Network Working Group
Internet-Draft
Intended status: Informational
Expires: 16 September 2026

F. Denis
Fastly Inc.

15 March 2026

Methods for IP Address Encryption and Obfuscation
draft-denis-ipcrypt-13

Abstract

This document specifies secure, efficient methods for encrypting IP addresses for privacy-preserving storage, logging, and analytics. Unlike truncation, which destroys data irreversibly, these methods are reversible with the encryption key while providing strong privacy guarantees.

Four modes are defined: ipcrypt-deterministic (format-preserving, IP-address output), ipcrypt-pfx (prefix-preserving, native address size), ipcrypt-nd and ipcrypt-ndx (non-deterministic with random tweaks). All support high-performance processing at network speeds and produce interoperable results across implementations.

Discussion Venues

This note is to be removed before publishing as an RFC.

Source for this draft and an issue tracker can be found at https://github.com/jedisct1/draft-denis-ipcrypt.

Status of This Memo

Table of Contents

1. Introduction

IP addresses are personally identifiable information requiring protection, yet common anonymization approaches have fundamental limitations. Truncation (zeroing parts of addresses) irreversibly destroys data while providing variable privacy levels; a /24 mask may obscure one user or thousands depending on network allocation. Hashing produces non-reversible outputs unsuitable for operational tasks such as abuse investigation. Ad-hoc encryption schemes often lack rigorous security analysis and have limited interoperability.

This document addresses these deficiencies by specifying secure, efficient, and interoperable methods for IP address encryption and obfuscation.

This specification addresses concerns raised in [RFC7624] regarding confidentiality when sharing data with third parties. Unlike existing practices that obscure addresses, these methods provide well-defined security properties, which are discussed throughout this document and summarized in Section 8.

1.1. Use Cases and Motivations

Organizations handling IP addresses require mechanisms to protect user privacy while maintaining operational capabilities. Generic encryption systems expand data unpredictably, lack compatibility with network tools, and are not designed for high-volume processing. The specialized methods in this specification address these requirements:

* Efficiency and Compactness: All variants operate on 128 bits, achieving single-block encryption speed required for network-rate processing. Non-deterministic variants add only 8-16 bytes of tweak overhead.

* High Usage Limits: Non-deterministic variants safely handle massive volumes: approximately 4 billion operations for ipcrypt-nd and 18 quintillion for ipcrypt-ndx per key, without degrading security. Generic encryption often requires complex key rotation schemes at lower thresholds.

* Format Preservation: The ipcrypt-deterministic and ipcrypt-pfx variants produce valid IP addresses rather than arbitrary ciphertext, enabling encrypted addresses to pass through existing network infrastructure, monitoring tools, and databases without modification. The ipcrypt-pfx variant uniquely preserves network prefix relationships while maintaining the original address type and size, enabling network-level analytics while protecting individual address identity (see Section 5.2).

* Interoperability: All conforming implementations produce identical results, enabling data exchange between different systems, vendors, and programming languages.

These specialized encryption methods enable several use cases:

* Privacy Protection: Prevents exposure of sensitive user information to third parties in logs, analytics data, and network measurements ([RFC6973]). The key holder retains decryption capability.

* Correlation Attack Resistance: Non-deterministic variants leverage random tweaks to hide patterns and enhance confidentiality (see Section 7).

* Privacy-Preserving Analytics: Encrypted IP addresses can be used directly for counting unique clients, rate limiting, or deduplication without revealing original values. This addresses the anonymization requirements for DNS query data sharing outlined in [RSSAC040]. The ipcrypt-pfx variant preserves network prefixes for network-level analytics, while other methods scramble network hierarchy.

* Third-Party Integration: Encrypted IP addresses can serve as privacy-preserving identifiers when interacting with untrusted services, cloud providers, or external platforms.

The following examples demonstrate how the same IP addresses transform under each method:

* Format-preserving: Valid IP addresses, same input always produces same output:

```
192.168.1.1   -> d1e9:518:d5bc:4487:51c6:c51f:44ed:e9f6
192.168.1.254 -> fd7e:f70f:44d7:cdb2:2992:95a1:e692:7696
192.168.1.254 -> fd7e:f70f:44d7:cdb2:2992:95a1:e692:7696  # Same output
```

* Prefix-preserving: Maintains network structure, same prefix when IPs share prefix:

```
192.168.1.1   -> 251.81.131.124
192.168.1.254 -> 251.81.131.159       # Prefix is preserved
172.16.69.42  -> 165.228.146.177
```

* Non-deterministic: Larger output, different each time:

```
192.168.1.1   -> f0ea0bbd...03aa9fcb
192.168.1.254 -> 620b58d8...2ff8086f
192.168.1.254 -> 35fc2338...25abed5d  # Same input, different outputs
```

These methods achieve optimal efficiency for their security goals: each variant uses the minimum ciphertext expansion and fewest cryptographic operations needed for its properties (format preservation, prefix preservation, or correlation resistance). [IPCRYPT].

For implementation guidelines, see Section 9.

1.2. Relationship to IETF Work

This section is to be removed before publishing as an RFC.

This document does not conflict with active IETF working group efforts. While the IETF has produced several RFCs related to privacy ([RFC6973], [RFC7258], [RFC7624]), there is no current standardization effort for IP address encryption methods. This specification complements existing IETF privacy guidance by providing implementation methods.

The AES-based cryptographic primitives used align with IETF cryptographic recommendations, and the document follows IETF formatting and terminology conventions where applicable.

2. Terminology

The key words “MUST”, “MUST NOT”, “REQUIRED”, “SHALL”, “SHALL NOT”, “SHOULD”, “SHOULD NOT”, “RECOMMENDED”, “NOT RECOMMENDED”, “MAY”, and “OPTIONAL” in this document are to be interpreted as described in BCP 14 [RFC8174] when, and only when, they appear in all capitals, as shown here.

Throughout this document, the following terms and conventions apply:

* IP Address: An IPv4 or IPv6 address as defined in [RFC4291].

* IPv4-mapped IPv6 Address: An IPv6 address format (::FFFF:a.b.c.d) used to represent IPv4 addresses within the IPv6 address space, enabling uniform processing of both address types.

* 16-Byte Representation: A fixed-length representation used for both IPv4 (via IPv4-mapped IPv6) and IPv6 addresses.

* Block Cipher: A deterministic cryptographic algorithm that encrypts fixed-size blocks of data (128 bits with AES) using a secret key.

* Permutation: A bijective function where each distinct input maps to a unique output, ensuring reversibility.

* Pseudorandom Function (PRF): A deterministic function that produces output computationally indistinguishable from random values.

* Tweakable Block Cipher (TBC): A block cipher that accepts an additional non-secret parameter (tweak) along with the key and plaintext, allowing domain separation without changing keys.

* Tweak: A non-secret, additional input to a tweakable block cipher that further randomizes the output.

* Deterministic Encryption: Encryption that always produces the same ciphertext for a given input and key.

* Non-Deterministic Encryption: Encryption that produces different ciphertexts for the same input due to the inclusion of a randomly sampled tweak.

* Prefix-Preserving Encryption: An encryption mode where IP addresses from the same network produce ciphertexts that share a common encrypted prefix, maintaining network relationships while obscuring actual network identities.

* Birthday Bound: The point at which collisions become statistically likely in a random sampling process, approximately 2^(n/2) operations for n-bit values.

* (Input, Tweak) Collision: A scenario where the same input is encrypted with the same tweak. This reveals that the input was repeated but not the input's value.

3. IP Address Conversion

This section describes the conversion of IP addresses to and from a 16-byte representation used by ipcrypt-deterministic, ipcrypt-nd, and ipcrypt-ndx. The ipcrypt-pfx method differs by maintaining native address sizes—4 bytes for IPv4 and 16 bytes for IPv6—to preserve network structure (see Section 6).

3.1. Converting to a 16-Byte Representation

3.1.1. IPv6 Addresses

IPv6 addresses (128 bits) are converted directly using network byte order as specified in [RFC4291].

Example:

```
IPv6 Address:           2001:0db8:85a3:0000:0000:8a2e:0370:7334
16-Byte Representation: [20 01 0d b8 85 a3 00 00 00 00 8a 2e 03 70 73 34]
```

3.1.2. IPv4 Addresses

IPv4 addresses (32 bits) are mapped using the IPv4-mapped IPv6 format as specified in [RFC4291]:

```
IPv4 Address:           192.0.2.1
16-Byte Representation: [00 00 00 00 00 00 00 00 00 00 FF FF C0 00 02 01]
```

Note on representation equivalence: For all ipcrypt variants, encrypting an IPv4 address and encrypting its IPv4-mapped IPv6 form (::FFFF:a.b.c.d) produce the same encrypted output.

This behavior is intentional: in the intended use cases, IP addresses function as identifiers, and producing a single encrypted identifier for the same endpoint across representations avoids ambiguity.

3.2. Converting from a 16-Byte Representation to an IP Address

Conversion algorithm:

1. Examine the first 12 bytes of the 16-byte representation
2. If they match the IPv4-mapped prefix (10 bytes of 0x00 followed by 0xFF, 0xFF):
    * Interpret the last 4 bytes as an IPv4 address in dotted-decimal notation
3. Otherwise:
    * Interpret the 16 bytes as an IPv6 address in colon-hexadecimal notation

4. Generic Constructions

This specification defines two generic cryptographic constructions:

1. 128-bit Block Cipher Construction:
    * Used in deterministic encryption (see Section 5)
    * Operates on a single 16-byte block
    * Example: AES-128 treated as a permutation
2. 128-bit Tweakable Block Cipher (TBC) Construction:
    * Used in non-deterministic encryption (see Section 7)
    * Accepts a key, a tweak, and a message
    * The tweak must be uniformly random when generated
    * Reuse of the same tweak on different inputs does not compromise confidentiality

Valid options for implementing a tweakable block cipher include, but are not limited to:

* SKINNY (see [SKINNY])
* DEOXYS-TBC (see [DEOXYS-TBC])
* KIASU-BC (see Section 9.9 for implementation details)
* AES-XTS (see Section 7.4.2 for usage)

Implementers MUST choose a cipher that provides robust resistance against related-tweak and other cryptographic attacks.

5. Deterministic Encryption

Deterministic encryption applies a 128-bit block cipher directly to the 16-byte representation of an IP address, producing the same ciphertext for the same input and key.

Deterministic encryption is appropriate when:

* Duplicate IP addresses need to be detected in encrypted form (e.g., for rate limiting)

* Storage space is critical (produces only 16 bytes output)

* Format preservation is required (output remains a valid IP address)

* Correlation of the same address across records is acceptable

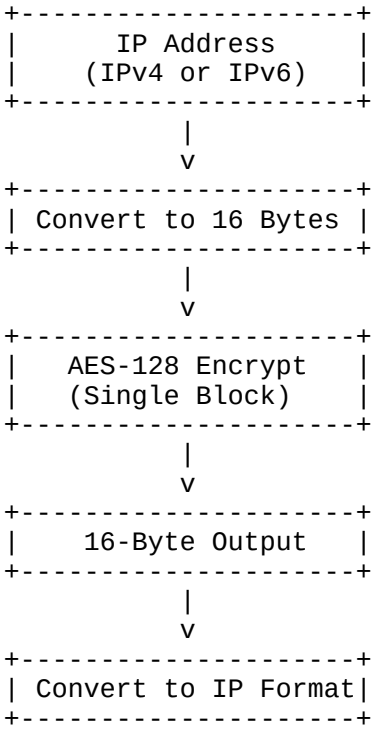
All instantiations (ipcrypt-deterministic, ipcrypt-pfx, ipcrypt-nd, and ipcrypt-ndx) are invertible. For non-deterministic modes, the tweak must be preserved along with the ciphertext to enable decryption.

Implementation details are provided in Section 9.

5.1. ipcrypt-deterministic

The ipcrypt-deterministic instantiation employs AES-128 in a single-block operation. The key MUST be 16 bytes (128 bits). As AES-128 is a permutation, each distinct input maps to a unique ciphertext, producing a valid IP address representation.

Test vectors are provided in Appendix A.1.



5.2. Format Preservation and Limitations

5.2.1. Network Hierarchy Preservation

Most encryption methods in this specification scramble network hierarchy, with the notable exception of ipcrypt-pfx:

* ipcrypt-deterministic, ipcrypt-nd, and ipcrypt-ndx: These methods completely scramble IPv4 and IPv6 prefixes in the encrypted output. Addresses from the same subnet appear unrelated after encryption, and geographic or topological proximity cannot be inferred.

* ipcrypt-pfx: This method preserves network prefix relationships in the encrypted output. Addresses from the same subnet share a common encrypted prefix, enabling network-level analytics while protecting the actual network identity. The encrypted prefixes themselves are cryptographically transformed and unrecognizable without the key.

5.2.2. Format Preservation Without Prefix Preservation

For methods other than ipcrypt-pfx, IPv4 addresses are typically encrypted as IPv6 addresses.

IPv4 format preservation without prefix preservation (maintaining IPv4 addresses as IPv4 in deterministic or non-deterministic modes) is not specified in this document and is generally discouraged due to the limited 32-bit address space, which significantly reduces encryption security.

If IPv4 format preservation is absolutely required despite the security limitations, implementers SHOULD implement a Format-Preserving Encryption (FPE) mode such as the FF1 algorithm specified in [NIST-SP-800-38G] or FAST [FAST].

5.2.3. Preserving Metadata for Analytics

Organizations requiring network metadata for analytics have two options:

1. Use ipcrypt-pfx to preserve network structure within the encrypted addresses, enabling network-level analysis while keeping actual network identities encrypted.

2. For non-prefix-preserving modes (ipcrypt-deterministic, ipcrypt-nd, ipcrypt-ndx), extract and store metadata (geographic location, ASN, network classification) as separate fields before encryption.

Both approaches provide advantages over IP address truncation, which provides inconsistent protection and irreversibly destroys data.

When auxiliary information such as AS number and geographical location is required, it SHOULD be stored as metadata at the time of logging. Performing these mappings later may yield different results as network allocations and routing information change over time. This recommendation applies to all encryption modes: even with ipcrypt-pfx, the preserved network structure does not retain these additional attributes.

6. Prefix-Preserving Encryption

Prefix-preserving encryption maintains network structure in encrypted IP addresses. Addresses from the same network produce encrypted addresses that share a common prefix, enabling privacy-preserving network analytics while preventing identification of specific networks or users.

Unlike standard encryption that completely scrambles addresses, prefix-preserving encryption enables network operators to:

* Detect traffic patterns from common networks without knowing which specific networks

* Perform network-level rate limiting on encrypted addresses

* Implement DDoS mitigation while preserving user privacy

* Analyze network topology without accessing raw IP addresses

This mode balances privacy with analytical capability: individual addresses remain encrypted and network identities are cryptographically transformed, but network relationships remain visible through shared encrypted prefixes.

6.1. Prefix-Preserving Construction

The encryption process achieves prefix preservation through a bit-by-bit transformation that maintains consistency across addresses with shared prefixes. For any two IP addresses sharing the first N bits, their encrypted forms also share the first N bits. This property holds recursively for all prefix lengths.

The algorithm operates as follows:

1. Process each bit position sequentially from most significant to least significant

2. For each bit position, extract the prefix (all bits processed so far) from the original IP address

3. Apply a pseudorandom function (PRF) that takes the padded prefix as input to generate a cipher bit

4. XOR the cipher bit with the original bit at the current position to produce the encrypted bit

5. The encrypted bit depends deterministically on the prefix from the original IP, ensuring identical prefixes always produce identical encrypted prefixes

This construction ensures:

* Identical prefixes always produce identical transformations for subsequent bits

* Different prefixes produce cryptographically distinct transformations

* The transformation is deterministic yet cryptographically secure

* Network relationships are preserved while actual network identities remain encrypted

The algorithm maintains native address sizes: IPv4 addresses remain 4 bytes (32 bits) and IPv6 addresses remain 16 bytes (128 bits).

6.2. Concrete Instantiation: ipcrypt-pfx

The ipcrypt-pfx instantiation implements prefix-preserving encryption using a pseudorandom function based on the XOR of two independently keyed AES-128 encryptions.

6.2.1. Encryption Process

The encryption uses a pseudorandom function based on the XOR of two independently keyed AES-128 encryptions. The 32-byte key is split into two independent 16-byte AES-128 keys (K1 and K2).

For each bit position (processing from MSB to LSB):

1. Prefix Padding: The prefix (all bits processed so far from the original IP address) is padded to 128 bits using the format zeros || 1 || prefix_bits, where:

   * The prefix bits are extracted from the most significant bits of the original IP address

   * A single 1 bit serves as a delimiter at position prefix_len_bits

   * The prefix bits are placed immediately after the delimiter, from high to low positions

   * For an empty prefix (processing the first bit), this produces a block with only a single 1 bit at position 0

2. Pseudorandom Function Computation: The padded prefix is encrypted independently with both K1 and K2, producing two 128-bit outputs (e1 and e2). The final PRF output is computed as e = e1 e2.

3. Bit Encryption: The least significant bit is extracted from the PRF output as the cipher bit, which is then XORed with the original bit at the current position to produce the encrypted bit.

Complete pseudocode implementation is provided in Section 9.6.

6.2.2. Key Requirements

The two 16-byte halves of the 32-byte key (K1 and K2) MUST NOT be identical. Using identical values for K1 and K2 (e.g., repeating the same 16 bytes twice) causes the XOR operation to cancel out, returning the original IP address unchanged.

When the 32-byte key is randomly sampled from a uniform distribution, the probability that K1 = K2 is statistically negligible.

6.2.3. Security Properties

The ipcrypt-pfx construction improves upon earlier designs such as CRYPTO-Pan through enhanced cryptographic security:

* Sum-of-Permutations: The XOR of two independently keyed AES-128 permutations provides 128-bit PRF security [SUM-OF-PRPS][PATARIN-2008][PATARIN-H-COEFFICIENTS], with distinguishing advantage growing on the order of q/2^128 for q queries. This construction ensures robust security even for massive-scale deployments.

* Prefix-based context isolation: shift-and-append updates make each bit depend on the full prefix history and ensure fresh PRF input each round.

* Fixed network-order representation and explicit bit indexing avoid endianness pitfalls.

Note: Prefix-preserving encryption intentionally reveals network structure to enable analytics. Organizations requiring complete address obfuscation should use non-prefix-preserving modes.

6.2.4. Implementation Considerations

Key implementation characteristics:

* Computational Requirements:

  - IPv4: 64 AES-128 operations per address (2 encryptions × 32 bits)

  - IPv6: 256 AES-128 operations per address (2 encryptions × 128 bits)

* Performance Optimizations:

  - Caching encrypted prefix values (e1 and e2) significantly improves performance for addresses sharing common prefixes

  - The encryption algorithm is inherently parallelizable since AES computations for different bit positions are independent

  - The padded prefix computation can be optimized by maintaining state across iterations: instead of recomputing the padded prefix from scratch for each bit position, implementations can shift the previous padded prefix left by one bit and insert the next input bit.

7. Non-Deterministic Encryption

Non-deterministic encryption enhances privacy by ensuring that the same IP address produces different ciphertexts each time it is encrypted, preventing correlation attacks that plague deterministic schemes. This is achieved through tweakable block ciphers that incorporate random values called tweaks.

Non-deterministic encryption is appropriate when:

* Preventing correlation of the same IP address across records is critical
* Storage can accommodate the additional tweak data (8-16 bytes)
* Stronger privacy guarantees than deterministic encryption provides are required
* Processing the same address multiple times without revealing repetition patterns

Implementation details are provided in Section 9.

7.1. Encryption Process

Encryption process for non-deterministic modes:

1. Generate a random tweak using a cryptographically secure random number generator
2. Convert the IP address to its 16-byte representation
3. Encrypt the 16-byte representation using the key and the tweak
4. Concatenate the tweak with the encrypted output to form the final ciphertext

The tweak is not considered secret and is included in the ciphertext, enabling its use for decryption.

7.2. Decryption Process

Decryption process:

1. Split the ciphertext into the tweak and the encrypted IP
2. Decrypt the encrypted IP using the key and the tweak
3. Convert the resulting 16-byte representation back to an IP address

Although the tweak is generated uniformly at random, occasional collisions may occur according to birthday bounds. An (input, tweak) collision reveals that the same input was encrypted with the same tweak but does not disclose the input' s value. The usage limits apply per cryptographic key; rotating keys extends secure usage beyond these bounds.

7.3. Output Format and Encoding

The output of non-deterministic encryption is binary data. For text representation, the binary output MUST be encoded (e.g., hexadecimal or Base64). Implementations SHOULD document their chosen encoding method.

7.4. Concrete Instantiations

This document defines two concrete instantiations:

* ipcrypt-nd: Uses the KIASU-BC tweakable block cipher with an 8-byte (64-bit) tweak. See [KIASU-BC] for details.

* ipcrypt-ndx: Uses the AES-XTS tweakable block cipher with a 16-byte (128-bit) tweak. See [XTS-AES] for background.

In both cases, if a tweak is generated randomly, it MUST be uniformly random. Reusing the same randomly generated tweak on different inputs is acceptable from a confidentiality standpoint.

Test vectors are provided in Appendix A.3 and Appendix A.4.

7.4.1. ipcrypt-nd (KIASU-BC)

The ipcrypt-nd instantiation uses the KIASU-BC tweakable block cipher with an 8-byte (64-bit) tweak. Implementation details are provided in Section 9.9. The output is 24 bytes total, consisting of an 8-byte tweak concatenated with a 16-byte ciphertext.

Random sampling of an 8-byte tweak yields an expected collision after about 2^32 operations (approximately 4 billion). An (input, tweak) collision indicates repetition without revealing the input's value.

These collision bounds apply per cryptographic key. Regular key rotation can extend secure usage beyond these bounds. The effective security is determined by the underlying block cipher's strength.

Test vectors are provided in Appendix A.3.

7.4.2. ipcrypt-ndx (AES-XTS)

The ipcrypt-ndx instantiation uses the AES-XTS tweakable block cipher with a 16-byte (128-bit) tweak. The output is 32 bytes total, consisting of a 16-byte tweak concatenated with a 16-byte ciphertext.

Since only a single block is encrypted, the construction is equivalent to AES-XEX, and identical to AES-XTS at block index 0, where the tweak is not multiplied by the primitive element α.

For single-block AES-XTS, independent sampling of a 16-byte tweak results in an expected collision after about 2^64 operations (approximately 18 quintillion).

Similar to ipcrypt-nd, collisions reveal repetition without compromising the input value. These limits are per key, and regular key rotation extends secure usage. The effective security is governed by AES-128 strength (approximately 2^128 operations).

7.4.3. Comparison of Modes

Mode selection depends on specific privacy requirements and operational constraints:

* Deterministic (ipcrypt-deterministic):
  - Output size: 16 bytes (most compact)
  - Privacy: Same IP always produces same ciphertext (allows correlation)
  - Use case: When duplicate identification is needed or when format preservation is critical
  - Performance: Fastest (single AES operation)
* Prefix-Preserving (ipcrypt-pfx):
  - Output size: 4 bytes for IPv4, 16 bytes for IPv6 (maintains native sizes)
  - Privacy: Preserves network prefix relationships while encrypting actual network identities
  - Use case: Network analytics, traffic pattern analysis, subnet monitoring, DDoS mitigation
  - Performance: Bit-by-bit processing (64 AES operations for IPv4, 256 for IPv6), fully parallelizable
* Non-Deterministic ipcrypt-nd (KIASU-BC):
  - Output size: 24 bytes (16-byte ciphertext + 8-byte tweak)

- Privacy: Same IP produces different ciphertexts (prevents most correlation)
- Use case: General privacy protection with reasonable storage overhead
- Collision resistance: Approximately 4 billion operations per key

* Non-Deterministic ipcrypt-ndx (AES-XTS):
  - Output size: 32 bytes (16-byte ciphertext + 16-byte tweak)
  - Privacy: Same IP produces different ciphertexts (prevents correlation)
  - Use case: Maximum privacy protection when storage permits
  - Collision resistance: Approximately 18 quintillion operations per key

7.5. Alternatives to Random Tweaks

While this specification recommends uniformly random tweaks for non-deterministic encryption, alternative approaches may be considered:

* Monotonic Counter: A counter could be used as a tweak, but this is difficult to maintain in distributed systems. If the counter is not encrypted and the tweakable block cipher is not secure against related-tweak attacks, this could enable correlation attacks.
* UUIDs: UUIDs (such as UUIDv6 or UUIDv7) could be used as tweaks; however, these would reveal the original timestamp of the logged IP addresses, which may not be desirable from a privacy perspective.

Although the birthday bound presents considerations with random tweaks, random tweaks remain the recommended approach for practical deployments.

8. Security Considerations

The methods specified in this document provide strong confidentiality guarantees but explicitly do not provide integrity protection:

These methods provide protection against:

* Unauthorized parties learning the original IP addresses (without the key)

* Statistical analysis revealing patterns in network traffic (non-deterministic modes)

* Brute-force attacks on the address space (128-bit security level)

These methods do not provide protection against:

* Active attackers modifying, reordering, or removing encrypted addresses

* Authorized key holders decrypting addresses (by design)

* Traffic analysis based on volume and timing (metadata)

Applications requiring integrity protection must additionally employ authentication mechanisms such as HMAC, authenticated encryption modes, or digital signatures over the encrypted data.

8.1. Deterministic Mode Security

A permutation ensures distinct inputs yield distinct outputs. Repeated inputs result in identical ciphertexts, revealing repetition.

This makes deterministic encryption suitable for applications where format preservation is required and linkability is acceptable.

8.2. Non-Deterministic Mode Security

The inclusion of a random tweak ensures that encrypting the same input generally produces different outputs. An (input, tweak) collision reveals only that the same input was processed with that tweak, not the input's value.

Security is determined by the underlying block cipher (2^128 for AES-128) on a per-key basis. Key rotation is recommended to extend secure usage beyond the per-key collision bounds.

8.3. Implementation Security

Implementations MUST ensure that:

1. Keys are generated using a cryptographically secure random number generator

2. Tweak values are uniformly random for non-deterministic modes

3. Side-channel attacks are mitigated through constant-time operations

4. Error handling does not leak sensitive information

8.4. Key Derivation for Multiple Variants

When using multiple encryption variants within the same deployment, implementations MUST derive separate keys for each variant to prevent cross-mode correlations. The RECOMMENDED approach uses HKDF ([RFC5869]) to derive per-variant subkeys from a single master key:

* K_deterministic = HKDF-Expand(PRK, "ipcrypt-deterministic", 16)

* K_pfx = HKDF-Expand(PRK, "ipcrypt-pfx", 32)

* K_nd = HKDF-Expand(PRK, "ipcrypt-nd", 16)

* K_ndx = HKDF-Expand(PRK, "ipcrypt-ndx", 32)

Where:

* PRK = HKDF-Extract(salt, K_master) is a pseudorandom key derived from the master key

* K_master is a uniformly random master key

* salt is either empty or a fixed random value for the application

* The strings "ipcrypt-deterministic", etc. are used as the info parameter for domain separation

* The third parameter specifies the output length in bytes (16 for single AES keys, 32 for ipcrypt-pfx and ipcrypt-ndx)

This ensures that:

1. Using the same master key across different variants does not enable cross-variant attacks

2. Key management is simplified by requiring only a single master key

3. Each variant operates with cryptographically independent keys

8.5. Key Management Considerations

Implementers MUST ensure:

1. Keys are generated using cryptographically secure random number generators (see [RFC4086])

2. Keys are stored securely and access-controlled appropriately for the deployment environment

3. Key rotation policies are established based on usage volume and security requirements

4. Key compromise procedures are defined and tested

9. Implementation Details

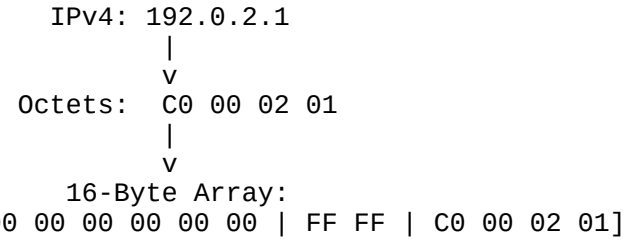
This section provides pseudocode and implementation guidance for the operations described in this document.

In the pseudocode throughout this document, the notation “for i from x to y” indicates iteration starting at x (inclusive) and ending before y (exclusive). For example, “for i from 0 to 4” processes values 0, 1, 2, and 3, but not 4.

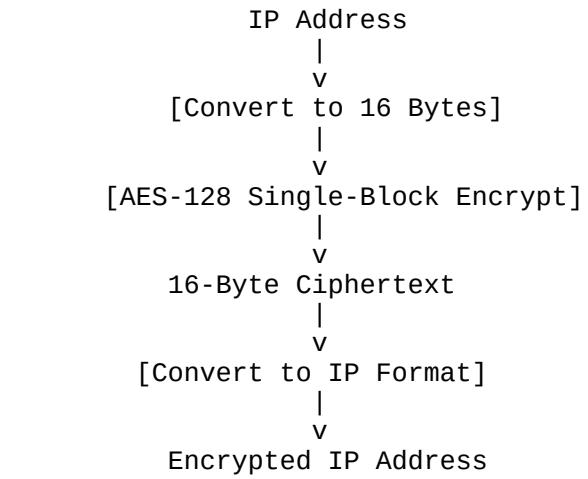
9.1. Visual Diagrams

The following diagrams illustrate the processes described in this specification.

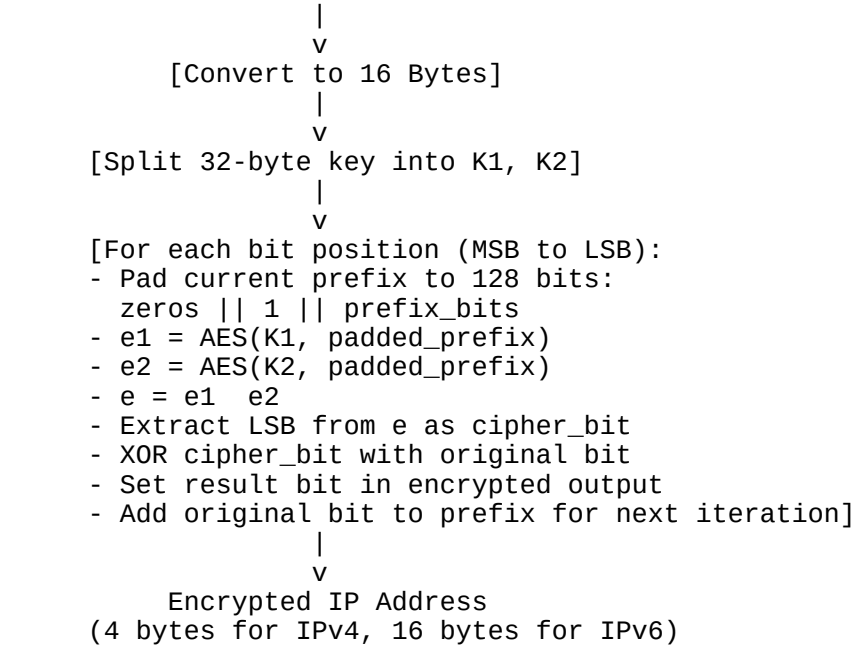
9.1.1. IPv4 Address Conversion Diagram



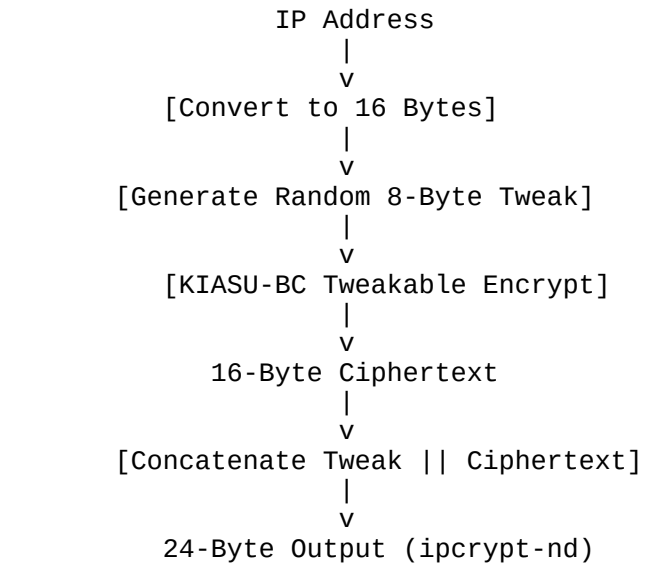
9.1.2. Deterministic Encryption Flow



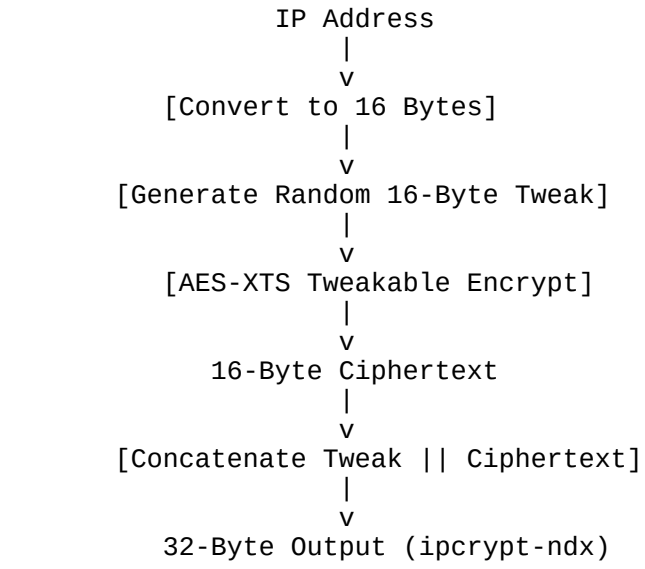
9.1.3. Prefix-Preserving Encryption Flow (ipcrypt-pfx)



9.1.4. Non-Deterministic Encryption Flow (ipcrypt-nd)



9.1.5. Non-Deterministic Encryption Flow (ipcrypt-ndx)



9.2. IPv4 Address Conversion

See Section 9.1.1 for a diagram of this process.

```
function ipv4_to_16_bytes(ipv4_address):
    // Parse the IPv4 address into 4 octets
    octets = parseIPv4(ipv4_address)  // Returns 4 octets
    // Create a 16-byte array with the IPv4-mapped prefix
    bytes16 = [0x00] * 10         // 10 bytes of 0x00
    bytes16.append(0xFF)          // 11th byte: 0xFF
    bytes16.append(0xFF)          // 12th byte: 0xFF
    // Append each octet (converted to an 8-bit integer)
    for octet in octets:
         bytes16.append(octet)
    return bytes16
```

Example: For "192.0.2.1", the function returns

[00, 00, 00, 00, 00, 00, 00, 00, 00, 00, FF, FF, C0, 00, 02, 01]

9.3. IPv6 Address Conversion

```
function ipv6_to_16_bytes(ipv6_address):
    // Parse the IPv6 address into eight 16-bit words.
    words = parseIPv6(ipv6_address)  // Expands shorthand notation and returns 8 words
    bytes16 = []
    for word in words:
         high_byte = (word >> 8) & 0xFF
         low_byte = word & 0xFF
         bytes16.append(high_byte)
         bytes16.append(low_byte)
    return bytes16
```

Example: For "2001:0db8:85a3:0000:0000:8a2e:0370:7334", the output is the corresponding 16-byte sequence {0x20, 0x01, 0x0d, 0xb8, ..., 0x34}.

9.4. Conversion from a 16-Byte Array to an IP Address String

This function converts a 16-byte array back to an IP address string. For IPv6 addresses, the output conforms to [RFC5952]. Implementers SHOULD use existing IP address formatting functions from standard libraries.

```
function bytes_16_to_ip(bytes16):
    if length(bytes16) != 16:
         raise Error("Invalid byte array")

    // Check for the IPv4-mapped prefix
    // When an IPv4-mapped IPv6 address (::ffff:x.x.x.x) is detected,
    // it is converted back to IPv4 format. This is expected
    // behavior as IPv4 addresses are internally represented as IPv4-mapped
```

```
       // IPv6 addresses for uniform processing.
       ipv4_mapped_prefix = [0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0xFF, 0xFF]
       if bytes16[0..12] == ipv4_mapped_prefix:
            // Convert the 4 last bytes to an IPv4 address
            ipv4_parts = []
            for i from 12 to 16:
                ipv4_parts.append(integer_to_string(bytes16[i]))
            ipv4_address = join(ipv4_parts, ".")
            return ipv4_address
       else:
            // Convert the 16 bytes to an IPv6 address with canonical representation
            words = []
            for i from 0 to 8:
                word = (bytes16[i*2] << 8) | bytes16[i*2+1]
                words.append(word)

            // Find longest run of consecutive zeros for compression
            best_run_start = -1
            best_run_length = 0
            run_start = -1

            for i from 0 to 8:
                if words[i] == 0:
                    if run_start == -1:
                        run_start = i
                else:
                    if run_start != -1:
                        run_length = i - run_start
                        if run_length > best_run_length:
                            best_run_start = run_start
                            best_run_length = run_length
                        run_start = -1

            // Check final run
            if run_start != -1:
                run_length = 8 - run_start
                if run_length > best_run_length:
                    best_run_start = run_start
                    best_run_length = run_length

            // Build IPv6 string with zero compression
            parts = []
            i = 0
            while i < 8:
                if best_run_length >= 2 and i == best_run_start:
                    // Insert :: for compressed zeros
                    parts.append("" if i == 0 else ":")
                    parts.append("")
```

```
                    i += best_run_length
                else:
                    parts.append(format_hex(words[i]))
                    i += 1

            return join(parts, ":")
```

9.5. Deterministic Encryption (ipcrypt-deterministic)

9.5.1. Encryption

```
   function ipcrypt_deterministic_encrypt(ip_address, key):
       // The key MUST be exactly 16 bytes (128 bits) in length
       if length(key) != 16:
           raise Error("Key must be 16 bytes")

       bytes16 = convert_to_16_bytes(ip_address)
       ciphertext = AES128_encrypt(key, bytes16)
       encrypted_ip = bytes_16_to_ip(ciphertext)
       return encrypted_ip
```

9.5.2. Decryption

```
   function ipcrypt_deterministic_decrypt(encrypted_ip, key):
       if length(key) != 16:
           raise Error("Key must be 16 bytes")

       bytes16 = convert_to_16_bytes(encrypted_ip)
       plaintext = AES128_decrypt(key, bytes16)
       original_ip = bytes_16_to_ip(plaintext)
       return original_ip
```

9.6. Prefix-Preserving Encryption (ipcrypt-pfx)

9.6.1. Encryption

```
   function ipcrypt_pfx_encrypt(ip_address, key):
       // The key MUST be exactly 32 bytes (256 bits)
       if length(key) != 32:
           raise Error("Key must be 32 bytes")

       // Convert IP to 16-byte representation
       bytes16 = convert_to_16_bytes(ip_address)

       // Split the key into two AES-128 keys
       // IMPORTANT: K1 and K2 MUST be different
       K1 = key[0:16]
       K2 = key[16:32]
```

```
       // Initialize encrypted result with zeros
       encrypted = [0] * 16

       // If we encrypt an IPv4 address, start where the IPv4 address starts (bit 96)
       // Note the first 12 bytes of bytes16 are already set to the prefix for IPv4 mappi
ng in that case
       // This provides domain separation between an IPv4 address and the first 32 bits o
f an IPv6 address
       if is_ipv4_mapped(bytes16):
           prefix_start = 96
           // Set up the IPv4-mapped IPv6 prefix
           encrypted[10] = 0xFF
           encrypted[11] = 0xFF
       else:
           prefix_start = 0

       // Initialize padded_prefix for the starting prefix length
       padded_prefix = pad_prefix(bytes16, prefix_start)

       // Process each bit position sequentially
       // Note: prefix_len_bits represents how many bits from the MSB have been processed
       // Range is [prefix_start, 128), i.e., up to but not including 128
       for prefix_len_bits from prefix_start to 128:
           // Compute pseudorandom function with dual AES encryption
           e1 = AES128_encrypt(K1, padded_prefix)
           e2 = AES128_encrypt(K2, padded_prefix)
           e = e1  e2
           // Output of the pseudorandom function is the least significant bit of e
           cipher_bit = get_bit(e, 0)

           // Encrypt the current bit position (processing from MSB to LSB)
           // For IPv6: prefix_len_bits=0 encrypts bit 127, prefix_len_bits=1 encrypts bi
t 126, etc.
           // For IPv4: prefix_len_bits=96 encrypts bit 31, prefix_len_bits=97 encrypts b
it 30, etc.
           bit_pos = 127 - prefix_len_bits
           original_bit = get_bit(bytes16, bit_pos)
           set_bit(encrypted, bit_pos, cipher_bit ^ original_bit)

           // Prepare padded_prefix for next iteration
           // Shift left by 1 bit and insert the next bit from bytes16
           padded_prefix = shift_left_one_bit(padded_prefix)
           set_bit(padded_prefix, 0, original_bit)

       // Convert back to IP format
       return bytes_16_to_ip(encrypted)
```

9.6.2. Decryption

```
   function ipcrypt_pfx_decrypt(encrypted_ip, key):
       // The key MUST be exactly 32 bytes (256 bits)
       if length(key) != 32:
           raise Error("Key must be 32 bytes")

       // Convert encrypted IP to 16-byte representation
       encrypted_bytes = convert_to_16_bytes(encrypted_ip)

       // Split the key into two AES-128 keys
       K1 = key[0:16]
       K2 = key[16:32]

       // Initialize decrypted result with zeros
       decrypted = [0] * 16

       // If we decrypt an IPv4 address, start where the IPv4 address starts (bit 96)
       if is_ipv4_mapped(encrypted_bytes):
           prefix_start = 96
           // Set up the IPv4-mapped IPv6 prefix
           decrypted[10] = 0xFF
           decrypted[11] = 0xFF
       else:
           prefix_start = 0

       // Initialize padded_prefix for the starting prefix length
       padded_prefix = pad_prefix(decrypted, prefix_start)

       // Process each bit position sequentially
       // Note: prefix_len_bits represents how many bits from the MSB have been processed
       // Range is [prefix_start, 128), i.e., up to but not including 128
       for prefix_len_bits from prefix_start to 128:
           // Compute pseudorandom function with dual AES encryption
           e1 = AES128_encrypt(K1, padded_prefix)
           e2 = AES128_encrypt(K2, padded_prefix)
           // e is expected to be the same as during encryption since the prefix is the s
ame
           e = e1  e2
           // Output of the pseudorandom function is the least significant bit of e
           cipher_bit = get_bit(e, 0)

           // Decrypt the current bit position (processing from MSB to LSB)
           // For IPv6: prefix_len_bits=0 decrypts bit 127, prefix_len_bits=1 decrypts bi
t 126, etc.
           // For IPv4: prefix_len_bits=96 decrypts bit 31, prefix_len_bits=97 decrypts b
it 30, etc.
           bit_pos = 127 - prefix_len_bits
           encrypted_bit = get_bit(encrypted_bytes, bit_pos)
           original_bit = cipher_bit ^ encrypted_bit
           set_bit(decrypted, bit_pos, original_bit)

           // Prepare padded_prefix for next iteration
```

```
            // Shift left by 1 bit and insert the next bit from decrypted
            padded_prefix = shift_left_one_bit(padded_prefix)
            set_bit(padded_prefix, 0, original_bit)

        // Convert back to IP format
        return bytes_16_to_ip(decrypted)
```

9.6.3. Helper Functions

The following helper functions are used in the ipcrypt-pfx implementation:

```
    function is_ipv4_mapped(bytes16):
        // Returns true if the 16-byte array has the IPv4-mapped IPv6 prefix (::ffff:0:0/9
6)
        ipv4_mapped_prefix = [0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0xFF, 0xFF]
        return bytes16[0..12] == ipv4_mapped_prefix

    function get_bit(data, position):
        // Extract bit at position from 16-byte array representing an IPv6 address in netw
ork byte order
        // position: 0 = LSB of byte 15, 127 = MSB of byte 0
        // Example: position 127 refers to bit 7 (MSB) of data[0]
        // Example: position 0 refers to bit 0 (LSB) of data[15]
        byte_index = 15 - (position / 8)
        bit_index = position % 8
        return (data[byte_index] >> bit_index) & 1

    function set_bit(data, position, value):
        // Set bit at position in 16-byte array representing an IPv6 address in network by
te order
        // position: 0 = LSB of byte 15, 127 = MSB of byte 0
        byte_index = 15 - (position / 8)
        bit_index = position % 8
        data[byte_index] |= ((value & 1) << bit_index)

    function pad_prefix(data, prefix_len_bits):
        // Specialized for the only two cases used: 0 and 96
        // For prefix_len_bits=0: Returns a block with only bit 0 set (position 0 = LSB of
 byte 15)
        // For prefix_len_bits=96: Returns the IPv4-mapped prefix with separator at positi
on 96

        if prefix_len_bits == 0:
            // For IPv6 addresses starting from bit 0
            padded_prefix = [0] * 16
            padded_prefix[15] = 0x01  // Set bit at position 0 (LSB of byte 15)
            return padded_prefix

        else if prefix_len_bits == 96:
            // For IPv4 addresses, always returns the same value since all IPv4 addresses
            // share the same IPv4-mapped prefix (00...00 ffff)
            padded_prefix = [0] * 16
```

```
            padded_prefix[3] = 0x01   // Set separator bit at position 96 (bit 0 of byte 3
)
            padded_prefix[14] = 0xFF  // IPv4-mapped prefix
            padded_prefix[15] = 0xFF  // IPv4-mapped prefix
            return padded_prefix

        else:
            raise Error("pad_prefix only supports prefix_len_bits of 0 or 96")

    function shift_left_one_bit(data):
        // Shift a 16-byte array one bit to the left
        // The most significant bit is lost, and a zero bit is shifted in from the right
        result = [0] * 16
        carry = 0

        // Process from least significant byte (byte 15) to most significant byte (byte 0)
        for i from 15 down to 0:
            // Current byte shifted left by 1, with carry from previous byte
            result[i] = ((data[i] << 1) | carry) & 0xFF
            // Extract the bit that will be carried to the next byte
            carry = (data[i] >> 7) & 1

        return result
```

9.7. Non-Deterministic Encryption using KIASU-BC (ipcrypt-nd)

9.7.1. Encryption

```
    function ipcrypt_nd_encrypt(ip_address, key):
        if length(key) != 16:
            raise Error("Key must be 16 bytes")

        // Step 1: Generate random tweak (8 bytes)
        tweak = random_bytes(8)  // MUST be uniformly random

        // Step 2: Convert IP to 16-byte representation
        bytes16 = convert_to_16_bytes(ip_address)

        // Step 3: Encrypt using key and tweak
        ciphertext = KIASU_BC_encrypt(key, tweak, bytes16)

        // Step 4: Concatenate tweak and ciphertext
        result = concatenate(tweak, ciphertext)  // 8 bytes || 16 bytes = 24 bytes total
        return result
```

9.7.2. Decryption

```
   function ipcrypt_nd_decrypt(ciphertext, key):
       // Step 1: Split ciphertext into tweak and encrypted IP
       tweak = ciphertext[0:8]  // First 8 bytes
       encrypted_ip = ciphertext[8:24]  // Remaining 16 bytes

       // Step 2: Decrypt using key and tweak
       bytes16 = KIASU_BC_decrypt(key, tweak, encrypted_ip)

       // Step 3: Convert back to IP address
       ip_address = bytes_16_to_ip(bytes16)
       return ip_address
```

9.8. Non-Deterministic Encryption using AES-XTS (ipcrypt-ndx)

9.8.1. Encryption

```
   function ipcrypt_ndx_encrypt(ip_address, key):
       if length(key) != 32:
           raise Error("Key must be 32 bytes (two AES-128 keys)")

       // Step 1: Generate random tweak (16 bytes)
       tweak = random_bytes(16)  // MUST be uniformly random

       // Step 2: Convert IP to 16-byte representation
       bytes16 = convert_to_16_bytes(ip_address)

       // Step 3: Encrypt using key and tweak
       ciphertext = AES_XTS_encrypt(key, tweak, bytes16)

       // Step 4: Concatenate tweak and ciphertext
       result = concatenate(tweak, ciphertext)  // 16 bytes || 16 bytes = 32 bytes total
       return result
```

9.8.2. Decryption

```
   function ipcrypt_ndx_decrypt(ciphertext, key):
       // Step 1: Split ciphertext into tweak and encrypted IP
       tweak = ciphertext[0:16]  // First 16 bytes
       encrypted_ip = ciphertext[16:32]  // Remaining 16 bytes

       // Step 2: Decrypt using key and tweak
       bytes16 = AES_XTS_decrypt(key, tweak, encrypted_ip)

       // Step 3: Convert back to IP address
       ip_address = bytes_16_to_ip(bytes16)
       return ip_address
```

9.8.3. Helper Functions for AES-XTS

```
function AES_XTS_encrypt(key, tweak, block):
    // Split the key into two halves
    K1, K2 = split_key(key)

    // Encrypt the tweak with the second half of the key
    ET = AES128_encrypt(K2, tweak)

    // Encrypt the block: AES128_encrypt(K1, block  ET)  ET
    return AES128_encrypt(K1, block  ET)  ET

function AES_XTS_decrypt(key, tweak, block):
    // Split the key into two halves
    K1, K2 = split_key(key)

    // Encrypt the tweak with the second half of the key
    ET = AES128_encrypt(K2, tweak)

    // Decrypt the block: AES128_decrypt(K1, block  ET)  ET
    return AES128_decrypt(K1, block  ET)  ET
```

9.9. KIASU-BC Implementation Guide

This section provides a guide for implementing the KIASU-BC tweakable block cipher used in ipcrypt-nd.

9.9.1. Overview

KIASU-BC extends AES-128 by incorporating an 8-byte tweak into each round. The tweak is padded to 16 bytes and XORed with the round key at each round.

9.9.2. Tweak Padding

The 8-byte tweak is padded to 16 bytes using the following method:

1. Split the 8-byte tweak into four 2-byte pairs

2. Place each 2-byte pair at the start of each 4-byte group

3. Fill the remaining 2 bytes of each group with zeros

Example:

```
8-byte tweak:    [T0 T1 T2 T3 T4 T5 T6 T7]
16-byte padded:  [T0 T1 00 00 T2 T3 00 00 T4 T5 00 00 T6 T7 00 00]
```

9.9.3. Round Structure

Each round of KIASU-BC consists of the following standard AES operations:

1. SubBytes: Apply the AES S-box to each byte of the state
2. ShiftRows: Rotate each row of the state matrix
3. MixColumns: Mix the columns of the state matrix (except in the final round)
4. AddRoundKey: XOR the state with the round key and padded tweak

Details about these operations are provided in [FIPS-197].

9.9.4. Key Schedule

The key schedule follows the standard AES-128 key expansion:

1. The initial key is expanded into 11 round keys
2. Each round key is XORed with the padded tweak before use
3. The first round key is used in the initial AddRoundKey operation

9.9.5. Implementation Steps

1. Key Expansion:
    * Expand the 16-byte key into 11 round keys using the standard AES key schedule
    * Each round key is 16 bytes
2. Tweak Processing:
    * Pad the 8-byte tweak to 16 bytes as described above
    * XOR the padded tweak with each round key before use
3. Encryption Process:
    * Perform initial AddRoundKey with the first tweaked round key
    * For rounds 1-9:
        - SubBytes

- ShiftRows
- MixColumns
- AddRoundKey (with tweaked round key)

* For round 10 (final round):
  - SubBytes
  - ShiftRows
  - AddRoundKey (with tweaked round key)

9.9.6. Example Implementation

The following pseudocode illustrates the core operations of KIASU-BC:

```
function pad_tweak(tweak):
    // Input: 8-byte tweak
    // Output: 16-byte padded tweak
    padded = [0] * 16
    for i in range(0, 4):
        padded[i*4]   = tweak[i*2]
        padded[i*4+1] = tweak[i*2+1]
    return padded

function kiasu_bc_encrypt(key, tweak, plaintext):
    // Input: 16-byte key, 8-byte tweak, 16-byte plaintext
    // Output: 16-byte ciphertext

    // Expand key and pad tweak
    round_keys = expand_key(key)
    padded_tweak = pad_tweak(tweak)

    // Initial round
    state = plaintext
    state = add_round_key(state, round_keys[0] ^ padded_tweak)

    // Main rounds
    for round in range(1, 10):
        state = sub_bytes(state)
        state = shift_rows(state)
        state = mix_columns(state)
        state = add_round_key(state, round_keys[round] ^ padded_tweak)

    // Final round
    state = sub_bytes(state)
    state = shift_rows(state)
    state = add_round_key(state, round_keys[10] ^ padded_tweak)

    return state
```

Key and tweak sizes for each variant:

* ipcrypt-deterministic: Key: 16 bytes (128 bits), no tweak, Output: 16 bytes

* ipcrypt-pfx: Key: 32 bytes (256 bits, split into two independent AES-128 keys), no external tweak (uses prefix as cryptographic context), Output: 4 bytes for IPv4, 16 bytes for IPv6

* ipcrypt-nd: Key: 16 bytes (128 bits), Tweak: 8 bytes (64 bits), Output: 24 bytes

* ipcrypt-ndx: Key: 32 bytes (256 bits, split into two AES-128 keys), Tweak: 16 bytes (128 bits), Output: 32 bytes

10. Implementation Status

This section is to be removed before publishing as an RFC.

This section records the status of known implementations of the protocol defined by this specification at the time of posting of this Internet-Draft, and is based on a proposal described in [RFC7942].

Multiple independent, interoperable implementations have been developed and include Awk, C, D, Dart, Elixir, Go, Java, JavaScript, Kotlin, Lua, PHP, Python, Ruby, Rust, Swift, and Zig.

A comprehensive list of implementations and their test results is available at: https://ipcrypt-std.github.io/implementations/

All implementations pass the common test vectors specified in this document, demonstrating interoperability across programming languages.

11. Licensing

This section is to be removed before publishing as an RFC.

Implementations of the ipcrypt methods are freely available under permissive open source licenses (MIT, BSD, or Apache 2.0) at the repository listed in the Implementation Status section.

There are no known patent claims on these methods.

12. References

12.1. Normative References

[FIPS-197] NIST, "Advanced Encryption Standard (AES)", FIPS PUB 197, 26 November 2001, <https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.197.pdf>.

[NIST-SP-800-38G] NIST, "Recommendation for Block Cipher Modes of Operation: Methods for Format-Preserving Encryption", NIST SP 800-38G, March 2016, <https://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-38G.pdf>.

[RFC4086] Eastlake 3rd, D., Schiller, J., and S. Crocker, "Randomness Requirements for Security", BCP 106, RFC 4086, DOI 10.17487/RFC4086, June 2005, <https://www.rfc-editor.org/rfc/rfc4086>.

[RFC4291] Hinden, R. and S. Deering, "IP Version 6 Addressing Architecture", RFC 4291, DOI 10.17487/RFC4291, February 2006, <https://www.rfc-editor.org/rfc/rfc4291>.

[RFC5869] Krawczyk, H. and P. Eronen, "HMAC-based Extract-and-Expand Key Derivation Function (HKDF)", RFC 5869, DOI 10.17487/RFC5869, May 2010, <https://www.rfc-editor.org/rfc/rfc5869>.

[RFC5952] Kawamura, S. and M. Kawashima, "A Recommendation for IPv6 Address Text Representation", RFC 5952, DOI 10.17487/RFC5952, August 2010, <https://www.rfc-editor.org/rfc/rfc5952>.

[RFC6973] Cooper, A., Tschofenig, H., Aboba, B., Peterson, J., Morris, J., Hansen, M., and R. Smith, "Privacy Considerations for Internet Protocols", RFC 6973, DOI 10.17487/RFC6973, July 2013, <https://www.rfc-editor.org/rfc/rfc6973>.

[RFC7258] Farrell, S. and H. Tschofenig, "Pervasive Monitoring Is an Attack", BCP 188, RFC 7258, DOI 10.17487/RFC7258, May 2014, <https://www.rfc-editor.org/rfc/rfc7258>.

[RFC7624] Barnes, R., Schneier, B., Jennings, C., Hardie, T., Trammell, B., Huitema, C., and D. Borkmann, "Confidentiality in the Face of Pervasive Surveillance: A Threat Model and Problem Statement", RFC 7624, DOI 10.17487/RFC7624, August 2015, <https://www.rfc-editor.org/rfc/rfc7624>.

[RFC7942] Sheffer, Y. and A. Farrel, "Improving Awareness of Running Code: The Implementation Status Section", BCP 205, RFC 7942, DOI 10.17487/RFC7942, July 2016, <https://www.rfc-editor.org/rfc/rfc7942>.

[RFC8174] Leiba, B., "Ambiguity of Uppercase vs Lowercase in RFC 2119 Key Words", BCP 14, RFC 8174, DOI 10.17487/RFC8174, May 2017, <https://www.rfc-editor.org/rfc/rfc8174>.

12.2. Informative References

[BRW2005] Black, J. and P. Rogaway, "Format-Preserving Encryption", CT-RSA 2002, LNCS 2271, pp. 114130 , 8 February 2002, <https://www.cs.ucdavis.edu/~rogaway/papers/subset.pdf>.

[DEOXYS-TBC] Jean, J., Nikoli, I., Peyrin, T., and Y. Seurin, "The Deoxys AEAD Family", Journal of Cryptology 34, 31 (2021) , 10 June 2021, <https://thomaspeyrin.github.io/web/assets/docs/papers/Jean-etal-JoC2021.pdf>.

[FAST] Betul Durak, F., Horst, H., and S. Vaudenay, "FAST: Format-Preserving Encryption via Shortened AES Tweakable Block Cipher", Cryptology ePrint Archive Paper 2021/1171, 14 September 2021, <https://eprint.iacr.org/2021/1171.pdf>.

[IEEE-P1619] IEEE, "IEEE Standard for Cryptographic Protection of Data on Block-Oriented Storage Devices", IEEE 1619-2007, 4 March 2008, <https://ieeexplore.ieee.org/document/4493450>.

[IPCRYPT] Denis, F., "IPCrypt: Optimal, Practical Encryption of IP Addresses for Privacy and Measurement", Cryptology ePrint Archive Paper 2025/1689, 9 January 2025, <https://eprint.iacr.org/2025/1689>.

[KIASU-BC] Jean, J., Nikoli, I., and T. Peyrin, "Tweaks and Keys for Block Ciphers: the TWEAKEY Framework", ASIACRYPT 2014, LNCS 8874, pp. 274288 , December 2014, <https://eprint.iacr.org/2014/831.pdf>.

[LRW2002] Liskov, M., Rivest, R., and D. Wagner, "Tweakable Block Ciphers", CRYPTO 2002, LNCS 2442, pp. 3146 , 18 August 2002, <https://people.csail.mit.edu/rivest/pubs/LRW02.pdf>.

[PATARIN-2008] Patarin, J., "A Proof of Security in O(2^n) for the Xor of Two Random Permutations", ICITS 2008, LNCS 5155, pp. 232248 , 2008, <https://link.springer.com/chapter/10.1007/978-3-540-85093-9_22>.

[PATARIN-H-COEFFICIENTS] Patarin, J., "The 'Coefficients H' Technique", SAC 2008, LNCS 5381, pp. 328345 , 2009, <https://link.springer.com/chapter/10.1007/978-3-642-04159-4_21>.

[RSSAC040] ICANN RSSAC, "RSSAC040: Recommendations on Anonymization Processes for Source IP Addresses Submitted for Future Analysis", ICANN RSSAC RSSAC040, 7 August 2018, <https://www.icann.org/en/system/files/files/rssac-040-07aug18-en.pdf>.

[SKINNY] Beierle, C., Jean, J., Koelbl, S., Leander, G., Moradi, A., Peyrin, T., Sasaki, Y., Sasdrich, P., and S. Meng Sim, "The SKINNY Family of Block Ciphers and its Low-Latency Variant MANTIS", CRYPTO 2016, LNCS 9815, pp. 123153 , 14 August 2016, <https://eprint.iacr.org/2016/660.pdf>.

[SUM-OF-PRPS] Lucks, S., "The Sum of PRPs Is a Secure PRF", EUROCRYPT 2000, LNCS 1807, pp. 470484 , 14 May 2000, <https://link.springer.com/content/pdf/10.1007/3-540-45539-6_34.pdf>.

[XTS-AES] "The XTS-AES Mode for Disk Encryption", IEEE 1619-2007, 4 March 2008, <https://ieeexplore.ieee.org/document/4493450>.

Appendix A. Test Vectors

This appendix provides test vectors for the ipcrypt variants. Each test vector includes the key, input IP address, and encrypted output. Non-deterministic variants include the tweak value.

Implementations MUST verify their correctness against these test vectors before deployment.

A.1. ipcrypt-deterministic Test Vectors

```
# Test vector 1
Key:          0123456789abcdeffedcba9876543210
Input IP:     0.0.0.0
Encrypted IP: bde9:6789:d353:824c:d7c6:f58a:6bd2:26eb

# Test vector 2
Key:          1032547698badcfeefcdab8967452301
Input IP:     255.255.255.255
Encrypted IP: aed2:92f6:ea23:58c3:48fd:8b8:74e8:45d8

# Test vector 3
Key:          2b7e151628aed2a6abf7158809cf4f3c
Input IP:     192.0.2.1
Encrypted IP: 1dbd:c1b9:fff1:7586:7d0b:67b4:e76e:4777
```

A.2. ipcrypt-pfx Test Vectors

The following test vectors demonstrate the prefix-preserving property of ipcrypt-pfx. Addresses from the same network produce encrypted addresses that share a common encrypted prefix.

A.2.1. Basic Test Vectors

```
# Test vector 1 (IPv4)
Key:          0123456789abcdeffedcba98765432101032547698badcfeefcdab8967452301
Input IP:     0.0.0.0
Encrypted IP: 151.82.155.134

# Test vector 2 (IPv4)
Key:          0123456789abcdeffedcba98765432101032547698badcfeefcdab8967452301
Input IP:     255.255.255.255
Encrypted IP: 94.185.169.89

# Test vector 3 (IPv4)
Key:          0123456789abcdeffedcba98765432101032547698badcfeefcdab8967452301
Input IP:     192.0.2.1
Encrypted IP: 100.115.72.131

# Test vector 4 (IPv6)
Key:          0123456789abcdeffedcba98765432101032547698badcfeefcdab8967452301
Input IP:     2001:db8::1
Encrypted IP: c180:5dd4:2587:3524:30ab:fa65:6ab6:f88
```

A.2.2. Prefix-Preserving Test Vectors

These test vectors demonstrate the prefix-preserving property. Addresses from the same network share common encrypted prefixes at the corresponding prefix length.

```
# IPv4 addresses from same /24 network (10.0.0.0/24)
Key:          2b7e151628aed2a6abf7158809cf4f3ca9f5ba40db214c3798f2e1c23456789a
Input IP:     10.0.0.47
Encrypted IP: 19.214.210.244

Key:          2b7e151628aed2a6abf7158809cf4f3ca9f5ba40db214c3798f2e1c23456789a
Input IP:     10.0.0.129
Encrypted IP: 19.214.210.80

Key:          2b7e151628aed2a6abf7158809cf4f3ca9f5ba40db214c3798f2e1c23456789a
Input IP:     10.0.0.234
Encrypted IP: 19.214.210.30

# IPv4 addresses from same /16 but different /24 networks
```

```
   Key:          2b7e151628aed2a6abf7158809cf4f3ca9f5ba40db214c3798f2e1c23456789a
   Input IP:     172.16.5.193
   Encrypted IP: 210.78.229.136

   Key:          2b7e151628aed2a6abf7158809cf4f3ca9f5ba40db214c3798f2e1c23456789a
   Input IP:     172.16.97.42
   Encrypted IP: 210.78.179.241

   Key:          2b7e151628aed2a6abf7158809cf4f3ca9f5ba40db214c3798f2e1c23456789a
   Input IP:     172.16.248.177
   Encrypted IP: 210.78.121.215

   # IPv6 addresses from same /64 network (2001:db8::/64)
   Key:          2b7e151628aed2a6abf7158809cf4f3ca9f5ba40db214c3798f2e1c23456789a
   Input IP:     2001:db8::a5c9:4e2f:bb91:5a7d
   Encrypted IP: 7cec:702c:1243:f70:1956:125:b9bd:1aba

   Key:          2b7e151628aed2a6abf7158809cf4f3ca9f5ba40db214c3798f2e1c23456789a
   Input IP:     2001:db8::7234:d8f1:3c6e:9a52
   Encrypted IP: 7cec:702c:1243:f70:a3ef:c8e:95c1:cd0d

   Key:          2b7e151628aed2a6abf7158809cf4f3ca9f5ba40db214c3798f2e1c23456789a
   Input IP:     2001:db8::f1e0:937b:26d4:8c1a
   Encrypted IP: 7cec:702c:1243:f70:443c:c8e:6a62:b64d

   # IPv6 addresses from same /32 but different /48 networks
   Key:          2b7e151628aed2a6abf7158809cf4f3ca9f5ba40db214c3798f2e1c23456789a
   Input IP:     2001:db8:3a5c:0:e7d1:4b9f:2c8a:f673
   Encrypted IP: 7cec:702c:3503:bef:e616:96bd:be33:a9b9

   Key:          2b7e151628aed2a6abf7158809cf4f3ca9f5ba40db214c3798f2e1c23456789a
   Input IP:     2001:db8:9f27:0:b4e2:7a3d:5f91:c8e6
   Encrypted IP: 7cec:702c:a504:b74e:194a:3d90:b047:2d1a

   Key:          2b7e151628aed2a6abf7158809cf4f3ca9f5ba40db214c3798f2e1c23456789a
   Input IP:     2001:db8:d8b4:0:193c:a5e7:8b2f:46d1
   Encrypted IP: 7cec:702c:f840:aa67:1b8:e84f:ac9d:77fb
```

A.3. ipcrypt-nd Test Vectors

```
# Test vector 1
Key:          0123456789abcdeffedcba9876543210
Input IP:     0.0.0.0
Tweak:        08e0c289bff23b7c
Output:       08e0c289bff23b7cb349aadfe3bcef56221c384c7c217b16

# Test vector 2
Key:          1032547698badcfeefcdab8967452301
Input IP:     192.0.2.1
Tweak:        21bd1834bc088cd2
Output:       21bd1834bc088cd2e5e1fe55f95876e639faae2594a0caad

# Test vector 3
Key:          2b7e151628aed2a6abf7158809cf4f3c
Input IP:     2001:db8::1
Tweak:        b4ecbe30b70898d7
Output:       b4ecbe30b70898d7553ac8974d1b4250eafc4b0aa1f80c96
```

A.4. ipcrypt-ndx Test Vectors

```
# Test vector 1
Key:          0123456789abcdeffedcba98765432101032547698badcfeefcdab8967452301
Input IP:     0.0.0.0
Tweak:        21bd1834bc088cd2b4ecbe30b70898d7
Output:       21bd1834bc088cd2b4ecbe30b70898d782db0d4125fdace61db35b8339f20ee5

# Test vector 2
Key:          1032547698badcfeefcdab89674523010123456789abcdeffedcba9876543210
Input IP:     192.0.2.1
Tweak:        08e0c289bff23b7cb4ecbe30b70898d7
Output:       08e0c289bff23b7cb4ecbe30b70898d7766a533392a69edf1ad0d3ce362ba98a

# Test vector 3
Key:          2b7e151628aed2a6abf7158809cf4f3c3c4fcf098815f7aba6d2ae2816157e2b
Input IP:     2001:db8::1
Tweak:        21bd1834bc088cd2b4ecbe30b70898d7
Output:       21bd1834bc088cd2b4ecbe30b70898d76089c7e05ae30c2d10ca149870a263e4
```

For non-deterministic variants, the tweak values shown are examples. Tweaks MUST be uniformly random for each encryption operation.

IANA Considerations

This document does not require any IANA actions.

Acknowledgments

We would like to thank the members of the IETF and the cryptographic community for their helpful comments on this work. Tobias Fiebig conducted a detailed review of this draft. Additional valuable feedback was provided by Eliot Lear. Assistance with implementation aspects was kindly provided by Wu Tingfeng.

Author’s Address

Frank Denis
Fastly Inc.
Email: fde@00f.net