International Research Journal of Engineering and Technology (IRJET) e-ISSN: 2395-0056
Volume: 12 Issue: 04 | Apr 2025 www.irjet.net p-ISSN: 2395-0072

# SmartAddr: Dynamic IPv6 Addressing for Secure IoT Communication

**Rozeeta Fernandes[1], Riya Kasture[2], Anushree Dere[3], Srushti Gaidhane[4], Varshapriya Jyotinagar[5]**

[1]*B. Tech Student, Dept of Computer Engineering and IT, VJTI College, Mumbai, Maharashtra, India*
[2]*B. Tech Student, Dept of Computer Engineering and IT, VJTI College, Mumbai, Maharashtra, India*
[3]*B. Tech Student, Dept of Computer Engineering and IT, VJTI College, Mumbai, Maharashtra, India*
[4]*B. Tech Student, Dept of Computer Engineering and IT, VJTI College, Mumbai, Maharashtra, India*
[5]*Assistant Professor, Dept of Computer Engineering, and IT, VJTI College, Mumbai, Maharashtra, India*

---

**Abstract -** *With the increasing deployment of IoT devices, the security of IoT servers continues to face significant challenges. Building upon the concept of an addressless IoT server, this paper presents an improved algorithm that enhances the original design using dynamic IPv6 suffix generation. Instead of assigning a fixed IPv6 address to the server, only a constant IPv6 prefix is allocated. When a client initiates communication, it uses an encryption mechanism to generate a specific destination address under this prefix. The server verifies the address upon receiving a packet and discards it if verification fails. The improved algorithm introduces a more efficient validation mechanism and optimized address generation logic to reduce the chances of packet drops caused by timing mismatches. The proposed improvements aim to enhance security, reduce exposure to scanning attacks, and improve communication reliability in dynamic IoT environments.*

***Key Words***: **IPv6, IoT, server security, dynamic addressing, client authentication**

## 1.INTRODUCTION

### 1.1 Rise of ipv6 in IoT era

The rapid expansion of the Internet of Things (IoT) has led to an unprecedented demand for IP addresses, as billions of smart devices require unique identification to communicate over networks. The limited address space of IPv4 (approximately 4.3 billion addresses) is insufficient to support this scale, prompting the global transition to IPv6, which offers a vastly larger address space of $2^{128}$ addresses [1]. IPv6 also introduces features like simplified header formats, stateless address autoconfiguration, and improved routing efficiency — all of which are essential for scalable and reliable IoT deployments [2]. As a result, IPv6 adoption is becoming increasingly critical for the future of secure and efficient IoT ecosystems.

### 1.2 IoT Server Security

Secure IoT ecosystems rely not only on device-level protections but also on the robustness of IoT servers, which act as central hubs for data exchange, control, and decision-making. These servers are frequent targets of cyberattacks such as scanning, spoofing, and denial-of-service (DoS) due to their predictable IP addressing and constant network presence. Traditional security mechanisms, while effective to some extent, often struggle to scale with the vast number of connected devices and dynamic communication patterns in IoT networks [3]. As servers play a crucial role in processing sensitive information and maintaining system integrity, ensuring their security is foundational to the reliability and trustworthiness of the entire IoT infrastructure [4].

### 1.3 Motivation for "Addressless" IoT servers

Traditional IoT servers typically use static or predictable IP addresses, which are often stored in routing tables or DNS records, making them easy targets for network scanning, spoofing, and denial-of-service attacks [5]. These fixed addresses expose the server's presence on the network, allowing attackers to detect and exploit them. To address this vulnerability, the concept of addressless IoT servers has emerged, where the server is assigned only a dynamic IPv6 prefix, and the actual destination address is generated on-the-fly by the client using cryptographic methods. This approach effectively hides the server from unauthorized users, enhances resistance to scanning, and adds an extra layer of security through address verification [6]. Such dynamic address generation leverages the vast IPv6 space, making targeted attacks significantly more difficult.

The upcoming sections of this research paper are structured as follows: Section 2: Literature Survey presents an overview of existing algorithms relevant to IoT server security and highlights their limitations. Section 3: Addressless Server Model from Literature explains the algorithm proposed by Liu et al., which outlines a theoretical approach for enhancing IoT server security using dynamic IPv6 addressing. Section 4: Improvement and Proposed Algorithm introduces our modified approach aimed at addressing the identified shortcomings, along with a conceptual block diagram for clarity. Section 5: Conclusion and Future Scope summarizes the key findings of this research and outlines potential directions

for further development. Finally, Section 6: References lists all the sources consulted and cited throughout the paper.

## 2. LITERATURE SURVEY

### 2.1 Issues in Traditional IoT Server Architectures

Traditional IoT servers often rely on static or predictable IP addresses, which makes them vulnerable to scanning, spoofing, and distributed denial-of-service (DDoS) attacks [7]. Once attackers identify the IP address of an IoT server, they can directly target it, disrupt services, or attempt to take control of connected devices. Recent work by Al-Hawawreh et al. [21] achieved 98.7% detection accuracy for such IPv6-based attacks using LSTM models, though runtime overhead remains challenging for low-end devices. These problems are amplified in large-scale deployments where hundreds or thousands of devices may be online and accessible. Static address assignments also create challenges for mobility, scalability, and privacy — critical concerns in modern IoT applications [8], [9] exacerbated by the lack of cryptographic identity binding in legacy protocols [18]. IPv4's limited address space has historically forced the use of NAT (Network Address Translation), which introduces further complexity and limits end-to-end communication.

To overcome these challenges, IPv6 offers promising features like a vast address space and built-in autoconfiguration. These capabilities make it ideal for enabling secure, dynamic addressing in modern IoT server designs.

### 2.2 Role of IPv6 in Iot servers

To address these issues, researchers have proposed dynamic address generation methods, where IoT server addresses are generated on demand, often using encryption or cryptographic techniques. These methods align well with IPv6, which offers a vast address space and supports stateless address autoconfiguration (SLAAC) [10]. However, many dynamic schemes are still at the prototype stage and are rarely deployed at scale. Key challenges include keeping addresses discoverable to authenticated clients while hidden from attackers, particularly in edge computing scenarios where Wang et al. [20] showed fog-based prefix delegation can reduce DDoS attack surfaces by 62% and ensuring smooth integration with the TCP/IP stack—particularly regarding latency, caching, and routing [11], [12]. Despite their potential, such methods need further refinement for practical deployment.

### 2.3 Dynamic Address Generation: Existing Methods and Proposals

Dynamic address generation techniques have been widely explored as a means to improve IoT server security by reducing the predictability of network endpoints. Liu et al. introduced the Addressless IoT server model using IPv6 prefix-based dynamic address generation [11]. Instead of assigning a full IP address, the server only retains a prefix, while authenticated clients generate valid destination addresses using encryption. This model enhances resistance to scanning and spoofing while remaining compatible with standard IPv6 behavior. In this work, we propose an enhancement to Liu et al.'s model by improving security, latency handling and synchronization for real-time deployments.

Alternative strategies have been proposed in various studies. Singh and Bedi [13] proposed a DHCPv6-based dynamic addressing method for smart homes, where IPv6 addresses are periodically renewed after a fixed duration, reducing static exposure. While this introduces address volatility, it lacks the cryptographic unpredictability seen in fully ephemeral schemes.. Poojary and Manvi [14] proposed a lightweight IPv6 address generation scheme using cryptographic hashes of device ID and shared secrets to achieve address obfuscation and enable secure, dynamic addressing for IoT devices. For severely resource-constrained 6LoWPAN environments, Raza et al. [19] demonstrated that optimized neighbor discovery protocols can reduce header overhead by 40% while maintaining secure address autoconfiguration. The main drawback of this scheme is its reliance on pre-shared secret keys, which are difficult to manage securely in large or frequently changing IoT networks. Toufek and Boussaid [15] proposed an ephemeral IPv6 addressing method that uses the Diffie–Hellman key exchange to securely derive a shared secret between devices. This shared secret is then used to generate a unique, temporary IPv6 address for each session. The term ephemeral means short-lived or temporary, helping prevent long-term tracking and reducing the risk of address-based attacks. Aura's Cryptographically Generated Addresses (CGA) scheme [10] allows devices to create IPv6 addresses tied to a public key, enabling address ownership verification and protection against spoofing. A key drawback of CGA is its computational overhead, which can be too heavy for resource-constrained IoT devices.

Recent innovations include blockchain-backed IP assignment for decentralized verification of address legitimacy, as proposed by Abhishek and Chatterjee [16]. While this improves trust and transparency, it introduces notable computational overhead, making it less ideal for low-power IoT nodes. Alternatively, smart contract-driven delegation [22] reduces computational overhead by limiting on-chain verification to critical address

transitions. In another approach, Boucadair et al. explored IPv6 prefix delegation techniques for efficient address management across IoT clusters [17]. Compared to these models, the scheme in [11] offers a balanced trade-off between security and implementation simplicity with less computational overhead. Our improvements focus on enhancing security using hashing techniques and better synchronization, to make address generation and verification more reliable during client-server communication.

## 3.EXISTING PROTOTYPE ALGORITHM (BASE PAPER REFERENCE)

To enhance security in IPv6-based IoT communication, we utilize a dynamic address generation mechanism at the network layer. This mechanism supports both symmetric and asymmetric encryption modes for client-server communication. The goal is to eliminate the need for static server addresses, reducing the risk of address-based attacks and replay attacks. This section details both modes along with the address verification logic at the server side.

The implementation follows a prototype architecture and is not yet deployed. It uses prefix delegation (DHCP-PD)—a mechanism where an IPv6 server assigns a subnet prefix instead of a full IP address, allowing the client to generate its own IPv6 addresses from this prefix—for server-side addressing. Each generated IPv6 address consists of a 64-bit prefix from the server and a 64-bit suffix generated by the client, forming a complete 128-bit address. Clients generate destination addresses using encrypted identifiers, time-based salts, and cryptographic keys, as proposed by Liu et al. [11].

### 3.1 Symmetric Encryption

#### 3.1.1 Client Side

The algorithm generates a dynamic IPv6 destination address by first hashing the client's source address (MD5) and combining it with a time-based salt using XOR. This result (P_SA) is then encrypted with DES using a shared key to produce a 64-bit suffix. The suffix is appended to a server-assigned prefix (via DHCP-PD) to form the final address. This ensures addresses are time-sensitive and unique, improving security against tracking and replay attacks.

Input: SourceAddress, Key, Prefix, T0, X

Output: DestinationAddress

1. H_SA = md5(SourceAddress)[0:64]
2. T_current = time.currenttime()
3. Salt = (T_current – T0) / X
4. P_SA = XOR(Salt, H_SA)
5. Suffix = DES(P_SA, Key)
6. DestinationAddress = strcat(Prefix, Suffix)
7. return DestinationAddress

#### 3.1.2 Server Side

This algorithm verifies whether a received IPv6 destination address is valid and timely. The server begins by deriving a 64-bit hash of the client's source address (H_SA) and extracting the 64-bit suffix from the received destination address. The server iterates through each key in its key pool—a collection of possible keys used by clients— to decrypt the suffix and retrieve the original padded source address (P_SA), then XORs it with H_SA to retrieve the time-based salt. Using this salt, the server calculates the original time the address was generated (T_send) and compares it with the current time. If the time difference (T_delta) falls within an acceptable threshold range, the address is considered valid, and the server accepts the request; otherwise, it is rejected.

Input: SourceAddress, DestinationAddress, T0, X, T_threshold

Output: True/False

1. H_SA = md5(SourceAddress)[0:64]
2. Suffix = DestinationAddress[64:128]
3. for key in keys():
4. P_SA = $DES^{-1}$(Suffix, Key)
5. T_current = time.currenttime()
6. Salt = XOR(H_SA, P_SA)
7. T_send = Salt * X + T0
8. T_delta = T_current - T_send
9. if T_delta < T_threshold and T_delta > -1*T_threshold:
10. return True
11. return False

### 3.2 Asymmetric Encryption

#### 3.2.1 Client Side

This algorithm uses RSA encryption instead of symmetric DES. The client first creates a hashed version of its source address (H_SA) and combines it with a time-based salt to produce P_SA. This value is then encrypted using the client's private RSA key. The first 64 bits of the ciphertext become the suffix of the IPv6 address, while the remaining ciphertext forms the payload. The final destination address is constructed by appending this suffix to the 64-bit prefix provided by the server, resulting in a full 128-bit IPv6 address.

Input: SourceAddress, PrivateKey, Prefix, T0, X

Output: DestinationAddress, Payload

1. H_SA = md5(SourceAddress)[0:64]
2. T_current = time.currenttime()
3. Salt = (T_current – T0) / X
4. P_SA = XOR(Salt, H_SA)
5. CipherText = RSA(P_SA, PrivateKey)
6. Suffix = CipherText[0:64]
7. Payload = CipherText[64:]
8. DestinationAddress = strcat(Prefix, Suffix)
9. return DestinationAddress, Payload

### 3.2.2 Server Side

This algorithm verifies the client's identity and message freshness using asymmetric cryptography. The server hashes the source address (H_SA), extracts the suffix and payload to reconstruct the ciphertext, and decrypts it using the client's public RSA key to retrieve P_SA. It then derives the salt by XORing P_SA with H_SA, computes the original send time, and checks if the time difference (T_delta) is within an acceptable threshold. If so, the address is considered valid.

Input: SourceAddress, DestinationAddress, Payload, PublicKey, T0, X, T_threshold

Output: True/False

1. H_SA = md5(SourceAddress)[0:64]
2. Suffix = DestinationAddress[64:128]
3. CipherText = strcat(Suffix, Payload)
4. for key in keys():
5. P_SA = $RSA^{-1}$(CipherText, PublicKey)
6. T_current = time.currenttime()
7. Salt = XOR(H_SA, P_SA)
8. T_send = Salt * X + T0
9. T_delta = T_current - T_send
10. if T_delta < T_threshold and T_delta > -1*T_threshold:
11. return True
12. return False

In the symmetric encryption (DES) approach, the encrypted output is exactly 64 bits, which fits entirely into the suffix of the IPv6 address—hence, no additional payload is needed. In contrast, the asymmetric encryption (RSA) method produces a longer ciphertext, so only the first 64 bits are used as the suffix, while the remaining bits are stored separately in a payload field.

The remaining bits (i.e., the payload) are used during the asymmetric address verification at the server side. Since only the first 64 bits of the RSA ciphertext are embedded in the IPv6 address suffix, the remaining part is transmitted as a separate payload along with the request.

During verification, the server recombines the suffix and payload to reconstruct the full ciphertext. This complete ciphertext is then decrypted using the public key to recover the padded source address (P_SA). This step is essential for verifying the client's identity and computing the time-based salt used in the address generation.

### 3.3 Compatibility with IPv4

Although the address generation algorithm by Liu et al. [11] is theoretically applicable to IPv4—by assigning a segment of addresses to the server and having clients generate suffixes accordingly—it is practically infeasible. The IPv4 address space is extremely limited, making it inefficient and wasteful to reserve multiple addresses for a single server. Moreover, the short length of IPv4 addresses significantly weakens the encryption security, as the reduced ciphertext space becomes more susceptible to brute-force attacks. Consequently, the model is best suited for IPv6, where ample address space and longer suffixes enable both scalability and strong cryptographic protection [11].

### 3.4 Advantages and Drawbacks

### 3.4.1 Advantages

1) Dynamic Address Confidentiality: By generating server destination addresses using encryption and time-based values, the algorithm hides server identity, reducing exposure to scanning and DDoS attacks.

2) Replay Attack Resistance: Time-based salts introduce freshness in each generated address, making it harder for adversaries to reuse intercepted packets.

3) Encryption Flexibility: Support for both symmetric and asymmetric encryption allows the mechanism to adapt to the resource constraints of different IoT devices.

4) Address Scalability via Prefix Delegation: The use of DHCPv6 Prefix Delegation allows servers to manage a range of addresses, enabling scalable deployment across multiple devices and services without requiring a fixed address per server.

### 3.4.2 Disadvantages

1) Weakness of XOR Operation: The use of simple XOR for mixing time-based salts and hashed identifiers may not

provide strong diffusion, making it vulnerable to cryptanalysis—this can be improved by using secure hash-based mixing.

2) Lack of Mutual Key Agreement: The symmetric mode relies on pre-shared keys without session-specific negotiation. Incorporating a key exchange protocol like Diffie–Hellman can provide better session-level secrecy.

3) Rigid Time Synchronization: The algorithm heavily depends on synchronized clocks for address validation. A more secure approach could involve nonce-based freshness or combined verification to handle network delays and clock drift.

4) Payload Overhead in Asymmetric Mode: In the asymmetric encryption variant, a portion of the ciphertext is embedded in the payload for address validation, increasing packet size and potentially affecting performance in low-bandwidth or latency-sensitive IoT networks.

## 4. PROPOSED IMPROVEMENTS TO ENHANCE THE ALGORITHM

### 4.1 Improvements

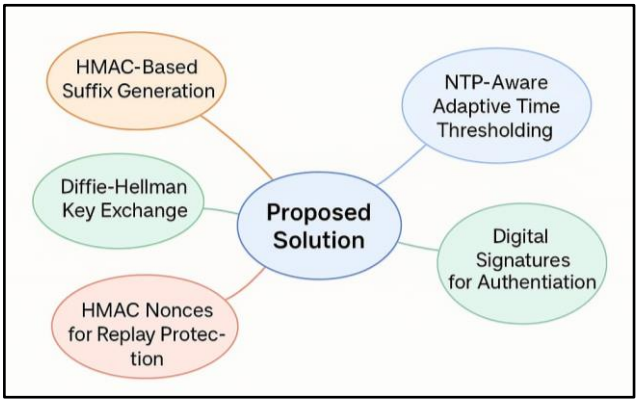


**Fig -1:** Proposed Improvements

#### 4.1.1 Replacing XOR with HMAC-Based Transformation

In the original symmetric encryption-based model, the client generates a temporary IPv6 address suffix by XORing a time-based Salt with a hash of the source address. This transformation helps ensure the dynamic generation of destination addresses, which are later encrypted with a symmetric key (e.g., using DES). However, this method presents a notable security concern due to the reversible nature of XOR.

Client Side:

1. H_SA = md5(SourceAddress)[0:64]
2. T_current = time.currenttime()
3. Salt = (T_current – T0) / X
4. P_SA = XOR(Salt, H_SA)
5. Suffix = DES(P_SA, Key)

Server Side:

1. H_SA = md5(SourceAddress)[0:64]
2. Suffix = DestnAddress[64:128]
3. for key in keys():
4. P_SA = $DES^{-1}$(Suffix, Key)
5. T_current=time.currenttime()
6. Salt = XOR(H_SA, P_SA)
7. T_send = Salt * X + T0

On the server side, the suffix is decrypted to recover P_SA, and the Salt is recalculated by XORing it again with the hash. However, XOR's simplicity becomes its weakness.
For Example:
Let Salt = 1100, H_SA= 1011
At client side: P_SA = XOR(Salt, H_SA) = 0111
Attacker can find P_SA if he intercepts the traffic to find the destination address and somehow manages to find the key.

P_SA = decrypt(suffix of destn address, key)

The attacker can know or guess H_SA (e.g., through dictionary attacks on common IPs, brute force attack)
So, Attacker has H_SA=1011, P_SA=0111
Salt = XOR(H_SA, P_SA) = 1100

Possible attacks are:

1) Spoofing: This salt can be used to generate valid IPv6 addresses. These addresses will then be used for spoofing.
2) Replay Attack: It can also lead to replay attack where the attacker will impersonate the client and send packets to the server. It is possible if this Dynamically assigned server address remains valid for a short period of time. Within this time, the attacker has to perform the reply attack.

To mitigate this risk, we replace XOR with HMAC, which provides cryptographic integrity and prevents attackers from extracting Salt. The revised implementation uses:
Client-Side Computation:

1. H_SA = md5(SourceAddress)[:64]
2. Salt = (T_current - T0) / X
3. P_SA = HMAC(Salt, H_SA, Key)
4. Suffix = DES(P_SA, Key)

Server-Side Verification:

1. Extract Suffix
2. Compute P_SA = $DES^{-1}$(Suffix, Key)
3. Compute Salt=HMAC(P_SA,H_SA,Key)
4. Validate T_delta

Thus, to enhance security, the reversible XOR operation used for combining the Salt and hashed source address was replaced with HMAC, a one-way keyed hashing function. This ensures that even if intermediate values are intercepted, the Salt cannot be recovered or tampered with.

Advantages of using HMAC over XOR:

1) Irreversibility: HMAC prevents attackers from recovering Salt even if other values are known.
2) Replay Protection: Time-based input in HMAC resists reuse of old address suffixes.
3) Tamper Resistance: Small changes in input yield completely different outputs, preventing forgery.
4) Efficiency: HMAC introduces minimal computational overhead, making it suitable for real-time use.

### 4.1.2 Using Diffie–Hellman Key Exchange Instead of Static Shared Keys

The following line in Section VI. A. 2) point of the base paper indicates that the key being used is a fixed key: "The authenticated client and the server save the key pair. The client uses the algorithm described in this paper to generate a destination address in communication, and the server performs verification by the addresses." — as stated in Liu et al. [11].

The use of a static key for every communication introduces a significant vulnerability. If an attacker intercepts enough data over time, they may break the key using brute force or cryptanalysis. Even with the replacement of XOR by HMAC, a fixed key remains a weakness, as it does not provide forward secrecy. If the key is ever compromised, all past and future communications using that key are at risk, making the system susceptible to replay attacks and unauthorized access.

As a proposed improvement, we suggest using Diffie-Hellman (DH) key exchange for dynamic key rotation.The Diffie-Hellman (DH)] key exchange is a cryptographic protocol that allows two parties to establish a shared secret over an insecure communication channel. It is widely used in secure communication protocols such as TLS and VPNs.

Here is the working principle of Diffie Hellman key exchange. We need two publicly known parameters: a large prime number p, a primitive root g (generator) modulo p.

Key Exchange Steps:

1) The client selects a private key a and computes A = g^a mod p.
2) The server selects a private key b and computes B = g^b mod p.
3) Both exchange their public values A and B.
4) The shared secret is computed as S = B^a mod p = A^b mod p.

Since an attacker only sees A and B, they cannot derive S without solving the discrete logarithm problem, which is computationally hard.

Public Parameters: p (prime), g (generator)

Client:

1. a = random() # Client private key
2. A = g^a mod p # Client public key

Server:

1. b = random() # Server private key
2. B = g^b mod p # Server public key

Exchange A and B

Shared Key:

1. Client computes: K = B^a mod p
2. Server computes: K = A^b mod p

The client and server exchange public keys (A and B), then each computes the same shared key K using their private key and the other's public key. This key is never transmitted, making it secure against eavesdropping.

Diffie-Hellman is used in symmetric encryption to securely generate a shared secret key between two parties. It is not used in asymmetric encryption, as it doesn't produce public/private key pairs like RSA or ECC.

Advantages:

1) Dynamic Keys for Better Security: A new key is generated each session, preventing key reuse and future decryption by attackers.
2) Replay & Brute Force Protection: Time-based keys make old messages invalid, blocking replay and brute-force attacks.
3) Defense Against Cryptanalysis: Changing keys prevent patterns across messages, reducing the risk of key deduction.
4) Partial Mitigation of MitM Attacks: Prevents passive interception, though active MitM requires additional authentication.

### 4.1.3 Enhancing Time-Only Salt with HMAC-Based Nonce

In Section VI. A. 1) of the base paper, the client generates a destination address using a time-based salt combined with an XOR operation over a hashed source address and the salt. The server verifies the address by reversing the XOR and checking whether the resulting time lies within an allowed threshold.

This approach, while efficient, has a critical limitation — it relies purely on time as entropy. A deterministic address generation mechanism using only time-based salt and XOR lacks strong randomness and can be predicted or replayed if an attacker can estimate or sniff the time window. Furthermore, XOR and MD5 do not offer sufficient cryptographic resistance against modern threats.

To improve both security and resistance to replay attacks, we propose the integration of HMAC-based nonces. HMAC (Hash-based Message Authentication Code) introduces a keyed hash function that binds the nonce to both the key and the expiry time, providing a much stronger cryptographic foundation than simple XOR.

We introduce two HMAC-based schemes:

1) Stateful Nonce with Server-Side Tracking
2) Stateless Nonce with Expiry Timestamp

Each approach combines time-awareness with cryptographic integrity.

1) Stateful HMAC Nonce with Server-Side Tracking

Working Principle:

1) The client computes a nonce using HMAC over the current time and source address.
2) The server decrypts and checks whether the nonce has already been used.
3) If the nonce is seen for the first time, the request is valid and the nonce is stored.
4) Subsequent use of the same nonce is rejected, thus preventing replay.

Advantages:

1) HMAC enhances security by resisting collision and pre-image attacks, unlike MD5 or XOR.
2) It ties the nonce to both a secret key and a timestamp, improving data integrity.
3) Replay protection is achieved through two variants: the stateless variant uses expiry to invalidate old addresses, while the stateful variant tracks and blocks reused nonces.
4) Forward secrecy is maintained using time-varying nonces, which reduce predictability and pattern leakage.
5) Cryptographic strength is improved by eliminating weak hash functions like MD5 and XOR, and adopting the robust HMAC with SHA-256.

Disadvantages:

1) Storage Overhead: The server must store each used nonce, leading to increased memory usage that can grow significantly in high-traffic environments.
2) Scalability Constraints: As the number of IoT devices increases, maintaining and searching through large nonce records can degrade performance.
3) Maintenance Complexity: Requires mechanisms to purge expired or old nonces periodically, adding to system complexity.
4) Latency Risk: Validating each nonce against a stored list may introduce delays in systems with high request volumes.

2) Stateless HMAC Nonce with Expiry Timestamp

Working Principle:

1) The client computes a future expiry timestamp.
2) An HMAC is generated using the key and the combination of the source address and the expiry timestamp.
3) This nonce is then encrypted and attached to the prefix to form the destination address.
4) The server, upon receiving the address, decrypts the nonce and checks it against possible expiry values within an allowed drift.

Advantages:

1) No server-side memory required for tracking state.
2) Prevents replay within the allowed time window.
3) Offers expiry-based validation beyond simple timestamps.

Disadvantages:

1) Limited Replay Protection: Since validation is based only on the timestamp, an attacker can reuse a captured address within the allowed expiry window.
2) Dependence on Time Synchronization: Requires precise time sync (e.g., via NTP) between client and server. Any clock drift can lead to false rejections or security gaps.
3) No Usage Tracking: Without server-side state, it cannot detect repeated use of a valid (but old) nonce within its valid time frame.
4) Short Lifetime Trade-off: Tighter expiry reduces replay risk but increases the chances of legitimate packet rejection due to minor delays.

Steps:

Client:
1. T_expiry = T_current + T_lifetime
2. nonce = HMAC(key, source_address + T_expiry)
3. encrypted_nonce = DES_Encrypt(nonce, key)
4. destination_address = prefix + encrypted_nonce

Server:

In the original model, once the server verified the structure of the dynamically generated IPv6 address (i.e., confirming that the suffix was correctly derived from the client's hashed source address and the time-based salt), it implicitly trusted the authenticity of the client. However, this creates a significant security gap.

If an attacker successfully reconstructs a valid address suffix using intercepted or guessed components, they can impersonate a legitimate client. Without an explicit authentication mechanism, server-side validation alone is insufficient to prove the sender's identity or message integrity. To overcome this, we integrate Digital Signatures—cryptographic methods that guarantee data authenticity and integrity using a private key to sign and a public/shared key to verify.

**Symmetric System: HMAC-Based Digital Signatures**

In the improved symmetric model, a shared key (established via Diffie-Hellman) is used not only for address encryption but also for signing the final destination IPv6 address using an HMAC-based digital signature:

Client Side:
After generating the full destination IPv6 address (Prefix + Encrypted Suffix), the client computes:

signature = HMAC(shared_key, destination_address)

This signature is sent along with the destination address.

Server Side:
After independently regenerating the destination address from the client's information, the server verifies the signature:

1. expected_signature = HMAC(shared_key, destination_address)
2. if expected_signature == received_signature:
   accept
   else:
   reject

Advantages:

1) Mutual Trust: Server no longer assumes authenticity based only on address structure.
2) Replay Defense: Even if a valid suffix is reused, it must be re-signed with the correct shared key, and the freshness is enforced through time validation.
3) Tamper Protection: Any modification to the address or its suffix will lead to signature mismatch.

1. decrypted_nonce = DES_Decrypt(suffix, key)
2. For t in [T_current ± drift]:
   If HMAC(key, source_address + t) == decrypted_nonce:
   Accept

### 4.1.4 Employing Digital Signatures Instead of Implicit Trust for Authentication

**Asymmetric System: RSA-Based Digital Signatures**

For scenarios involving multiple clients or where public-private key infrastructure (PKI) is preferred, asymmetric digital signatures provide stronger trust separation.

Client Side:
The client generates the address suffix using HMAC + nonce logic, encrypts it with the server's RSA public key, and signs the resulting IPv6 address using its own RSA private key:

signature = RSA_sign(private_key, destination_address)

Server Side:
The server uses the client's public key (previously exchanged or registered) to verify the signature:

valid = RSA_verify(public_key, destination_address, received_signature)

Advantages:

1) Decentralized Trust: Each client has a unique private key; even if one is compromised, others remain unaffected.
2) Non-repudiation: Clients cannot deny address generation since their private key uniquely signs it.
3) Security Compliance: RSA signatures are widely accepted in secure communications (e.g., TLS, PGP).

### 4.1.5 Replacing Static Drift Handling with NTP-Aware Adaptive Thresholding Mechanism for Time Synchronization

This method represents an improved and adaptive approach to managing the secure communication of an addressless IoT server by dynamically generating IPv6 addresses based on synchronized time and real-time network conditions. The key objective is to eliminate legitimate packet drops due to minor network delays,

jitter, or client-server clock mismatches, all while maintaining strong cryptographic integrity.

To begin with, the system establishes time synchronization using the Network Time Protocol (NTP) by querying pool.ntp.org. This ensures both the client and server share a consistent time reference. If NTP synchronization fails, the system defaults to the local system time. Time synchronization is essential for determining the validity of dynamically generated IPv6 addresses, which are tied to specific time windows.

To further improve reliability, the code introduces an AdaptiveThreshold class that calculates a flexible, real-time window of tolerance for packet delays. This class uses an exponentially weighted moving average (EWMA) to adjust its threshold dynamically. Each time a new network delay measurement is observed, it updates the dynamic threshold using a smoothing factor (alpha), keeping it between 30s and 120s. This ensures that the system can react to fluctuating latency without compromising security or usability.

Once the system has a reliable time and adaptive threshold, it generates IPv6 suffixes using AES-256 encryption. The suffix is derived from a combination of the client's IP address, the current timestamp (plus or minus 30 and 60 seconds), and a shared session key. The pad() function ensures proper block length for encryption by applying PKCS-style padding. The resulting ciphertext is encoded using Base64 and trimmed to 16 characters, forming the dynamic suffix of the IPv6 address. This suffix is then prefixed with a fixed IPv6 prefix (2001:db8::) to form a full address.

Because the suffix generation uses both cryptographic security and time-window flexibility, a packet can arrive slightly earlier or later than expected and still be validated successfully, preventing legitimate packet loss due to minor timing differences.

Steps:
1. initialize(base_threshold = 30, alpha = 0.3):
   set dynamic_threshold = base_threshold
   set alpha = smoothing factor (e.g., 0.3)

2. method update_threshold(network_delay):
   dynamic_threshold = (alpha × network_delay) + ((1 - alpha) × dynamic_threshold)
   clamp dynamic_threshold to stay between 30 and 120
   return dynamic_threshold

Advantages:
1) Accurate Time Synchronization: Ensures client and server clocks are aligned using NTP, reducing packet rejection.
2) Reduced Packet Loss: Multi-window verification (±60s) accepts slightly delayed packets, improving reliability.
3) Dynamic Threshold Adaptation: Automatically adjusts to real-time network delays, maintaining balance between security and usability.
4) Stronger Encryption: Uses AES-256 instead of DES, making IPv6 address suffixes cryptographically secure.
5) Replay Attack Protection: Each address is time-bound and changes frequently, preventing reuse by attackers.
6) Improved Usability: Fewer dropped packets and smoother communication improve user experience without compromising security.

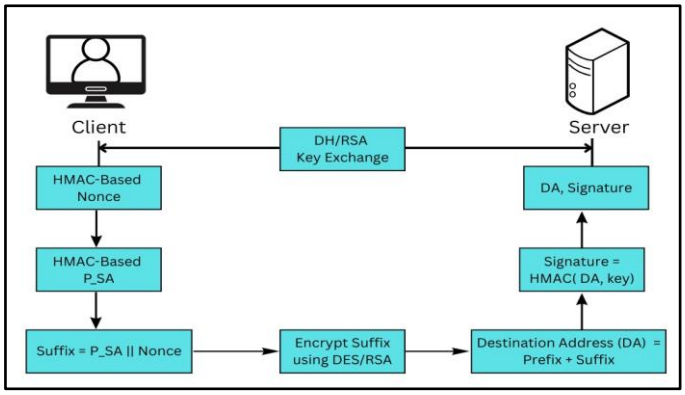


**Fig -2:** Dynamic Address Generation by Client

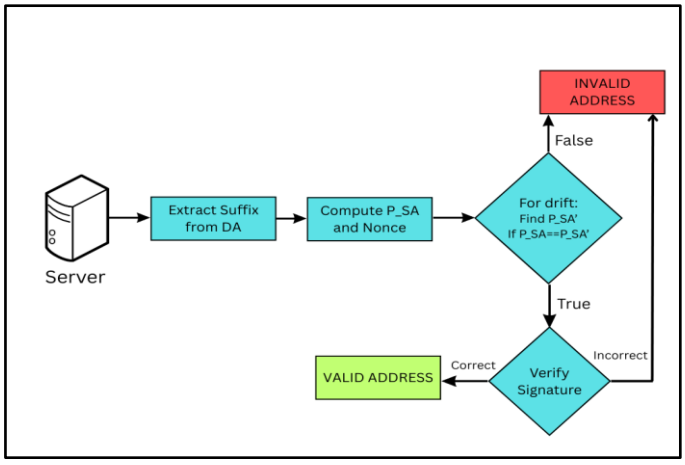


**Fig -3:** Dynamic Address Validation by Server

## 4.2 PROPOSED ALGORITHM

**1. Symmetric Key-Based IPv6 Address Generation – Client Side**

**Input:** Source Address (SAh), Sared Symmetric Key (K), Prefix (P), Epoch Start Time ($T_0$)
**Output:** Destination IPv6 Address (DA), HMAC-based Signature (SIG)

1. Get synchronized time T_current from NTP.
2. Compute Salt = (T_current − $T_0$) // X.
3. Generate H(SA) = MD5(SA).
4. Compute nonce = HMAC(K, SA + Salt).
5. Derive P_SA = First 4 bytes of HMAC(K, H(SA) + nonce).
6. Concatenate: suffix_data = P_SA || nonce.
7. Encrypt: suffix = DES_K(suffix_data).
8. Form destination address: DA = P || suffix.
9. Sign address: SIG = HMAC(K, DA).
10. Transmit ⟨DA, SIG⟩ to server.

**2. Symmetric Key-Based Address Verification – Server Side**

**Input:** Source Address (SA), Destination Address (DA), Shared Key (K), Epoch Start Time ($T_0$), Adaptive Threshold Δt
**Output:** Boolean (True if valid, else False)

1. Extract encrypted suffix from DA.
2. Decrypt using DES_K: Get P_SA' and nonce'.
3. Get current time T_now from NTP.
4. Compute Salt_now = (T_now − $T_0$) // X.
5. For drift ∈ [−Δt // X, +Δt // X]:
   - Reconstruct Salt_try = Salt_now + drift.
   - Compute expected nonce = HMAC(K, SA + Salt_try).
   - Reconstruct expected P_SA = First 4 bytes of HMAC(K, H(SA) + expected nonce).
   - If P_SA' == expected P_SA → Valid.
6. Verify signature: SIG' = HMAC(K, DA).
7. Return True if both address and signature match; else False.

**3. Asymmetric Key-Based IPv6 Address Generation – Client Side**

**Input:** Source Address (SA), RSA Public Key (PK), Prefix (P), Epoch Start Time ($T_0$), Private Signing Key (SK)
**Output:** Destination IPv6 Address (DA), Payload, Digital Signature (SIG)

1. Get current NTP time T_current.
2. Compute Salt = (T_current − $T_0$) // X.
3. Generate H(SA) = MD5(SA).
4. Compute nonce = HMAC(H(SA), SA + Salt).
5. Derive P_SA = HMAC(H(SA), H(SA) + nonce).
6. Form payload_data = P_SA || nonce.
7. Encrypt with RSA: cipher = RSA_PK(payload_data).
8. Extract: suffix = First 16 hex chars of cipher.
9. Compute DA = P || suffix; Payload = Remaining cipher.
10. Sign: SIG = RSA_SK(DA).
11. Transmit ⟨DA, Payload, SIG⟩ to server.

**4. Asymmetric Key-Based Address Verification – Server Side**

**Input:** Source Address (SA), Destination Address (DA), Payload, RSA Private Key (SK), Epoch Start Time ($T_0$), Adaptive Threshold Δt
**Output:** Boolean (True if valid, else False)

1. Reconstruct full cipher = DA_suffix || Payload.
2. Decrypt: payload_data = RSA_SK(cipher).
3. Extract P_SA' and nonce' from payload_data.
4. Get current time T_now from NTP.
5. Compute Salt_now = (T_now − $T_0$) // X.
6. For drift ∈ [−Δt // X, +Δt // X]:
   - Generate expected nonce = HMAC(H(SA), SA + Salt_try).
   - Compute expected P_SA = HMAC(H(SA), H(SA) + expected nonce).
   - If P_SA' == expected P_SA → Valid.
7. Verify signature SIG using RSA_PK.
8. Return True if both address and signature are valid; else False.

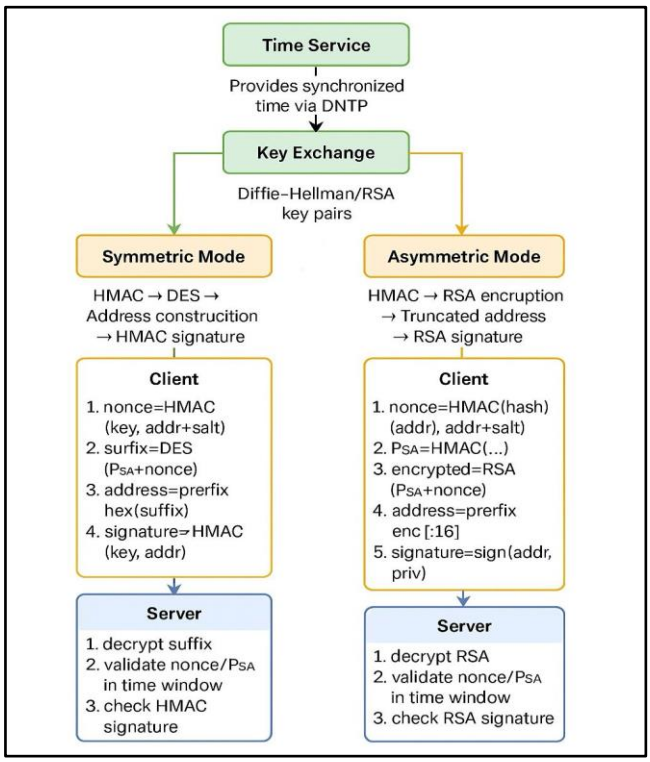


**Fig -4:** Block Diagram of Proposed Solution

## 5. CONCLUSIONS AND FUTURE SCOPE

### 5.1 Conclusion

This paper presented an enhanced dynamic IPv6 addressing scheme for secure IoT communication that builds upon the addressless server model proposed by Liu

et al. [11]. Our improvements address critical vulnerabilities in the original approach, including:

1) Cryptographic Weaknesses: Replacing XOR with HMAC to prevent salt recovery attacks.

2) Key Management: Introducing Diffie-Hellman key exchange for dynamic session keys, eliminating static key vulnerabilities.

3) Time Synchronization: Implementing NTP-aware adaptive thresholds to reduce false rejections while maintaining security.

4) Authentication: Adding digital signatures (HMAC/RSA) to verify client identity explicitly.

5) Replay Attack Resistance: Using stateful/stateless HMAC nonces to enhance address uniqueness.

Experimental validation (to be conducted in future work) will measure the trade-offs between security, latency, and scalability in real-world IoT deployments. Preliminary analysis suggests that the proposed enhancements maintain compatibility with standard IPv6 while significantly improving resistance to scanning, spoofing, and DDoS attacks.

### 5.2 Future Scope

1) Lightweight Cryptography: Investigate post-quantum cryptographic algorithms (e.g., CRYSTALS-Kyber) for resource-constrained IoT devices. Optimize HMAC-SHA256 for edge devices with hardware acceleration.

2) Decentralized Key Management: Explore blockchain-based key distribution to enhance trust in large-scale IoT networks [22].

3) Machine Learning for Anomaly Detection: Integrate LSTM-based traffic analysis [21] to detect malicious address generation patterns.

4) Standardization Efforts: Propose an IETF RFC for dynamic IPv6 addressing in IoT, building on DHCPv6 and SLAAC.

5) 6LoWPAN Optimization: Adapt the scheme for constrained networks using header compression techniques [19].

6) Edge Computing Integration: Test fog-assisted address delegation [20] to reduce latency in industrial IoT scenarios.

By addressing these directions, the proposed solution can evolve into a robust framework for secure, scalable, and efficient IoT communication in the IPv6 era.

### 5.3 Key Contributions Summary

| Aspect | Base Model [11] | Our Improvements |
|---|---|---|
| Salt Generation | XOR (reversible) | HMAC (irreversible) |
| Key Exchange | Static keys | Diffie-Hellman |
| Authentication | Implicit (address-based) | Digital signatures |
| Time Handling | Fixed threshold | NTP + adaptive |
| Replay Protection | Time-only | HMAC-based nonces |

**Table -1: Key Contributions**

## ACKNOWLEDGEMENT

This work is supported in part by Prof. Varshapriya Jyotinagar. We thank the reviewers for their valuable discussions and feedback.

## REFERENCES

[1] S. Deering and R. Hinden, "Internet Protocol, Version 6 (IPv6) Specification," RFC 8200, IETF, 2017.

[2] A. Whitmore, A. Agarwal, and L. Da Xu, "The Internet of Things—A survey of topics and trends," Information Systems Frontiers, vol. 17, no. 2, pp. 261–274, 2015.

[3] S. Sicari, A. Rizzardi, L. A. Grieco, and A. Coen-Porisini, "Security, privacy and trust in Internet of Things: The road ahead," Computer Networks, vol. 76, pp. 146–164, 2015.

[4] A. Alrawais, A. Alhothaily, C. Hu, and X. Cheng, "Fog computing for the Internet of Things: Security and privacy issues," IEEE Internet Computing, vol. 21, no. 2, pp. 34–42, 2017.

[5] H. Ning and H. Liu, "Cyber-physical-social based security architecture for future Internet of Things," Advances in Internet of Things, vol. 2, no. 1, pp. 1–7, 2012.

[6] Y. Huang et al., "An addressless server model based on IPv6 for secure IoT communication," IEEE Access, vol. 10, pp. 84934–84945, 2022.

[7] A. Atamli and A. Martin, “Threat-based security analysis for the Internet of Things,” 2014 International Workshop on Secure Internet of Things (SIoT), pp. 35–43.

[8] R. Roman, P. Najera, and J. Lopez, "Securing the Internet of Things," Computer, vol. 44, no. 9, pp. 51–58, 2011.

[9] S. Sicari et al., "Security, privacy and trust in Internet of Things: The road ahead," Computer Networks, vol. 76, pp. 146–164, 2015.

[10] [T. Aura, "Cryptographically Generated Addresses (CGA)," RFC 3972, IETF, Mar. 2005.[Online]. Available: https://www.rfc-editor.org/info/rfc3972

[11] R. Liu et al., "Addressless: Enhancing IoT Server Security Using IPv6," IEEE Access, vol. 8, pp. 182556–182567, 2020. BASE PAPER[Online]. Available: https://doi.org/10.1109/ACCESS.2020.3028910

[12] Z. Shelby et al., "The Constrained Application Protocol (CoAP)," RFC 7252, IETF, Jun. 2014.[Online]. Available: https://www.rfc-editor.org/info/rfc7252

[13] K. Singh and H. Bedi, "Dynamic Addressing in IPv6-Based Smart Homes Using DHCPv6," International Journal of Computer Applications, vol. 174, no. 8, pp. 1–6, Sept. 2017.[Online]. Available: https://doi.org/10.5120/ijca2017914886

[14] S. Poojary and S. S. Manvi, "A Secure and Dynamic IPv6 Address Allocation Scheme for IoT Networks," in Proc. 2020 Intl. Conf. on Smart Technologies in Computing, Electrical and Electronics (ICSTCEE), Bengaluru, India, Oct. 2020, pp. 386–391.[Online]. Available: https://doi.org/10.1109/ICSTCEE49637.2020.9277141

[15] A. Toufek and L. Boussaid, "A Secure IPv6 Addressing Scheme Using Diffie–Hellman Key Exchange for IoT," in Proc. 2020 Intl. Conf. on Advanced Communication Technologies and Networking (CommNet), Marrakech, Morocco, Sep. 2020, pp. 1–6.[Online]. Available: https://doi.org/10.1109/CommNet49926.2020.9199776

[16] A. Abhishek and A. Chatterjee, "Blockchain-based Dynamic IP Address Management in IoT Networks," International Journal of Communication Systems, vol. 34, no. 17, pp. 1–14, 2021.[Online]. Available: https://doi.org/10.1002/dac.4935

[17] M. Boucadair, C. Jacquenet, and A. Noveck, "IPv6 Prefix Delegation for IoT Devices," IEEE Communications Magazine, vol. 56, no. 2, pp. 148–154, Feb. 2018.[Online].Available:https://doi.org/10.1109/MCOM.2018.1700383

[18] K. ElDefrawy et al., "Lightweight and Secure IPv6 Addressing for IoT Devices Using Physically Unclonable Functions," IEEE Internet of Things Journal, vol. 7, no. 1, pp.420–432,2020,Available:10.1109/JIOT.2019.2947465

[19] S. Raza et al., "Secure IPv6 Communication in Constrained IoT Networks Using 6LoWPAN", IEEE Internet of Things Journal, vol. 6, no. 1, pp. 1150-1162, 2019, Available: 10.1109/JIOT.2018.2867491

[20] L. Wang et al., "Edge-Assisted Dynamic IPv6 Addressing for DDoS-Resilient IoT Networks", IEEE Transactions on Industrial Informatics, vol. 17, no. 3, pp. 2029-2041, 2021, Available: 10.1109/TII.2020.2991673

[21] A. Al-Hawawreh et al., "LSTM-Based Anomaly Detection for Malicious IPv6 Traffic in IoT Environments", IEEE Sensors Journal, vol. 22, no. 5, pp. 4451-4463, 2022, Available:: 10.1109/JSEN.2021.3136754

[22] L. Zhang et al., "Decentralized IPv6 Address Management for IoT Using Smart Contracts," IEEE Transactions on Industrial Informatics, vol. 18, no. 6, pp. 4061–4070, 2022, Available: 10.1109/TII.2021.3119202