption (public key) and two decryptions (parameters p and g), while the receiver performs two encryptions (parameters p and g) and one decryption (public key). Moreover, the use of DH in PrivateDH still requires large key sizes to maintain a modern level of security, which ultimately increases the computational burden and reduces the efficiency of the protocol [13].

To address the weaknesses of PrivateDH, this study proposes the integration of several modern cryptographic approaches, specifically selected based on their relevance to the identified issues. First, ECDHE Curve25519 is employed to replace DH because it uses fixed parameters, thereby reducing computational overhead, improving key exchange efficiency, and offering smaller key sizes, faster computation, and stronger security compared to DH or CurveP256 [14–18]. Second, RSASSA-PSS is chosen as the digital authentication mechanism because it is probabilistic and more resistant to padding oracle attacks than PKCS#1 v1.5, aligning with modern cryptographic standards [19,20]. Third, AES is retained as in PrivateDH

Ardi Saputra and **Ronsen Purba** are with the Information Technology Program, Universitas Mikroskil, Medan 20238, Indonesia.
*Correspondence: ardisaputraa12@gmail.com

© 2025 IISTR

but configured as AES-256 to provide stronger resistance against brute-force attacks and a higher security margin compared to AES-128 [21,22].

PrivateDH still exhibits weaknesses in the context of modern communication, such as the lack of adequate authentication, vulnerability to manipulative attacks due to identical ciphertexts, and limited efficiency stemming from the use of classical DH, which also does not optimally support forward secrecy. To improve both the security and efficiency of PrivateDH, this study proposes the integration of two main approaches: (1) ECDHE Curve25519 to replace DH in generating more secure and efficient session keys, and (2) RSASSA-PSS to strengthen user authentication and prevent misidentification when identical ciphertexts are received from different sources. Meanwhile, RSA and AES are retained as part of the original protocol structure. This proposed approach is expected to enhance the capability of PrivateDH in supporting secure and efficient end-to-end communication.

II. Related Work

The Diffie–Hellman (DH) key exchange [9,10] remains a foundational mechanism for establishing shared keys in insecure networks. However, DH by itself does not provide authentication, making it vulnerable to Man-in-the-Middle (MITM) attacks [11]. To address this, modern protocols have integrated DH variants with authentication and forward secrecy as baseline requirements. TLS 1.3 [23], for instance, employs Elliptic-Curve Diffie–Hellman Ephemeral (ECDHE) to ensure forward secrecy and combines it with digital signatures (RSA, ECDSA, EdDSA) for authentication. The Noise Framework [24] generalizes this concept by providing modular handshake patterns that combine DH, symmetric encryption, and hashing. Similarly, the Signal protocol [25] extends Noise by introducing the double ratchet mechanism to achieve post-compromise security in asynchronous messaging contexts. These standardized approaches illustrate the evolution of authenticated key exchange (AKE) protocols toward stronger security guarantees, flexibility, and resilience against key compromise. Additionally, the SIGMA protocol [33] is widely recognized as a foundational approach for authenticated DH exchanges, forming the basis of IKE and influencing the design of TLS and Noise patterns. Our work differs by embedding a SIGMA-like signed transcript within the PrivateDH architecture while maintaining a lightweight client-to-client model.

One notable enhancement to DH is the PrivateDH protocol proposed by Patgiri [12]. PrivateDH integrates RSA and AES into DH to mitigate its weaknesses. Specifically, RSA is applied for encrypting initial symmetric key and AES for encrypting exchanged parameters, while an anomaly detection mechanism is included to identify suspicious cases such as identical ciphertexts arriving from different sources. These additions improve confidentiality and offer a degree of verification compared to vanilla DH. Nevertheless, PrivateDH exhibits several limitations. Its anomaly detection operates reactively rather than preventively, meaning attacks may succeed before being flagged. This approach generates computational overhead by encrypting and decrypting DH parameters, which may result in increased system burden. Moreover, PrivateDH continues to rely on classical DH, requiring large key sizes for adequate security—leading to reduced efficiency in modern contexts.

Recent research on authenticated key exchange has shifted toward elliptic-curve–based solutions due to their balance of efficiency and strong security guarantees. Introduced by Bernstein [26], Curve25519 is widely adopted for its efficiency and strong security properties, it combines small key sizes with fast computations and resistance to side-channel attacks [27]. On the authentication side, RSASSA-PSS is recommended in RFC 8017 [28], its probabilistic padding scheme offers stronger protection against adaptive chosen-message attacks compared to deterministic alternatives like PKCS#1 v1.5. Standardized by NIST [29], AES is a block cipher that ensures confidentiality through multiple rounds of transformation. Paar et al. [22] evaluate trade-offs between AES-128 and AES-256, showing that while AES-256 requires slightly more computation, it significantly raises brute-force resistance. These primitives collectively define the modern cryptographic landscape in which efficiency, scalability, and security must be balanced.

From the above studies, it is evident that while PrivateDH pioneered the integration of asymmetric and symmetric primitives into DH, its weaknesses—reactive anomaly detection, reliance on heavy RSA operations, and dependence on classical DH—limit its suitability for modern lightweight environments. In contrast, standard authenticated key exchange (AKE) protocols such as TLS 1.3, Noise, and Signal already employ ECDHE and modern digital signatures, but they are designed as comprehensive ecosystems. TLS 1.3, for instance, targets internet-scale client–server communication with certificate-based PKI and session resumption, while Noise and Signal rely on long-term identity keys and support additional features such as group communication, asynchronous messaging, and post-compromise security.

This research positions itself differently: it does not aim to provide a universal framework, but an incremental improvement over PrivateDH with a narrower scope. The proposed protocol introduces Curve25519-based ECDHE for forward secrecy and efficiency, integrates RSASSA-PSS for preventive authentication, and leverages AES for confidentiality. While the Noise Framework offers flexible handshake patterns and the Signal Protocol extends this into asynchronous, multi-session settings with ratcheting, the present work focuses on a narrower use case: a single-session handshake that redesigns the PrivateDH message flow into a Noise-like pattern adapted to constrained contexts. Unlike TLS or Noise, the trust model here does not depend on a CA or permanent identity keys; instead, a semi-trusted server merely relays RSA public keys to facilitate the handshake. By redesigning the PrivateDH flow in this manner, the protocol provides a lightweight client-to-client authenticated key exchange for constrained contexts, while still ensuring forward secrecy and end-to-end confidentiality.

III. METHODS

Figure 1 illustrates the research methodology, focusing on four main processes: generating the initial shared secret key, exchanging ECDHE public keys, generating the final shared secret key, and using the shared secret key. In this study, it is assumed that user A intends to send a message to user B, and both require each other's RSA public keys for encryption and authentication. Therefore, public keys are obtained from a trusted third party (TTP).

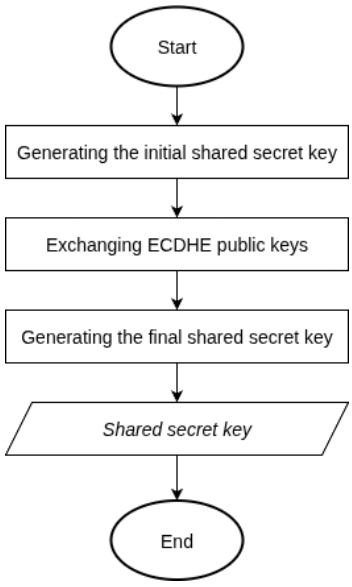


Fig. 1. Research methodology.

The concept of a TTP for RSA public key distribution is adopted from previous work [12], where a trusted entity facilitates the exchange of public keys between users. While the earlier study did not detail the implementation of the TTP, in this study the TTP is implemented as a centralized server that stores and distributes users' public keys upon request. The server provides public keys over a secure communication channel, such as HTTPS, assuming that HTTPS is implemented correctly with valid certificates and protection against MITM attacks. However, if the server or communication channel is compromised, the authenticity of the public keys cannot be guaranteed, which introduces risks of impersonation and MITM attacks.

In this protocol, RSA public keys are used only once by user A to encrypt the AES key, which will be used for the rest of the communication. This minimizes the exposure of RSA keys in the communication process, with AES providing the primary encryption mechanism for the bulk of the message exchange. The RSA public key is also used for authentication through digital signatures, where user A signs the initial message with their private key to guarantee the integrity and authenticity of the communication. This ensures that, even though the TTP is semi-trusted, the message's authenticity is preserved by the private key signing process, preventing impersonation.

Unlike traditional Public Key Infrastructures (PKI) that rely on certificates, certificate revocation lists (CRL), or Online Certificate Status Protocol (OCSP) for key validation, this implementation simplifies the process by relying solely on the TTP to distribute keys without additional validation or key revocation mechanisms. This approach minimizes overhead, making it suitable for constrained environments but comes with the assumption that the TTP is trustworthy and secure. Although these risks are acknowledged, they fall outside the scope of the current threat model, which focuses on the protocol's efficiency and lightweight nature in constrained environments. To mitigate risks of compromised key distribution, mechanisms such as certificate revocation via OCSP/CRL or certificate pinning may be applied, ensuring that endpoint authentication remains robust even under partial PKI compromise.

Given that the RSA public key is used only for the initial key exchange and authentication, and that the communication subsequently relies on AES for encryption, the risks associated with the TTP become more limited in later stages of the communication. Once the AES key is securely exchanged, the protocol shifts to AES for further communication, reducing the security risks linked to the TTP. All messages exchanged during the handshake phase are signed using RSASSA-PSS. This ensures that each party can verify the authenticity of the transcript and bind all elements of the session to the corresponding identity.

A. Generating the Initial Shared Secret Key

Figure 2 (Stage 1) illustrates the process of generating the initial shared secret key between User A and User B. User A performs the following operations:

1) Generate a random secret key.
2) Sign the secret key using their own RSA private key, producing a signature of the secret key.
3) Encrypt the secret key using user B's RSA public key, producing the ciphertext of the secret key.
4) Send the ciphertext and the signature to user B.

User B performs the following operations:

1) Decrypt the ciphertext using their own RSA private key; if decryption fails, the program terminates.
2) Verify that the signature on the secret key was created by user A using user A's RSA public key.

Once both users possess the initial shared secret key, they can use it with AES to encrypt and decrypt messages.

B. Exchanging ECDHE Public Keys

Figure 2 (Stage 2) illustrates the process of exchanging ECDHE public keys between User B and User A. User B performs the following operations:

1) Generate their own random ECDHE private key.
2) Compute the ECDHE public key based on predefined ECDHE parameters.
3) Sign the ECDHE public key using their own RSA private key, producing a signature of the ECDHE public key.
4) Encrypt the ECDHE public key and its signature using AES with the initial shared secret key, producing the ciphertext.
5) Send the ciphertext to user A.

User A performs the following operations:

1) Decrypt the ciphertext using AES with the initial shared secret key; if decryption fails, the program terminates.

2) Verify that the signature on the ECDHE public key was created by user B using user B's RSA public key; if verification fails, the program terminates.
3) Perform the same activities as user B (steps 1-4) to create the ciphertext of their own ECDHE public key and signature.
4) Send the ciphertext to user B.

The data received by user B is decrypted and verified; if the process fails, the program terminates. At this point, both users have successfully exchanged ECDHE public keys.

C. Generating the Final Shared Secret Key

Figure 2 (Stage 3) illustrates the process by which User A and User B generate the final shared secret key. Both users perform the following operations:

1) Ensure that the received ECDHE public key is valid and lies within the curve domain; if validation fails, the program terminates.
2) Compute the final shared secret key using the received public key with ECDHE.
3) Delete the ECDHE private key.

D. Shared Secret Key

At this stage, both users possess the same ECDHE shared secret, which is subsequently used to derive a 256-bit AES session key for encrypting and decrypting all communication. This shared secret is processed using HKDF with SHA-256, incorporating a session-specific salt and a labeled context (info) string. The salt is derived deterministically from ephemeral session parameters—specifically, the previously established initial shared secret key—while the info string is a predefined, static context label known to both parties. Consequently, both values are identical on both sides without requiring transmission. This design ensures that both parties derive the same session-unique AES key, enhancing security while minimizing communication overhead. The experimental setup simulates a communication session between two clients (User A and User B), where each user exchanges encrypted messages using this derived key.

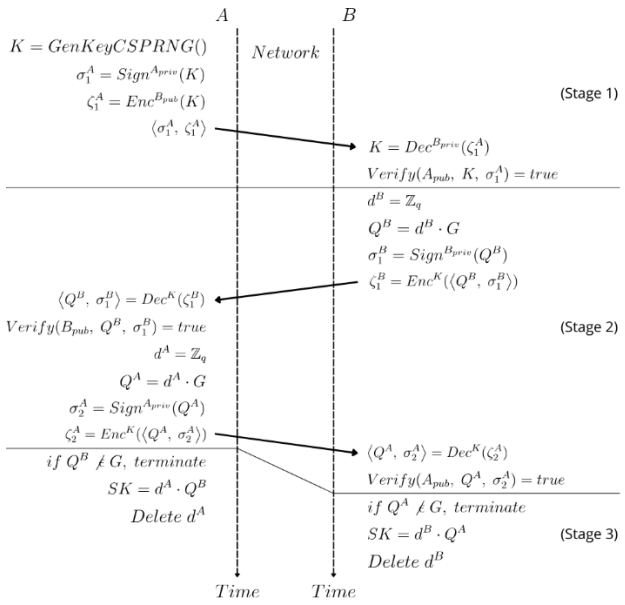


Fig. 2. The message flow diagram of the proposed key exchange protocol, comprising three stages: (1) generating the initial shared secret key, (2) exchanging ECDHE public keys, and (3) generating the final shared secret key.

TABLE I
EXPLANATION OF NOTATIONS IN THE PROPOSED PROTOCOL DIAGRAM

Notation	Description
A	The Sender.
B	The receiver.
K	Initial secret key for AES.
SK	Final shared secret key.
$GenKeyCSPRNG()$	Function that generates a cryptographic key using a CSPRNG.
σ^A, σ^B	Digital signatures generated by A and B, respectively.
ζ^A, ζ^B	Ciphertexts transmitted by A and B, respectively.
q	Prime order of the finite field for Curve25519 ($q = 2^{255} - 19$).

Notation	Description
$\mathbb{Z}_q$	Field of integers modulo q ($0 \leq x < q$); domain for the private key.
G	Generator point on Curve25519 (with x-coordinate = 9).
d^A, d^B	ECDHE private keys of A and B, respectively.
Q^A, Q^B	ECDHE public keys of A and B, respectively.
$Sign^{A_{priv}}, Sign^{B_{priv}}$	RSA signing process using the private keys of A and B, respectively.
$Verify(\cdot)$	Signature verification function.
$Enc^{B_{pub}}$	Asymmetric encryption process using RSA by the public key of the B.
$Dec^{B_{priv}}$	Asymmetric decryption process using RSA by the private key of the B.
Enc^K	Symmetric encryption by AES using secret key K.
Dec^K	Symmetric decryption by AES using secret key K.

The proposed key exchange protocol was implemented in Python 3.12 using libraries such as socketio, flask, requests, psutil, and time[1]. It was run on Ubuntu 24.04.2 LTS, with tests conducted on a machine equipped with an Intel(R) Core(TM) i3-1005G1 CPU @ 1.20GHz, 8 GB DDR4 RAM, and 256 GB NVMe SSD. Testing took place in a controlled local network environment to isolate computational performance factors. The configuration of the test devices is as follows:

- AES was configured in GCM mode with a 256-bit key length, a 96-bit (12-byte) initialization vector, and a 128-bit authentication tag. Nonce reuse was prevented by generating a fresh random IV for each encryption.
- RSA was configured with a 2048-bit key length, using OAEP padding for encryption/decryption with MGF1-SHA-256, PSS padding for signing/verification with MGF1-SHA-256 and a salt length of 32 bytes. SHA-256 was used consistently across all RSA operations.
- Classic DH was configured with a 2048-bit key length.
- ECDHE was configured using the X25519 curve with a fixed key length of 256 bits.

RSA-2048 with RSASSA-PSS is used for endpoint authentication to maintain continuity with the PrivateDH baseline and to rely on mature, widely audited implementations and HSM/PKCS#11 support. In our single-handshake setting, RSA's higher signing cost is amortized by network latency, while forward secrecy and bulk performance are ensured by X25519 and AES-GCM. This algorithm produces larger artifact sizes (~256 bytes for public key and signature) compared to Ed25519/Ed448 (32/64 bytes), and has a slower signing process but competitive verification [32]. Operationally, our semi-trusted key-relay already distributes RSA public keys (by design of PrivateDH), so RSASSA-PSS avoids introducing a new identity-key lifecycle; migrating to EdDSA would change the trust/registration model and is left as future work.

In the protocol implementation, the management of the nonce or initialization vector (IV) for AES-GCM is performed by generating a 12-byte IV using the `os.urandom(12)` function for each encryption session. This approach ensures that each encryption operation employs a unique IV, preventing reuse that could compromise cryptographic security. Furthermore, symmetric keys, nonces, and other random values are generated using a cryptographically secure random number generator (CSPRNG) to ensure sufficiently high entropy levels, as mandated by NIST SP 800-90A [30], thereby reducing predictability and maintaining cryptographic strength.

To evaluate the performance of the proposed protocol, experiments were conducted on both the proposed scheme and the baseline protocol, PrivateDH. This enabled explicit comparison of key performance metrics, including handshake duration, CPU usage, and memory consumption. Measurements covered a full handshake session, from connection initiation to the establishment of the shared secret key, excluding any additional data transmission. Timing was recorded using `time.perf_counter()` in Python, while system resource usage was monitored with the `psutil` library. Each experiment began with 5 warmup runs to reduce cold start and caching effects. Performance metrics were then collected over 10 repeated trials per protocol, and the average and standard deviation were calculated for each data point.

Additionally, a DH 256-bit variant was included in the evaluation to illustrate the lower bound of performance, although it is acknowledged that such a configuration does not meet modern cryptographic security standards. The 256-bit classical DH baseline was selected to approximate a 128-bit security level comparable to X25519, ensuring that performance differences reflect algorithmic efficiency rather than security strength. It was not used as a secure baseline, but rather to emphasize the efficiency trade-offs when reducing cryptographic strength. The primary performance and security comparison remains between ECDHE Curve25519 and DH 2048-bit.

The security analysis of the proposed protocol is based on a well-defined threat model that describes potential adversary capabilities and the intended security goals. All

[1]The source code and experimental scripts are available at: https://github.com/ardisyahputra12/Enhanced-PrivateDH-Key-Exchange-Protocol

security testing scenarios were implemented and executed through customized program code, simulating various attack vectors in a controlled environment. These simulations were designed to mimic realistic adversarial behaviors according to the defined threat model. We assume the adversary may have one or more of the following capabilities:

- Network attacker (eavesdropping): The attacker can monitor all network traffic between the two parties (A and B).
- Active Man-in-the-Middle (MITM): The attacker can modify, inject, or drop messages sent between A and B in real time.
- Key leakage scenario: The attacker may obtain one party's RSA public key, or even a specific session key, but cannot obtain that party's RSA private key. In partial compromise scenarios, the ephemeral ECDHE private key, which is specific to a session, may be exposed after the session ends. Additionally, the initial shared key might be revealed, which tests the protocol's forward secrecy and resistance to impersonation attacks.
- Replay attack: The attacker may attempt to reuse valid protocol messages from previous sessions to impersonate a party.
- Chosen-Message and signature forgery: The attacker may attempt to modify signed messages or create forged signatures without the private signing key.

To mitigate the above threats, the protocol ensures the following security properties:

- Confidentiality: Ensure that session keys and encrypted payloads remain secret from both passive and active attackers.
- Integrity: Detect any unauthorized modification of exchanged data.
- Authentication: Verify that messages originate from the legitimate sender using RSASSA-PSS digital signatures.
- Forward Secrecy: Prevent compromise of past session keys even if long-term private keys are exposed.
- Replay Protection: Ensure that previously valid protocol messages cannot be reused to gain unauthorized access.
- Unlinkability: Prevent session correlation by ensuring each handshake generates unique ephemeral key material.

Each security claim presented in Section 4 explicitly references the adversary capabilities and attack scenarios defined here, ensuring that the evaluation is directly linked to the stated threat model. The cryptographic primitives used in the protocol—X25519 for key exchange, AES-GCM for authenticated encryption, and RSASSA-PSS for digital signatures—are assumed to be secure under current cryptographic standards and resistant to brute-force and modern cryptanalytic attacks. The RSA public key is distributed through a semi-trusted relay, which is assumed to preserve its authenticity.

IV. Results

The protocol testing was conducted in a peer-to-peer communication simulation between users A and B. The results were divided into three main aspects: functionality, security, and performance.

TABLE II
Security Testing Scenarios

Scenario	Test Steps	Test Data	Expected Result
Compromise of the Main ECDHE Private Key	1. User A and User B perform a full key exchange using the protocol. 2. Simulate a key compromise by exposing the long-term ECDHE private key of User A. 3. Attempt to regenerate the original shared key using only the public key of User B from the new session and the compromised private key of User A, assuming the context of a previous session. 4. Compare the newly computed shared key with the original shared key.	• User A's private key • User B's public key • Original session shared key • Newly computed shared key	It is not possible to reconstruct the original shared key without the ephemeral private key used during the actual session. Forward secrecy is verified.
MITM Attack with Random Secret Key Modification	1. Assume that the RSA public key of User B has been leaked and is known to the attacker. 2. User A sends a ciphertext containing a random secret key to User B, along with a digital signature. 3. The attacker intercepts the data and modifies the original random secret key with a new one. 4. The attacker forwards the modified ciphertext and the original signature (unchanged) to User B.	• User B's RSA public key • Original random secret key • Modified random secret key • Original ciphertext • Modified ciphertext • User A's original signature • Attacker's (unchanged) signature	Signature verification fails due to data modification, resulting in an invalid signature. The data is rejected.

Scenario	Test Steps	Test Data	Expected Result
	5. User B receives the data and proceeds with the protocol steps, including signature verification.		
MITM Attack Without Authentication	1. Assume that the RSA public key of User B has been leaked and is known to the attacker. 2. Temporarily disable the digital signature scheme (RSASSA-PSS). 3. Re-execute the MITM attack scenario by modifying the random secret key within the ciphertext sent from User A to User B. 4. Evaluate whether the communication continues without detection.	• User B's RSA public key • Original random secret key • Modified random secret key • Original ciphertext • Modified ciphertext	The attacker successfully inserts themselves into the communication flow, and the session proceeds without detection.
MITM Attack with Signature Modification	1. Assume that the initial shared key between User A and User B has been compromised and is known to the attacker. 2. User B sends a ciphertext containing a ECDHE public key along with a valid digital signature to User A. 3. The attacker intercepts the data and replaces the original signature with a new signature generated using the attacker's own private key. 4. The attacker forwards the modified ciphertext and forged signature to User A. 5. User A receives the data and proceeds with signature verification.	• Initial shared key • Original ciphertext from User B • Modified ciphertext from attacker • Original signature from User B • Forged signature from attacker	Signature verification fails because the signature does not originate from the legitimate sender. The message is rejected.
MITM Attack with Modification of the ECDHE Public Key	1. Assume that the initial shared key has been compromised and is known to the attacker. 2. User B sends a ciphertext containing their ECDHE public key along with a digital signature to User A. 3. The attacker intercepts the message and replaces User B's ECDHE public key with their own ECDHE public key. 4. The attacker forwards the modified ciphertext, still containing User B's original signature, to User A. 5. User A receives the data and performs signature verification.	• Initial shared key • User B's original ECDHE public key • Attacker's ECDHE public key • Original ciphertext • Modified ciphertext • User B's original signature	Signature verification fails due to data modification causing signature mismatch. The data is rejected.
Replay Attack	1. User A completes a full handshake session with User B, resulting in the establishment of a valid shared session key. 2. The attacker captures all messages exchanged during this session, including the encrypted payloads that contain the exchanged ECDHE public keys. 3. The attacker replays User A's previously sent ECDHE public key ciphertext to User B in a later session, unchanged. 4. User B receives the ciphertext and attempts to process it, assuming it is part of a valid handshake.	User A's ECDHE public key ciphertext from the previous session	Replayed messages are rejected because the session key changes every time due to ephemeral ECDHE, making old encrypted data unusable.

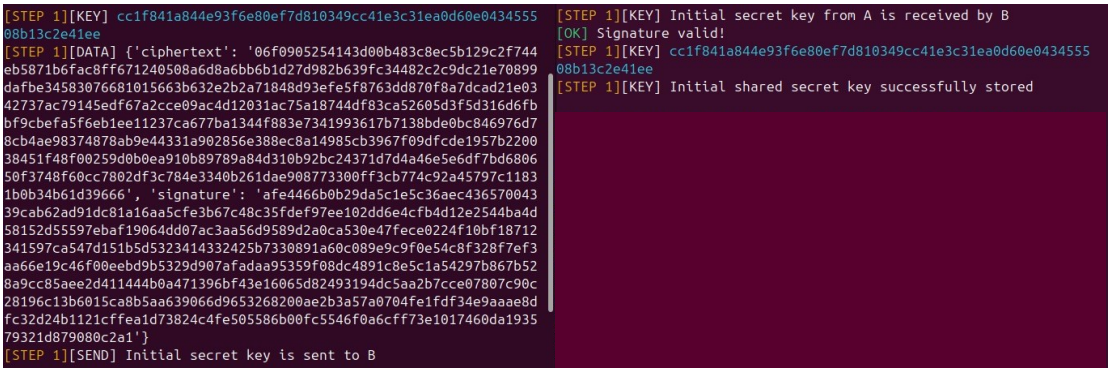


Fig. 3. Initial shared secret key generation stages (User A on the left and User B on the right).



Fig. 4. ECDHE public key exchange stages (User A on the left and User B on the right).

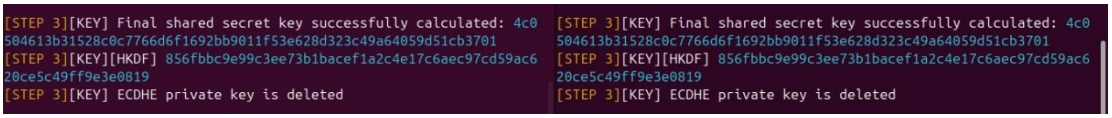


Fig. 5. Final shared secret key generation stage (User A on the left and User B on the right).



Fig. 6. Stages of using shared secret key in communication (User A on the left and User B on the right).

TABLE III
TEST DATA IN THE SCENARIO OF MAIN ECDHE PRIVATE KEY COMPROMISE.

Test Data	Session 1	Session 2	Testing
User A's private key	b025d5cff7d8373cb5b7e3acb22a36b193220b23445f1b4964ea3948a15ffc6d	80eb6464929b3ffb0f55d7d7c5ab417da25f31505919869fef03f69cbe54d257	b025d5cff7d8373cb5b7e3acb22a36b193220b23445f1b4964ea3948a15ffc6d
User B's public key	adf0501e63acdb63005f48e3cdeafc58b1ab80874d7ae57ec3f08912ba85a379	3577be028e3a4c2a0136b6cee0f5f53d95dcf78bce6c0045db788051b8a96d7f	3577be028e3a4c2a0136b6cee0f5f53d95dcf78bce6c0045db788051b8a96d7f
Shared key result	85431edbd4960a3c8509531f31c6728037dd086c120cce3bfaed628ab6c32e6a	1d40ef52167347cf03961e84706737c6bf584d652256004733a1329f6d806f1e	96f85ee4a46415b9e7291f19e16ce3b46422557380be76be2ff04996d3484a06

TABLE IV
HANDSHAKE TIME TEST RESULTS (IN MILLISECONDS) FOR USER A ONLY, COMPARING PROPOSED PROTOCOL WITH BASELINE PRIVATEDH (2048-BIT AND 256-BIT)

Test #	Proposed Protocol	Baseline PrivateDH (2048 bit)	Baseline PrivateDH (256 bit)
1	267.111	19257.339	243.526
2	238.493	11476.929	274.415
3	235.470	5633.612	231.257
4	235.723	11301.952	261.620
5	256.319	29406.852	249.284
6	247.784	21335.500	259.621
7	251.766	27024.088	245.685
8	249.045	7578.000	287.693
9	250.081	10186.798	245.666
10	259.641	6276.494	272.073
Min	**235.47**	**5633.61**	**231.26**
Max	**267.11**	**29406.85**	**287.69**
Mean	**249.14**	**14947.76**	**257.08**
Std Dev	**10.41**	**8682.94**	**17.19**
Median	**249.56**	**11389.44**	**254.45**
IQR	**14.37**	**12585.76**	**23.79**
n	**10**	**10**	**10**
95% CI	**[241.70, 256.59]**	**[8736.35, 21159.16]**	**[244.78, 269.38]**

TABLE V
CPU USAGE TEST RESULTS (IN %) FOR USER A AND USER B, COMPARING PROPOSED PROTOCOL WITH BASELINE PRIVATEDH (2048-BIT AND 256-BIT)

Test #	Proposed Protocol		Baseline PrivateDH (2048 bit)		Baseline PrivateDH (256 bit)	
	User A	User B	User A	User B	User A	User B
1	7.5	0.5	0.2	98.9	8.2	12.3
2	4.2	0.3	0.3	98.1	7.3	21.9
3	4.2	0.5	0.5	96.0	4.3	8.5
4	8.5	0.3	0.3	97.9	3.8	18.8
5	7.8	0.3	0.1	99.1	8.0	15.9
6	8.1	0.6	0.1	99.0	3.8	19.0
7	7.9	0.4	0.1	99.2	4.1	12.1
8	8.0	0.6	0.4	96.8	6.9	17.2
9	8.0	0.3	0.4	97.9	4.1	8.1
10	7.7	0.7	0.5	96.5	3.7	21.8
Min	**4.2**	**0.3**	**0.1**	**96.0**	**3.7**	**8.1**
Max	**8.5**	**0.7**	**0.5**	**99.2**	**8.2**	**21.9**
Mean	**7.19**	**0.45**	**0.29**	**97.94**	**5.42**	**15.56**
Std Dev	**1.60**	**0.15**	**0.16**	**1.16**	**1.92**	**5.08**
Median	**7.85**	**0.45**	**0.30**	**98.0**	**4.20**	**16.55**
IQR	**0.45**	**0.28**	**0.28**	**1.90**	**3.33**	**6.80**
n	**10**	**10**	**10**	**10**	**10**	**10**
95% CI	**[6.05, 8.33]**	**[0.34, 0.56]**	**[0.18, 0.40]**	**[97.11, 98.77]**	**[4.05, 6.79]**	**[11.93, 19.19]**

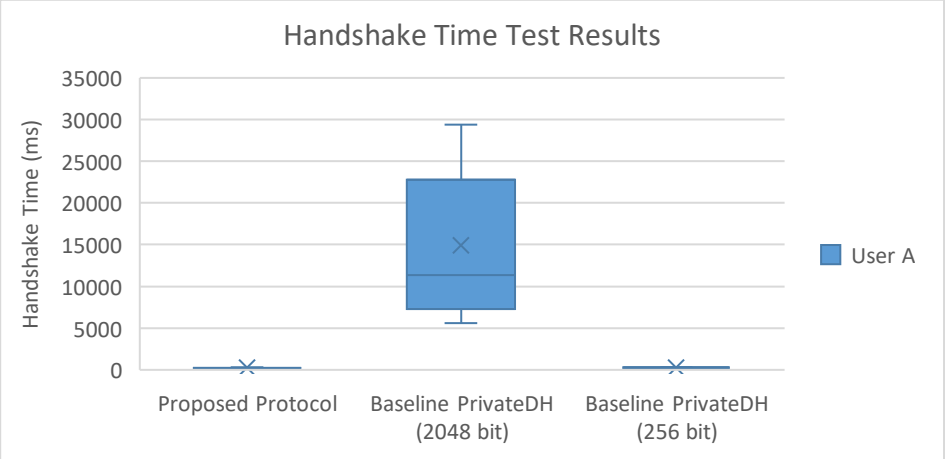


Fig. 7. Boxplot showing handshake time results for User A over 10 test runs, comparing the proposed protocol with baseline PrivateDH using 2048-bit and 256-bit key sizes.

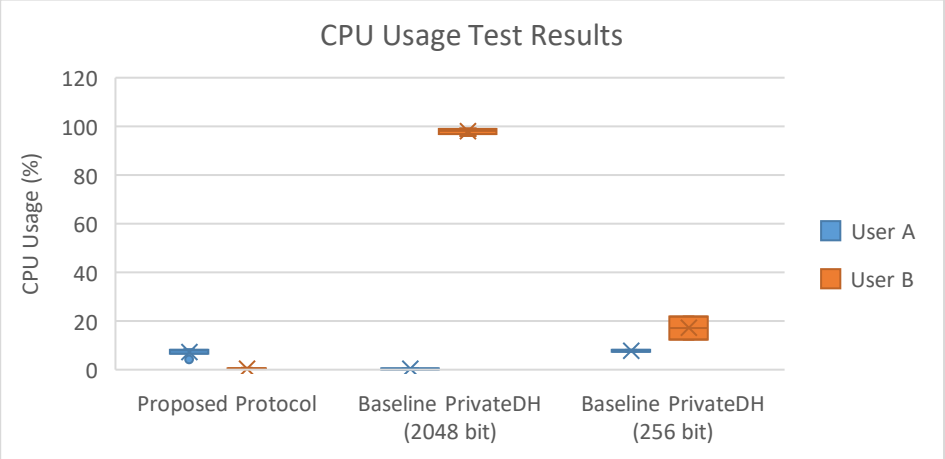


Fig. 8. Boxplot of CPU usage (%) for User A and User B across 10 test runs, comparing the proposed protocol with baseline PrivateDH using 2048-bit and 256-bit key sizes.

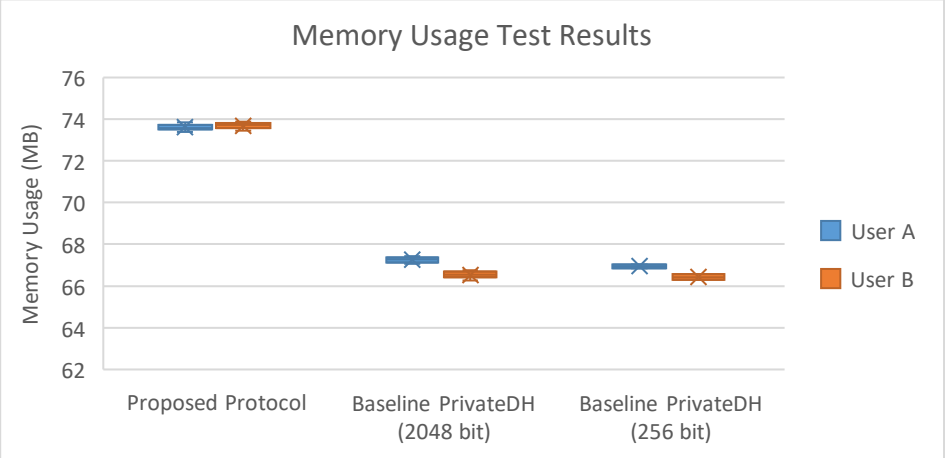


Fig. 9. Boxplot of memory usage (MB) for User A and User B across 10 test runs, comparing the proposed protocol with baseline PrivateDH using 2048-bit and 256-bit key sizes.

TABLE VI
MEMORY USAGE TEST RESULTS (IN MEGABYTES) FOR USER A AND USER B, COMPARING PROPOSED PROTOCOL WITH BASELINE PRIVATEDH (2048-BIT AND 256-BIT)

Test #	Proposed Protocol		Baseline PrivateDH (2048 bit)		Baseline PrivateDH (256 bit)	
	User A	User B	User A	User B	User A	User B
1	73.50	73.66	67.36	66.43	66.86	66.56
2	73.68	73.83	67.36	66.74	67.03	66.29
3	73.84	73.65	67.12	66.76	66.99	66.44
4	73.82	73.76	67.29	66.25	66.86	66.56
5	73.57	73.44	67.44	66.54	66.96	66.29
6	73.58	73.86	67.09	66.52	66.99	66.30
7	73.47	73.54	67.27	66.45	67.12	66.37
8	73.38	73.80	67.06	66.38	67.23	66.34
9	73.66	73.80	67.21	66.66	67.12	66.45
10	73.55	73.56	67.41	66.58	66.83	66.34
Min	**73.38**	**73.44**	**67.06**	**66.25**	**66.83**	**66.29**
Max	**73.84**	**73.86**	**67.44**	**66.76**	**67.23**	**66.56**
Mean	**73.60**	**73.69**	**67.26**	**66.53**	**66.99**	**66.39**
Std Dev	**0.15**	**0.14**	**0.14**	**0.16**	**0.13**	**0.10**
Median	**73.57**	**73.71**	**67.28**	**66.53**	**66.99**	**66.36**
IQR	**0.16**	**0.22**	**0.22**	**0.20**	**0.21**	**0.14**
n	**10**	**10**	**10**	**10**	**10**	**10**
95% CI	**[73.50, 73.71]**	**[73.59, 73.79]**	**[67.16, 67.36]**	**[66.42, 66.65]**	**[66.91, 67.09]**	**[66.32, 66.47]**

A. Functionality

The functionality of the protocol refers to the successful execution of each stage in the key exchange process, from initiation to the utilization of the shared key for encrypted communication. The evaluation is conducted to ensure that all stages operate as designed and are capable of producing valid outputs that can be effectively used in end-to-end communication scenarios.

1) **Generating the Initial Shared Secret Key**
User A generates an initial secret key randomly and signs it using their RSA private key. The secret key is then encrypted with User B's RSA public key and transmitted. Upon receipt, User B successfully verifies the signature using User A's RSA public key and decrypts the ciphertext using their RSA private key. Log results indicate that the signature verification returns a 'valid' status, and the received initial secret key is identical to the original input. A visualization of this step is shown in Figure 3, illustrating the transmission of the initial secret key using RSA.
2) **Exchanging ECDHE Public Keys**
Users A and B generate an ECDHE key pair using the Curve25519 curve. The resulting public keys are signed using RSASSA-PSS and encrypted using AES-256, with the initial shared secret key serving as the symmetric encryption key. Both parties transmit the ciphertext containing the ECDHE public key and its corresponding signature to the other party. Upon receipt, each party decrypts the message using the initial shared secret key and verifies the accompanying signature before proceeding. If the verification fails, the communication process is immediately terminated to prevent potential attacks or data leakage. A visualization of this step is presented in Figure 4, showing the encrypted ECDHE public key exchange process.
3) **Generating the Final Shared Secret Key**
The verified ECDHE public keys are then used to compute the final shared secret key using the ECDH algorithm over the Curve25519 curve. Log data confirms that the final shared secret key values are identical on both sides, demonstrating the successful and secure synchronization of the key exchange. Additionally, to preserve forward secrecy, each user deletes their ECDHE private key after the final shared secret key has been derived. This step is visualized in Figure 5, which illustrates the computation of the final shared secret key using ECDHE.
4) **Shared Secret Key**
The final shared secret key is employed as a symmetric key in the AES-256 algorithm to encrypt and decrypt messages exchanged between Users A and B. Encrypted messages are sent from User A to User B and vice versa, using this key. Testing results show that messages can be decrypted accurately without data loss or verification failures, confirming that the shared secret key functions reliably and consistently on both ends. This step is illustrated in Figure 6.

B. Security

Security testing was conducted to ensure that the developed protocol is capable of countering the threats defined in the threat model. A series of attack simulations, summarized in Table II, were carried out to evaluate the protocol's resilience against potential security attacks, in accordance with the adversary capabilities and security goals outlined in Section III.

1) **Compromise of the Main ECDHE Private Key**
Demonstrates forward secrecy and resistance to key recovery across sessions. The shared secret key value for each session is unique, while within the same session, the values on both sides are identical. The testing results prove that ECDHE Curve25519 exhibits cryptographic forward secrecy properties, as the shared key cannot be reconstructed in subsequent sessions without the ephemeral key. The test data used in this scenario can be seen more clearly in Table III.

2) **MITM Attack with Random Secret Key Modification**
In this scenario, it is assumed that User B's RSA public key has been leaked and is known to the attacker. The attacker attempts to intercept the communication between User A and User B by modifying the random secret key data sent by User A. User A sends a ciphertext containing the random secret key along with their digital signature to User B. The attacker intercepts the data, replaces the random secret key with a new value, and retains the original signature from User A (unchanged). The attacker then forwards the modified ciphertext to User B. Upon receiving the data, User B verifies the signature using User A's public key. However, since the data has been altered, the signature becomes invalid, and the verification process fails. The system rejects the data, proving that even though the attacker knows User B's RSA public key and can re-encrypt the new random secret key into the ciphertext, the signature cannot be forged without User A's private key. The system successfully detects the change in the data because the signature becomes invalid.

3) **MITM Attack Without Authentication**
This scenario tests the impact of disabling the digital authentication scheme (RSASSA-PSS) on communication security. In this test, it is assumed that User B's RSA public key has been leaked and is known to the attacker. With this condition, a MITM attack is carried out, similar to the previous scenario (modification of the random secret key), but without any active digital signature mechanism. User A sends the ciphertext containing the random secret key to User B without a digital signature. User B receives and processes the data without being able to verify the authenticity or integrity of the message since no signature mechanism is active. The communication proceeds even though unauthorized modifications have occurred. The results show that without the use of a digital signature, the system fails to detect data manipulation by the attacker. The attacker successfully inserts themselves into the communication path (man-in-the-middle) and alters the message content without detection.

4) **MITM Attack with Signature Modification**
In this scenario, it is assumed that the initial shared key has been compromised and is known to the attacker. The test observes whether the system can detect and reject a message that has been re-signed by an unauthorized party. User B sends a ciphertext containing their ECDHE public key and signature to User A. The attacker intercepts the data, modifies the signature with their own, and forwards the data to User A. User A receives the altered data and performs a signature verification process. Since the signature does not match the original one from User B, the verification process fails. The system detects that the data did not originate from the legitimate sender, and thus the data is rejected. The results show that the system successfully rejects data that has been re-signed by the attacker, effectively countering the MITM attack with signature modification. This confirms that the protocol enforces message authenticity and is resistant to MITM attacks involving signature replacement.

5) **MITM Attack with Modification of the ECDHE Public Key**
This scenario tests the system's resilience against a MITM attack where the attacker replaces User B's ECDHE public key with their own. In this test, the initial shared key is assumed to be compromised and known to the attacker. User B sends a ciphertext containing their ECDHE public key and signature to User A. The attacker intercepts the data, replaces the ECDHE public key with their own, without re-signing it, and forwards the modified ciphertext to User A. Upon receiving the data, User A verifies the signature using User B's public key. However, since the data has been modified (the ECDHE public key has been replaced), the signature is no longer valid. The system detects the discrepancy between the signature and the received data, causing the verification to fail and the data to be rejected. The results show that the system successfully detects data manipulation, even when the signature itself is not altered by the attacker. This demonstrates that the system guarantees data integrity and sender authentication, as any modification to the data results in an invalid signature. This demonstrates the protocol's capability to detect data integrity violations and authenticate the sender correctly, preventing MITM attacks involving key replacement.

6) **Replay Attack**
This scenario tests the protocol's resistance against replay attacks, in which an attacker attempts to reuse valid messages from a previous session. In this test, User A and User B first complete a standard handshake procedure, resulting in the generation of a valid shared session key using the ECDHE mechanism. The attacker captures the handshake message from User A, specifically the ciphertext containing the ECDHE public key. In a later session, the attacker replays this previously captured ciphertext without modification and sends it to User B. Upon receiving the message, User B attempts to process it as if it were part of a new handshake. However, because the session key depends on newly

generated ephemeral keys in each session, the replayed message fails to establish a valid session. Signature verification and decryption are unsuccessful, and the message is rejected. These results confirm that the protocol provides implicit replay protection by using fresh ephemeral keys in each session, thereby making all past messages cryptographically unusable in future sessions.

C. Performance

Performance testing was conducted to evaluate the efficiency of the protocol in terms of processing time and resource usage. In this study, the testing focused on comparing the proposed protocol with the baseline PrivateDH protocol in the session key establishment process. The results of this testing aim to assess whether the designed protocol is not only cryptographically secure but also efficient for deployment in real-world digital communication environments.

1) **Handshake Time**
   The handshake time was measured from User A's side (initiator), starting from the creation of the initial shared secret key until the final shared secret key is established. The time measurement on User B's side (responder) was not considered the primary reference, as it is influenced by the waiting time before receiving a message from the initiator. The measured handshake time per test session can be seen in Table IV and Figure 7.
2) **CPU Usage**
   CPU usage was measured during the handshake process on both user sides. The recorded value represents the average CPU usage percentage for each protocol and user. The CPU usage measurements per test session are presented in Table V and Figure 8.
3) **Memory Usage**
   Memory usage (in megabytes) was measured on both user sides during the handshake process. The memory usage measurements per test session are presented in Table VI and Figure 9.

V. Discussion

This section discusses the results of testing the proposed protocol implementation, which integrates ECDHE Curve25519, RSASSA-PSS, and AES-256 to enhance the PrivateDH protocol. The analysis comprehensively evaluates the protocol's functionality, security, and performance.

A. Effectiveness of ECDHE Curve25519 as a Replacement for DH

Research Question 1: *How can the integration of ECDHE Curve25519 replace traditional DH in generating secure and efficient session keys within the PrivateDH protocol?*

Based on the previous problem analysis, classical Diffie-Hellman (DH) suffers from high computational overhead due to its unbalanced encryption and decryption structure and the need for large parameter values (p and g) to maintain an adequate security level. In the testing environment used in this study, replacing DH with ECDHE Curve25519 significantly improved both the protocol's security and computational efficiency. The test results show the following:

1) **Ephemeral Key Exchange & Forward Secrecy**
   The resulting shared secret key is different for each communication session, and any compromise in one session does not affect the others. This confirms that ECDHE supports forward secrecy, unlike classical DH, due to its use of ephemeral private keys. The test results show that this protocol successfully maintains forward secrecy and prevents key recovery outside the ongoing session. This confirms that key compromise in one session does not jeopardize other sessions.
2) **Handshake Time Efficiency**
   The proposed protocol using ECDHE Curve25519 demonstrates a significantly faster handshake time compared to the baseline PrivateDH using 2048-bit DH. The average handshake time from user A's perspective was as follows:
   - Proposed Protocol (ECDHE Curve25519): 249.14ms
   - Baseline PrivateDH (DH 2048-bit): 14947.76ms
   - Baseline PrivateDH (DH 256-bit): 257.08ms

   This indicates a drastic performance improvement, with ECDHE Curve25519 completing the handshake approximately 60 times faster than DH 2048-bit. Even when compared with DH 256-bit, ECDHE remains consistently superior with smaller time variation, indicating promising efficiency and stability in session key generation under controlled conditions.
3) **CPU Utilization During Handshake**
   The average CPU usage during the handshake process for the proposed protocol was:
   - User A: 7.19%
   - User B: 0.45%

   In comparison, the baseline protocol using DH 2048-bit recorded:
   - User A: 0.29%
   - User B: 97.94%

   While the proposed protocol shows higher CPU usage on user A's side compared to the baseline, and significantly lower CPU usage on user B's side, this indicates a more balanced distribution of processing load relative to the highly asymmetric pattern observed in the baseline DH 2048-bit protocol. The extreme CPU burden on user B in the baseline suggests a potential performance bottleneck in real-world applications. Compared to DH 256-bit, the proposed protocol also proves more efficient on user B's side (0.45% vs 15.56%), highlighting Curve25519's lightweight nature despite offering stronger cryptographic security. The observed CPU usage patterns between the proposed protocol and the baseline DH implementations can be attributed to fundamental differences in their key exchange mechanisms. In the baseline DH 2048-bit configuration, the responder (User B) performs computationally intensive modular exponentiation operations and also handles the encryption and transmission of large Diffie-Hellman parameters (P and G). This results in an extremely high CPU load

on User B's side (~98%), which may also be partially influenced by system-level or measurement-specific factors. In contrast, ECDHE Curve25519 leverages efficient elliptic curve operations that are more symmetric between the two parties. Furthermore, the initiator (User A) in the proposed protocol handles signature generation and verification processes, leading to slightly higher CPU usage compared to the responder. These characteristics contribute to a more balanced distribution of computational load between the two endpoints. A boxplot analysis of User A's CPU usage revealed two low-value outliers at 4.2%, which fall below the lower bound (6.925) based on IQR calculations (Q1 = 7.6, Q3 = 8.05). These outliers are likely caused by temporary fluctuations in background system activity during testing and do not significantly affect the overall performance trend, as the majority of values are clustered near the median (7.85%). Nevertheless, this assessment is based on average CPU utilization, and detailed per-core utilization measurements were not conducted. Further analysis is necessary to comprehensively characterize CPU load patterns on both endpoints.

4) **Memory Consumption**
The average memory usage for the proposed protocol was approximately:
- User A: 73.60 MB
- User B: 73.69 MB

In comparison, the baseline PrivateDH protocol recorded:
- DH 2048-bit: 67.26 MB (A) and 66.53 MB (B)
- DH 256-bit: 66.99 MB (A) and 66.39 MB (B)

Although ECDHE Curve25519 incurs slightly higher memory usage (around 6–7 MB more), this increase is relatively small and acceptable given the substantial improvements in execution time and CPU usage. The additional memory consumption is mainly due to extra signature and validation operations that enhance security, and remains within reasonable limits for general-purpose computing systems in controlled environments.

B. Effectiveness of RSASSA-PSS in Preventing Identity Spoofing

Research Question 2: *How can RSASSA-PSS be used to authenticate users and prevent identification failures when identical ciphertext is received from different sources?*

One of the main weaknesses of the original PrivateDH protocol is its inability to distinguish between a legitimate sender and an attacker when identical ciphertexts are sent simultaneously by two different entities. The proposed protocol addresses this issue through the integration of RSASSA-PSS. Based on simulation testing, the following observations were made:

1) **MITM Scenario with Signature Modification**
In this scenario, an attacker attempted to modify a message sent by user B and re-sign it using the attacker's own private key. However, since user A only accepts and verifies signatures originating from user B using a previously known and trusted public key, the forged signature failed verification. As a result, the message was rejected and the communication was aborted. This confirms that RSASSA-PSS signatures cannot be forged without the legitimate sender's private key.

2) **MITM Scenario with ECDHE Public Key Modification**
In this scenario, even though the attacker successfully replaced the ECDHE public key in transit, the signature—still originating from user B—no longer matched the altered content. Consequently, the signature verification failed, and the communication was terminated. This demonstrates that any modification to signed content invalidates the signature, thereby preserving the integrity and authenticity of the message.

3) **MITM Scenario Without Authentication**
When RSASSA-PSS signatures were disabled in this scenario, the attacker was able to insert themselves into the communication and manipulate message content (e.g., the random secret key) without being detected. This highlights the critical importance of digital signatures in preventing unauthorized entities from impersonating legitimate users. Without authentication, the protocol becomes vulnerable to man-in-the-middle attacks and message tampering.

4) **Replay Attack Scenario**
In addition to relying on cryptographic freshness from ephemeral ECDHE key exchanges, the protocol also benefits from the use of AES-GCM, an AEAD (Authenticated Encryption with Associated Data) scheme. Since AES-GCM generates authentication tags that are tightly bound to both the session key and the ciphertext, any attempt to replay encrypted messages from previous sessions results in decryption failure. This is because the session keys derived via ECDHE are unique for every session, and reused ciphertext will produce invalid authentication tags under a new key. Therefore, even without explicit timestamps or session identifiers, the combined use of ephemeral keys and AEAD encryption ensures strong protection against replay attacks.

If the RSA private key of a user (e.g., User A) is compromised, an attacker can exploit this key to forge valid signatures. As a result, the attacker can impersonate another party (User B) and convince User A that the communication is with User B, while it is actually with the attacker. This scenario exemplifies a Key Compromise Impersonation (KCI) attack. Since the protocol's authentication relies heavily on RSA signatures, the compromise of a private key undermines the trust in authentication and enables impersonation without requiring compromise of the other party's keys. To mitigate this risk, it is advisable for the protocol to explicitly bind identities within the signature process and incorporate additional mechanisms that do not rely solely on long-term RSA keys. This approach may help mitigate the impact of a potential private key compromise. While the KCI scenario is described conceptually, its experimental validation is deferred to future work due to the complexity of simulating private key compromise in a controlled setting. Instead, mitigation strategies are provided to demonstrate awareness of this risk.

Furthermore, ephemeral keys are erased from memory

immediately after being used in the session key derivation process through the `del` command, minimizing the risk of residual sensitive data being retained in memory. All cryptographic operations are implemented using the cryptography library, which wraps OpenSSL and provides mitigations against timing attacks and cache-based side-channel attacks for core primitives such as AES, ECDHE, RSA, and HKDF [31]. Although side-channel leakage was not experimentally evaluated in this study, its potential impact was conceptually assessed, and corresponding hardening measures were incorporated into the design.

VI. Conclusion

Based on the analysis and experiments conducted, it can be concluded that the proposed protocol provides significant improvements in both security and efficiency compared to the original PrivateDH protocol. The main conclusions are as follows:

1) ECDHE Curve25519 effectively replaces DH with a more secure and efficient key exchange mechanism. The achieved forward secrecy prevents cross-session compromise, while its computational efficiency reduces the overhead compared to classical DH.
2) RSASSA-PSS provides robust digital authentication to cryptographically distinguish the sender's identity and prevent message tampering by adversaries. The protocol not only detects identical ciphertexts but also rejects messages with invalid signatures, thereby directly addressing a key weakness of PrivateDH.

Regarding suitability for modern deployments, the proposed protocol demonstrated promising computational efficiency and stability under controlled local network conditions. However, evaluations under varying network environments and on resource-constrained platforms such as mobile or IoT systems were not conducted in this study. These remain as important future work to more comprehensively validate the protocol's applicability in diverse, real-world deployment scenarios.

Acknowledgment

I would like to express my sincere gratitude to Universitas Mikroskil and to everyone who contributed, directly or indirectly, to the completion of this work. Finally, I am deeply grateful to my family and loved ones for their unwavering support, patience, and encouragement during the course of this thesis.

References

[1] A. Premana, G. Fitralisma, A. Yulianto, M. B. Zaman, and M. A. Wiryo, "Pemanfaatan teknologi informasi pada pertumbuhan ekonomi dalam era disrupsi 4.0," *J. Econ. Manag. (JECMA)*, vol. 2, no. 2, pp. 1–6, 2020.

[2] D. Wiryany, S. Natasha, and R. Kurniawan, "Perkembangan teknologi informasi dan komunikasi terhadap perubahan sistem komunikasi Indonesia," *J. Nomosleca*, vol. 8, no. 2, pp. 242–252, 2022.

[3] V. Sharma, A. Chauhan, H. Saxena, S. Mishra, and S. Bansal, "Secure file storage on cloud using hybrid cryptography," in *Proc. 5th Int. Conf. Inf. Syst. Comput. Netw. (ISCON)*, Oct. 2021, pp. 1–6.

[4] Ö. Aslan, S. S. Aktuğ, M. Ozkan-Okay, A. A. Yilmaz, and E. Akin, "A comprehensive review of cyber security vulnerabilities, threats, attacks, and solutions," *Electronics*, vol. 12, no. 6, p. 1333, 2023.

[5] F. R. S. Taka, "Secure communication by combined Diffe-Hellman key exchange based AES encryption and Arabic text steganography," *J. Inf. Hiding Multim. Signal Process.*, vol. 12, no. 4, pp. 186–198, 2021.

[6] S. Dusane, M. Iliyaz, and A. Suresh, "Implementation of E2EE using Python," in *Proc. Int. Conf. Emerg. Technol. Data Mining Inf. Secur. (IEMIS)*, Kolkata, India, 2022, pp. 635–642.

[7] K. K. Nalla, "Securing chat applications: Strategies for end-to-end encryption and cloud data protection," *World Journal of Advanced Engineering Technology and Sciences*, vol. 13, no. 2, pp. 528–540, 2024.

[8] T. Isobe and R. Ito, "Security analysis of end-to-end encryption for Zoom meetings," *IEEE Access*, vol. 9, pp. 90677–90689, 2021.

[9] A. Adithya, K. Kulkarni, and S. Saha, "Applications of RSA and AES256 in end-to-end encryption using Diffie-Hellman key exchange," *International Research Journal of Engineering and Technology (IRJET)*, vol. 9, no. 9, Sep. 2022.

[10] C. Gupta and N. S. Reddy, "Enhancement of security of Diffie-Hellman key exchange protocol using RSA cryptography," in *J. Phys.: Conf. Ser.*, vol. 2161, no. 1, p. 012014, 2022.

[11] A. Ghani, S. U. Jan, S. A. Chaudhry, R. Ahmad, and D. H. Kim, "MCDH-SLKAP: Modified computational Diffie-Hellman based secure and lightweight key agreement protocol for decentralized edge computing networks," *IEEE Access*, vol. 12, pp. 133923–133936, 2024.

[12] R. Patgiri, "privateDH: An Enhanced Diffie-Hellman Key-Exchange Protocol using RSA and AES Algorithm," *Cryptology ePrint Archive, Paper 2021/647*, 2021. [Online]. Available: https://eprint.iacr.org/2021/647

[13] E. Dritsas, M. Trigka, and P. Mylonas, "Performance and security analysis of the Diffie-Hellman key exchange protocol," in *Proc. 19th Int. Workshop Semantic Soc. Media Adapt. Pers. (SMAP)*, Nov. 2024, pp. 166–171.

[14] A. A. Ahmed and O. M. Barukab, "Unforgeable digital signature integrated into lightweight encryption based on effective ECDH for cybersecurity mechanism in Internet of Things," *Processes*, vol. 10, no. 12, p. 2631, 2022.

[15] O. İşler, "Implementation and Performance Evaluation of Elliptic Curve Cryptography over SECP256R1 on STM32 Microprocessor," *Cryptology ePrint Archive*, Paper 2024/1121, 2024. [Online]. Available: https://eprint.iacr.org/2024/1121

[16] Y. Yan, "The overview of elliptic curve cryptography (ECC)," in *J. Phys.: Conf. Ser.*, vol. 2386, no. 1, p. 012019, 2022.

[17] B. Arunkumar and G. Kousalya, "Secure and lightweight elliptic curve cipher suites in SSL/TLS," *Comput. Syst. Sci. Eng.*, vol. 40, no. 1, 2022.

[18] S. Amanlou, M. K. Hasan, and K. A. A. Bakar, "Lightweight and secure authentication scheme for IoT network based on publish–subscribe fog computing model," *Comput. Netw.*, vol. 199, p. 108465, 2021.

[19] S. Cohney, A. Kwong, S. Paz, D. Genkin, N. Heninger, E. Ronen, and Y. Yarom, "Pseudorandom black swans: Cache attacks on CTR_DRBG," in *Proc. IEEE Symp. Security Privacy (SP)*, May 2020, pp. 1241–1258.

[20] M. Rodríguez, J. Lázaro, U. Bidarte, J. Jiménez, and A. Astarloa, "A fixed-latency architecture to secure GOOSE and sampled value messages in substation systems," *IEEE Access*, vol. 9, pp. 51646–51658, 2021.

[21] S. H. Murad and K. H. Rahouma, "Implementation and performance analysis of hybrid cryptographic schemes applied in cloud computing environment," *Procedia Comput. Sci.*, vol. 194, pp. 165–172, 2021.

[22] C. Paar and J. Pelzl, *Understanding Cryptography*. Berlin, Germany: Springer-Verlag, 2010.

[23] E. Rescorla, "The Transport Layer Security (TLS) Protocol Version 1.3," *Internet Engineering Task Force (IETF)*, RFC 8446, Aug. 2018.

[24] T. Perrin, "The Noise Protocol Framework," Noise Specification, Revision 34, Jul. 2018. [Online]. Available: https://noiseprotocol.org

[25] J. Alwen, S. Coretti, and Y. Dodis, "The double ratchet: Security notions, proofs, and modularization for the Signal protocol," in *Proc. Int. Conf. Theory Appl. Cryptographic Techniques (EUROCRYPT)*, Cham: Springer, Apr. 2019, pp. 129–158.

[26] D. J. Bernstein, "Curve25519: New Diffie-Hellman speed records," in *Lect. Notes Comput. Sci.*, vol. 3958, pp. 207–228, Springer, 2006.

[27] A. Langley, M. Hamburg, and S. Turner, "RFC 7748: Elliptic Curves for Security," *IETF*, Jan. 2016.

[28] J. Jonsson and B. Kaliski, "Public-Key Cryptography Standards (PKCS) #1: RSA Cryptography Specifications Version 2.2," RFC 8017, Nov. 2016.

[29] National Institute of Standards and Technology (NIST), "FIPS PUB 197: Advanced Encryption Standard (AES)," Gaithersburg, MD, USA, 2001.
[30] National Institute of Standards and Technology (NIST), "Recommendation for Random Number Generation Using Deterministic Random Bit Generators," NIST Special Publication 800-90A, Gaithersburg, MD, USA, 2015.
[31] Y. Yarom, D. Genkin, and N. Heninger, "CacheBleed: a timing attack on OpenSSL constant-time RSA," *J. Cryptogr. Eng.*, vol. 7, no. 2, pp. 99–112, 2017.
[32] S. Serengil and A. Ozpinar, "LightDSA: A Python-Based Hybrid Digital Signature Library and Performance Analysis of RSA, DSA, ECDSA and EdDSA in Variable Configurations, Elliptic Curve Forms and Curves," arXiv preprint arXiv:2505.23773, 2025.
[33] H. Krawczyk, "SIGMA: The 'SIGn-and-MAc' approach to authenticated Diffie-Hellman and its use in the IKE protocols," in *Proc. Annual Int. Cryptology Conf. (CRYPTO)*, Berlin, Germany: Springer, 2003, pp. 400–425.



Ardi Saputra is a Master's candidate in the Information Technology program at Universitas Mikroskil, Medan, Indonesia, where he is researching cryptographic approaches to enhance the PrivateDH key exchange protocol (email: ardisaputraa12@gmail.com).



Ronsen Purba is a Lecturer in Master of Information Technology (MTI) Program at Universitas Mikroskil, Medan, Indonesia (email: ronsen@mikroskil.ac.id).