Network Working Group
Internet-Draft
Intended status: Informational
Expires: 30 March 2026

F. Denis
Fastly Inc.

26 September 2025

Prefix-Preserving Encryption for URIs
draft-denis-uricrypt-02

Abstract

This document specifies URICrypt, a deterministic, prefix-preserving encryption scheme for Uniform Resource Identifiers (URIs). URICrypt encrypts URI paths while preserving their hierarchical structure, enabling systems that rely on URI prefix relationships to continue functioning with encrypted URIs. The scheme provides authenticated encryption for each URI path component, preventing tampering, reordering, or mixing of encrypted segments.

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-uricrypt.

Status of This Memo

This Internet-Draft is submitted in full conformance with the provisions of BCP 78 and BCP 79.

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF). Note that other groups may also distribute working documents as Internet-Drafts. The list of current Internet-Drafts is at https://datatracker.ietf.org/drafts/current/.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

This Internet-Draft will expire on 30 March 2026.

Table of Contents

1. Introduction

This document specifies URICrypt, a method for encrypting Uniform Resource Identifiers (URIs) while preserving their hierarchical structure. The primary motivation is to enable systems that rely on URI prefix relationships for routing, filtering, or access control to continue functioning with encrypted URIs.

URICrypt achieves prefix preservation through a chained encryption model where the encryption of each URI component depends cryptographically on all preceding components. This ensures that URIs sharing common prefixes produce ciphertexts that also share common encrypted prefixes.

The scheme uses an extendable-output function (XOF) as its cryptographic primitive and provides authenticated encryption for each component, preventing tampering, reordering, or mixing of encrypted segments. URICrypt is a reversible encryption scheme: encrypted URIs can be fully decrypted to recover the original URIs, but only with possession of the secret key.

1.1. Use Cases and Motivations

The main motivations include:

* Access Control in CDNs: Content Delivery Networks often use URI prefixes for routing and access control. URICrypt allows encryption of resource paths while preserving the prefix structure needed for CDN operations.

* Privacy-Preserving Logging: Systems can log encrypted URIs without exposing sensitive path information, while still enabling analysis based on URI structure.

* Confidential Data Sharing: When sharing links to sensitive resources, URICrypt prevents the path structure itself from revealing confidential information.

* Token-Based Access Systems: Systems that issue time-limited access tokens can use URICrypt to obfuscate the underlying resource location while maintaining routability.

* Multi-tenant Systems: In systems where multiple tenants share infrastructure, URICrypt can isolate tenant data while allowing shared components to be processed efficiently.

* Privacy-preserving Analytics: URICrypt can complement IPCrypt [I-D.draft-denis-ipcrypt]. Together, they enable systems to perform analytics on encrypted network flows and resource access patterns without exposing sensitive information about either the network endpoints or the specific resources being accessed.

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 [RFC2119] [RFC8174] when, and only when, they appear in all capitals, as shown here.

Throughout this document, the following terms and conventions apply:

* URI: Uniform Resource Identifier as defined in [RFC3986].

* URI Component: A segment of a URI path, terminated by '/', '?', or '#' characters. For encryption purposes, components include the trailing terminator except for the final component.

* Scheme: The URI scheme (e.g., "https://") which is preserved in plaintext.

* XOF: Extendable-Output Function, a cryptographic function that can produce output of arbitrary length.

* SIV: Synthetic Initialization Vector, a value derived from the accumulated state of all previous components, used for authentication and as input to keystream generation.

* SIVLEN: The length of the Synthetic Initialization Vector in bytes, defined as 16 bytes (128 bits) for this specification.

* PADBS: Padding Block Size, the number of bytes to which ciphertext components are aligned. Defined as 3 bytes for this specification to ensure efficient Base64url encoding without padding characters.

* Domain Separation: The practice of using distinct inputs to cryptographic functions to ensure outputs for different purposes are not compatible.

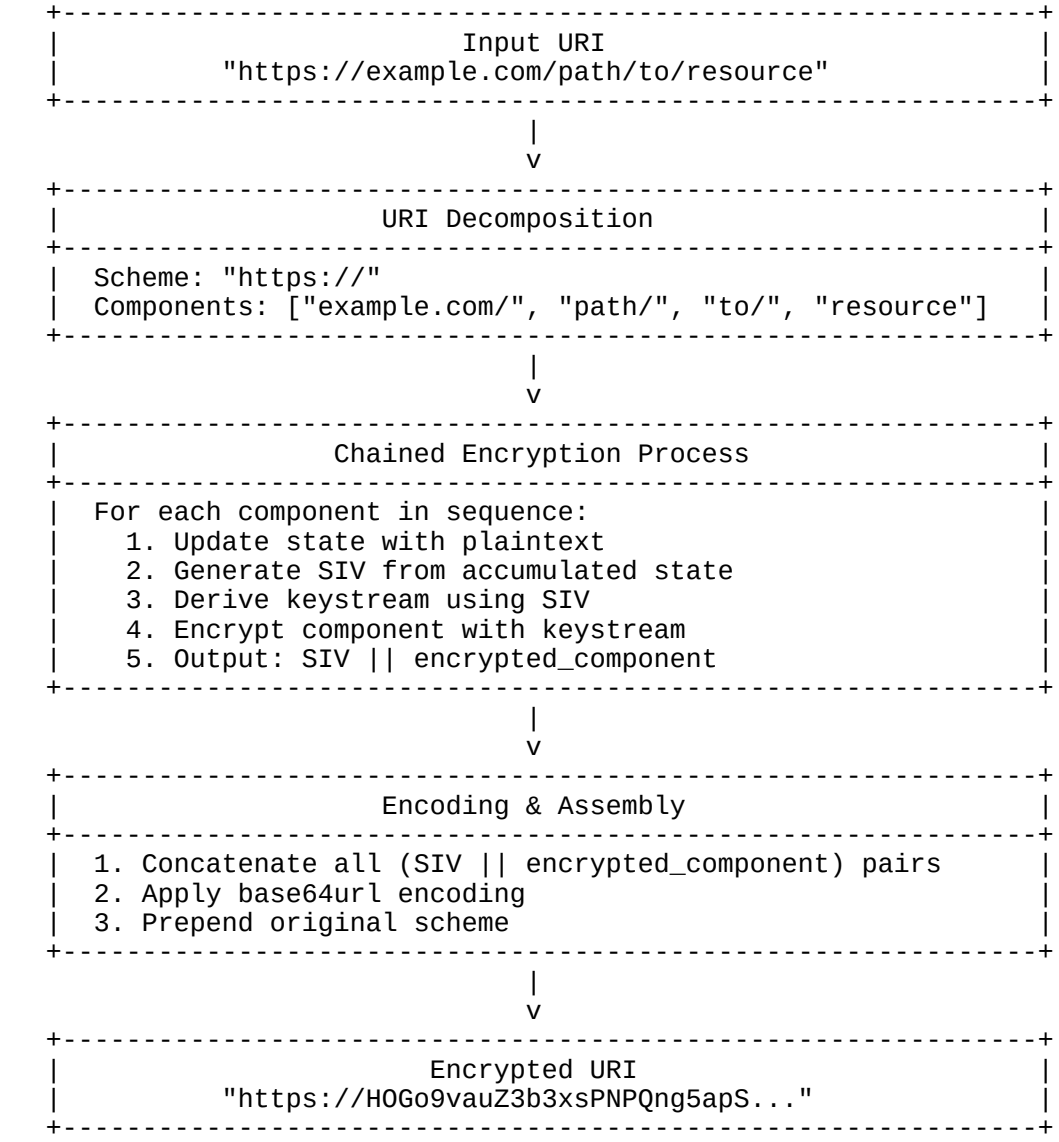
* Prefix-preserving Encryption: An encryption scheme where, if two plaintexts share a common prefix, their corresponding ciphertexts also share a common (encrypted) prefix.

* Chained Encryption: A mode where encryption of each component depends cryptographically on all preceding components.

3. URI Processing

This section describes how URIs are processed for encryption and decryption.

The overall encryption flow transforms a plaintext URI into an encrypted URI while preserving its hierarchical structure:



3.1. URI Component Extraction

Before encryption, a URI must be split into its scheme and path components. The path is further divided into individual components for chained encryption. Components are terminated by '/', '?', or '#' characters, which allows proper handling of query strings and fragments.

3.1.1. Full URIs

For a full URI including a scheme:

Input: "https://example.com/a/b/c"

Components:

- Scheme: "https://"
- Component 1: "example.com/"
- Component 2: "a/"
- Component 3: "b/"
- Component 4: "c"

For a URI with query parameters:

Input: "https://example.com/path?foo=bar&baz=qux"

Components:

- Scheme: "https://"
- Component 1: "example.com/"
- Component 2: "path?"
- Component 3: "foo=bar&baz=qux"

For a URI with a fragment:

Input: "https://example.com/path#section"

Components:

- Scheme: "https://"
- Component 1: "example.com/"
- Component 2: "path#"
- Component 3: "section"

Note that all components except the last include their trailing terminator character (‘/’ , ‘?’ , or ‘#’). This ensures proper reconstruction during decryption.

3.1.2. Path-Only URIs

For absolute paths (URIs starting with ‘/’ but without a scheme), the leading ‘/’ is treated as the first component:

Input: "/a/b/c"

Components:

- Scheme: "" (empty)
- Component 1: "/"
- Component 2: "a/"
- Component 3: "b/"
- Component 4: "c"

For a path with query parameters:

Input: "/path/to/file?param=value"

Components:

- Scheme: "" (empty)
- Component 1: "/"
- Component 2: "path/"
- Component 3: "to/"
- Component 4: "file?"
- Component 5: "param=value"

The leading ‘/’ is explicitly encrypted as a component to maintain consistency and enable proper prefix preservation for absolute paths.

This character receives its own SIV and is encrypted, ensuring that the root path is authenticated like any other path component and that different keys and contexts produce different ciphertexts for that path, consistently with other paths.

In applications where all paths are guaranteed to be absolute and the ’/’ path can be considered a special case, ciphertext expansion can be reduced by removing the leading ’/’ character from the URI prior to encryption, making the path relative with ’/’ as an implicit current path.

3.2. Component Reconstruction

During decryption, components are joined to reconstruct the original path:

Components: ["example.com/", "a/", "b/", "c"]
Reconstructed Path: "example.com/a/b/c"

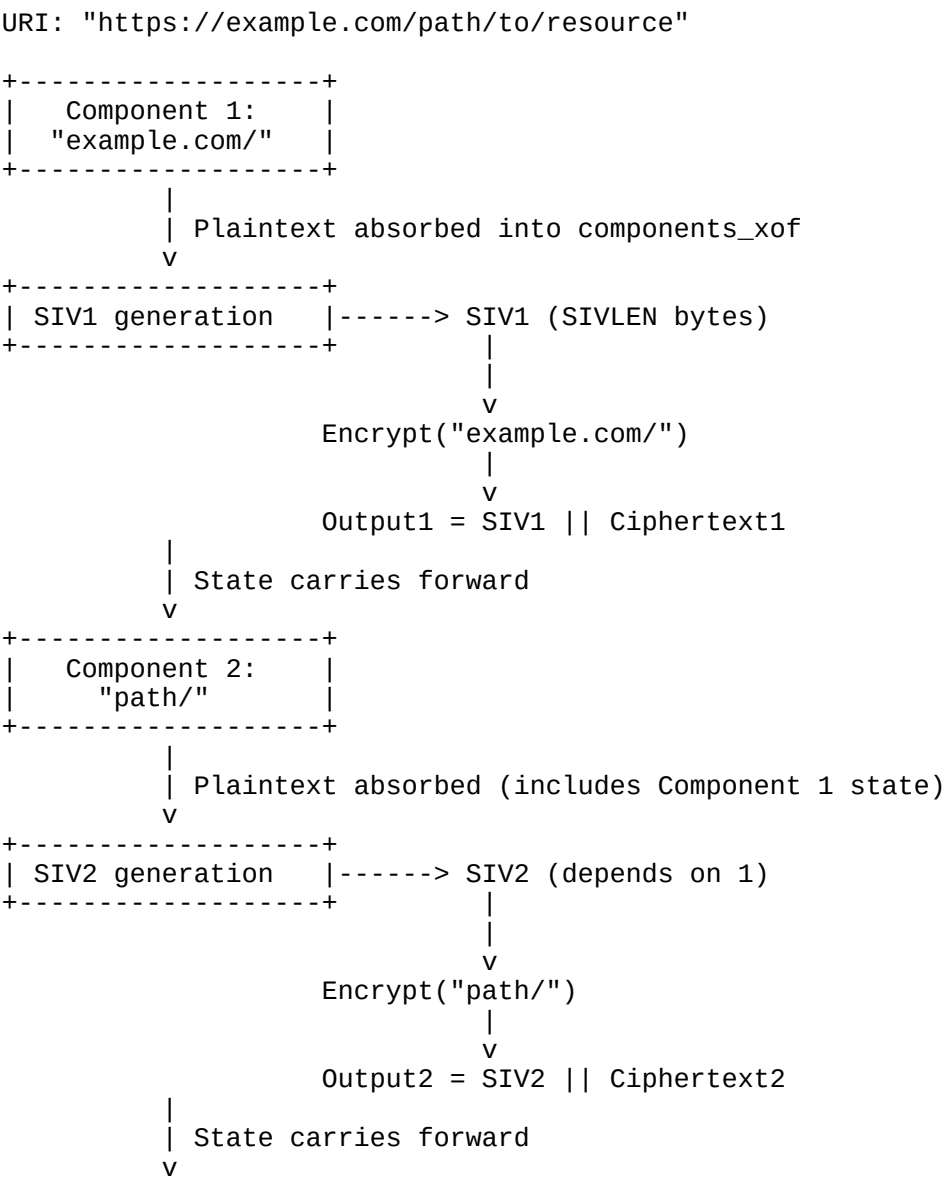
When combined with the scheme: "https://example.com/a/b/c"

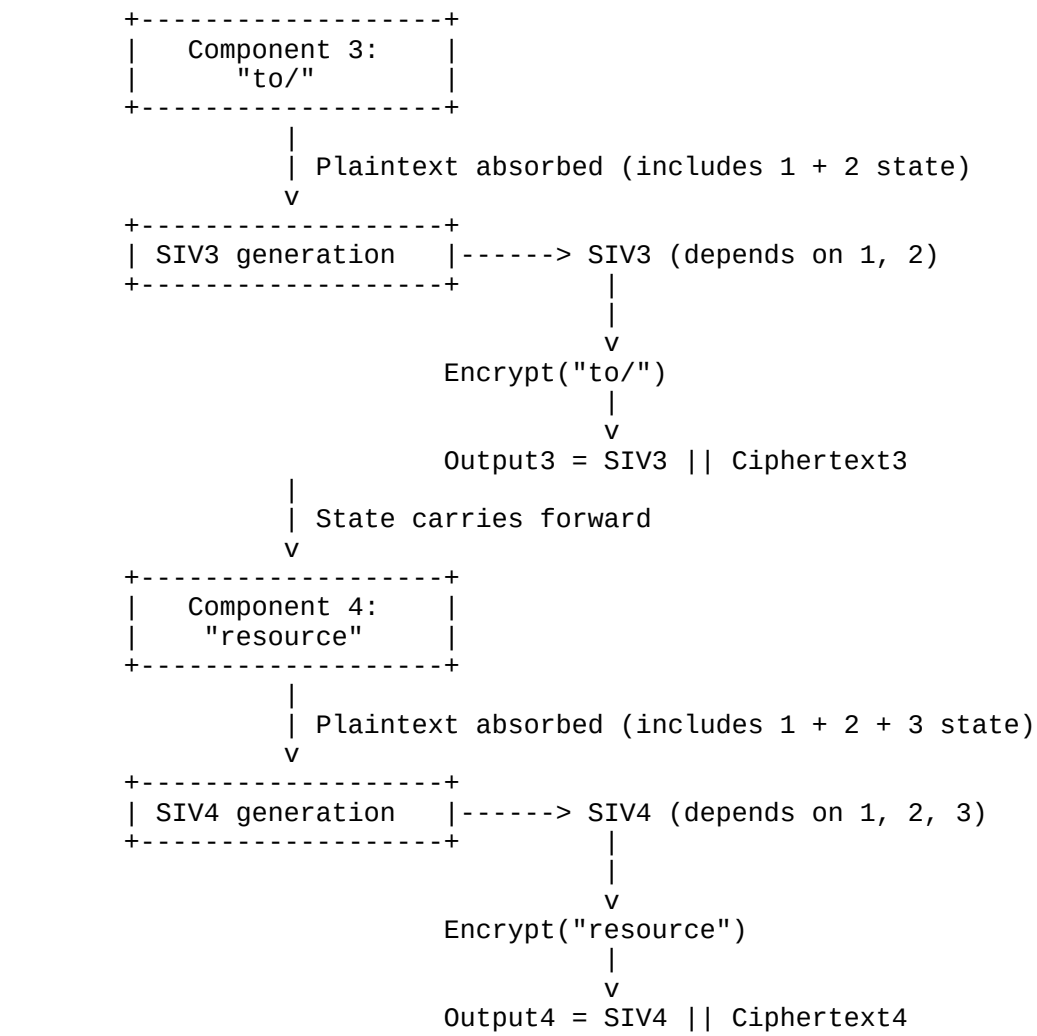
For absolute paths without a scheme:

```
Components: ["/", "a/", "b/", "c"]
Reconstructed Path: "/a/b/c"
```

4. Cryptographic Operations

The chained encryption model creates cryptographic dependencies between components, ensuring prefix preservation.





Final Output: Output1 || Output2 || Output3 || Output4

If URIs share a common prefix example.com/path/, their Output1 and Output2 will be identical.

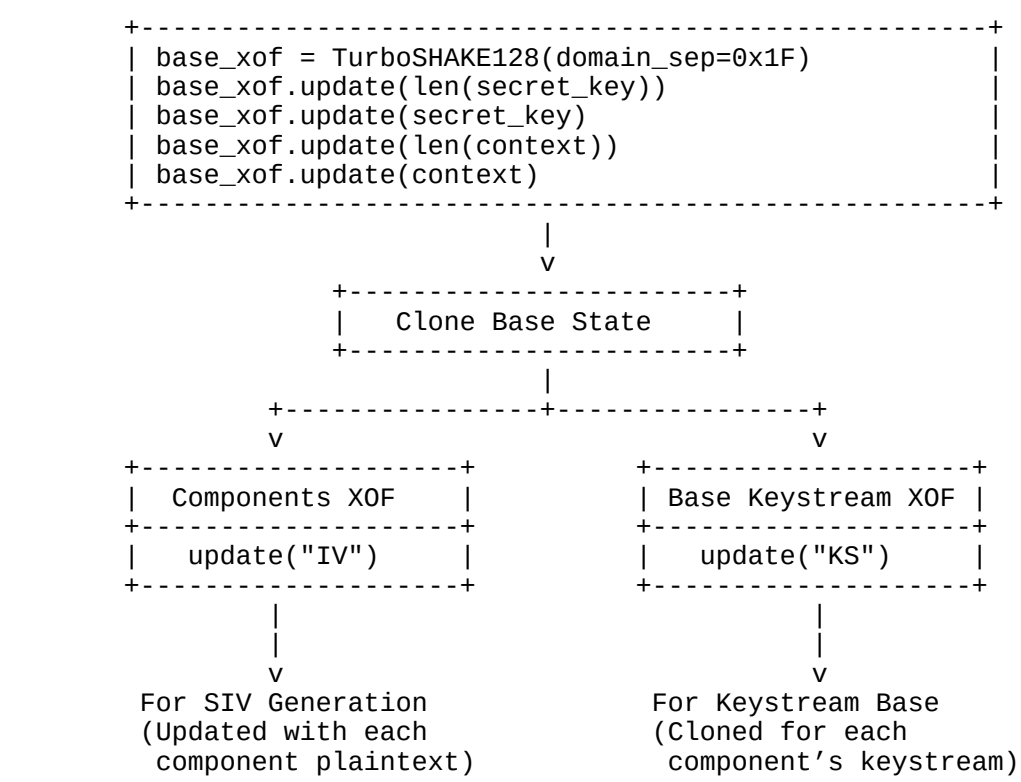
4.1. XOF Initialization

The base XOF is initialized with the secret key and context parameters using length-prefixed encoding to prevent ambiguities.

Two XOF instances are derived from the base XOF:

1. Components XOF: Updated with each component’ s plaintext to generate SIVs

2. Base Keystream XOF: Used as the starting point for generating keystream for each component



The initialization process is:

```
base_xof = TurboSHAKE128()
base_xof.update(len(secret_key))
base_xof.update(secret_key)
base_xof.update(len(context))
base_xof.update(context)

components_xof = base_xof.clone()
components_xof.update("IV")

base_keystream_xof = base_xof.clone()
base_keystream_xof.update("KS")
```

Note on XOF cloning: The .clone() operation creates a new XOF instance with an identical internal state, preserving all previously absorbed data. After cloning, the original and cloned XOFs can be updated and read from independently. This allows the components_xof to maintain a running state across all components while base_keystream_xof remains unchanged for creating per-component keystreams.

4.2. Component Encryption

For each component, the encryption process follows a precise sequence that ensures both confidentiality and authenticity:

1. Update components_xof with the component plaintext

2. Squeeze the SIV from components_xof (SIVLEN bytes). This requires cloning components_xof before reading, as reading may finalize the XOF.

3. Create keystream_xof by cloning base_keystream_xof and updating it with SIV

4. Calculate padding needed for base64 encoding

5. Generate a keystream of length (component_length + padding)

6. XOR the padded component with the keystream

7. Output SIV concatenated with encrypted_component

The padding ensures clean base64url encoding without padding characters. Since base64 encoding works with groups of 3 bytes (producing 4 characters), we pad each (SIV || encrypted_component) pair to have a length that's a multiple of PADBS:

```
total_bytes = SIVLEN (SIV) + component_len
padding_len = (PADBS - total_bytes % PADBS) % PADBS
```

This formula calculates:

* How many bytes are needed to reach the next multiple of PADBS

* The outer modulo handles the case where total_bytes is already a multiple of PADBS

The components_xof maintains state across all components. After generating the SIV for component N, the XOF can be updated with component N+1's plaintext. This chaining ensures that each component's encryption depends on all previous components, thus enabling the prefix-preserving property.

4.3. Component Decryption

For each encrypted component, the decryption process is:

1. Read SIV from input (SIVLEN bytes)
2. Create keystream_xof by cloning base_keystream_xof and updating it with SIV
3. Decrypt bytes incrementally to determine component boundaries:
    * Generate keystream bytes one at a time from the XOF
    * XOR each encrypted byte with its corresponding keystream byte
    * Check each decrypted byte for component terminators ('/', '?', '#')
    * When a terminator is found, the component is complete.
    * Skip any padding bytes (null bytes) after the component
4. Update components_xof with the complete plaintext component (including terminator)
5. Generate the expected SIV from components_xof
6. Compare the expected SIV with the received SIV (constant-time)
7. If mismatch, return error

4.3.1. Component Boundary Detection

During decryption, component boundaries are discovered dynamically by examining the decrypted plaintext:

* Each component (except possibly the last) ends with a terminator character ('/', '?', or '#')
* When a terminator is encountered, we know the component is complete

* After finding the terminator, we skip padding bytes to align to the next PADBS-byte boundary.

* The padding length can be calculated: padding = (PADBS - ((SIV_size + bytes_read) % PADBS)) % PADBS

This approach eliminates the need for explicit length encoding, as the component structure itself provides the necessary boundary information.

Any tampering with the encrypted data will cause the SIV comparison to fail.

4.4. Padding and Encoding

To enable clean base64url encoding without padding characters (‘=’), each encrypted component pair (SIV || ciphertext) is padded to be a multiple of PADBS bytes. This is necessary because base64 encoding processes 3 bytes at a time to produce 4 characters of output.

The padding calculation (PADBS - (SIVLEN + component_len) % PADBS) % PADBS ensures the following:

* If (SIVLEN + component_len) % PADBS = 0: no padding needed (already aligned)

* If (SIVLEN + component_len) % PADBS = 1: add 2 bytes of padding

* If (SIVLEN + component_len) % PADBS = 2: add 1 byte of padding

With the default value of PADBS=3, this padding scheme provides partial length-hiding. For example, with SIVLEN=16, components “abc” , “abcd” , and “abcde” all produce 21-byte outputs after padding. Without the secret key, a passive adversary cannot determine the exact original component size.

The final output is encoded using URL-safe base64 [RFC4648], with ‘- ‘ replacing ‘+’ and ‘_’ replacing ‘/’ for URI compatibility.

5. Algorithm Specification

This section provides the complete algorithms for encryption and decryption. The following functions and operations are used throughout the algorithms:

* TurboSHAKE128(): Creates a new TurboSHAKE128 XOF instance with domain separation parameter 0x1F. This function produces an extensible output function (XOF) that can generate arbitrary-length outputs.

* .update(data): Absorbs the provided data into the XOF state. Data is processed sequentially and updates the internal state of the XOF.

* .read(length): Squeezes the specified number of bytes from the XOF's output. Each call continues from where the previous read left off, producing a continuous stream of pseudorandom bytes.

* .clone(): Creates a new XOF instance with an identical internal state to the original. This enables multiple independent computation paths from the same initial state.

* XOR operation: The bitwise exclusive OR operation between two byte sequences of equal length. This operation is used to combine plaintext with keystream for encryption, and ciphertext with keystream for decryption.

* base64url_encode(data): Converts binary data to a base64 string using URL-safe encoding (replacing '+' with '- ' and '/' with '_') and omitting padding characters.

* base64url_decode(string): Converts a URL-safe base64 string back to binary data, automatically handling the absence of padding characters.

* Stream(data): Creates a sequential reader for binary data, enabling byte-by-byte or block-based access to the contents.

* constant_time_compare(a, b): Compares two byte sequences in constant time, regardless of their contents. This prevents timing attacks by ensuring the comparison duration does not depend on which bytes differ.

* len(data): Returns the length of the provided data in bytes.

* Concatenation: The operation of joining two byte sequences end-to-end to form a single sequence.

* zeros(count): Generates a sequence of zero-valued bytes of the specified length, used for padding.

* remove_padding(data): Removes trailing zero bytes from a byte sequence to recover the original data length.

* join(components): Combines multiple path components into a single path string, preserving the terminator characters ('/', '?', '#') that are included in each component.

5.1. Encryption Algorithm

Input: secret_key, context, uri_string

Output: encrypted_uri

Steps:

1. Split URI into scheme and components
2. Initialize XOF instances as described in Section 4.1
3. encrypted_output = empty byte array
4. For each component:
    * Update components_xof with component.
    * SIV = components_xof.clone().read(SIVLEN).
    * keystream_xof = base_keystream_xof.clone().
    * keystream_xof.update(SIV).
    * padding_len = (PADBS - (SIVLEN + len(component)) % PADBS) % PADBS.
    * keystream = keystream_xof.read(len(component) + padding_len).
    * padded_component = component concatenated with zeros(padding_len).
    * encrypted_part = padded_component XOR keystream.
    * encrypted_output = encrypted_output concatenated with SIV concatenated with encrypted_part.
5. base64_output = base64url_encode(encrypted_output).
6. If scheme is not empty: Return scheme + base64_output
7. Else if original URI started with '/' : Return '//' + base64_output
8. Else: Return base64_output

5.2. Decryption Algorithm

Input: secret_key, context, encrypted_uri

Output: encrypted_uri

Note: For path-only URIs (those starting with '/'), the output format is: - '/' followed by the base64url-encoded encrypted components - This preserves the absolute path indicator in the encrypted form or error

Steps:

1. Split encrypted URI into scheme and base64 part
2. decoded = base64url_decode(base64_part). If decoding fails, return error.
3. Initialize XOF instances as described in Section 4.1
4. decrypted_components = empty list
5. position = 0
6. While position < len(decoded):
    * SIV = decoded[position:position+SIVLEN]. If not enough bytes, return error.
    * keystream_xof = base_keystream_xof.clone().update(SIV).
    * component_start = position + SIVLEN
    * component = empty byte array
    * position = position + SIVLEN
    * While position < len(decoded):
        - decrypted_byte = decoded[position] XOR keystream_xof.read(1)
        - position = position + 1
        - If decrypted_byte == 0x00: continue (skip padding)
        - component.append(decrypted_byte)

- If decrypted_byte is '/', '?', or '#':

  - total_len = position - component_start

  - position = position + ((PADBS - ((SIVLEN + total_len) % PADBS)) % PADBS)

  - Break inner loop

* Update components_xof with component.

* expected_SIV = components_xof.clone().read(SIVLEN).

* If constant_time_compare(SIV, expected_SIV) == false, return error.

* decrypted_components.append(component).

7. decrypted_path = join(decrypted_components).

8. Return scheme + decrypted_path

6. Implementation Details

6.1. TurboSHAKE128 Usage

Implementations MUST use TurboSHAKE128 with a domain separation parameter of 0x1F for all operations. The TurboSHAKE128 XOF is used for:

* Generating SIVs from the components XOF

* Generating keystream for encryption/decryption

* All XOF operations in the initialization

TurboSHAKE128 is specified in [RFC9861] and provides the security properties needed for this construction.

6.2. Key and Context Handling

The secret key MUST be at least SIVLEN bytes long. Keys shorter than SIVLEN bytes MUST be rejected. Implementations SHOULD validate that the key does not consist of repeated patterns (e.g., identical first and second halves) as a best practice.

The context parameter is a string that provides domain separation. Different applications SHOULD use different context strings to prevent cross-application attacks. The context string MAY be empty.

Both key and context are length-prefixed when absorbed into the base XOF:

```
base_xof.update(len(secret_key) as uint8)
base_xof.update(secret_key)
base_xof.update(len(context) as uint8)
base_xof.update(context)
```

The length is encoded as a single byte, limiting keys and contexts to 255 bytes. This is sufficient for all practical use cases.

6.3. Error Handling

Implementations MUST NOT reveal the cause of decryption failures. All error conditions (invalid base64, incorrect padding, SIV mismatch, insufficient data) MUST result in identical, generic error messages.

SIV comparison MUST be performed in constant-time to prevent timing attacks.

7. Security Guarantees

URICrypt provides the following cryptographic security guarantees:

7.1. Confidentiality

URICrypt achieves semantic security for URI path components through its use of TurboSHAKE128 as a pseudorandom function. Each component is encrypted using a unique keystream derived from the following:

* The secret key
* The application context
* A synthetic initialization vector (SIV) that depends on all preceding components

This construction ensures that:

* An attacker without the secret key cannot recover plaintext components from ciphertexts.

* The keystream generation is computationally indistinguishable from random for each unique (key, context, path-prefix) tuple.

* Components are protected by at least 128 bits of security against brute-force attacks.

7.2. Authenticity and Integrity

Each URI component is authenticated through the SIV mechanism:

* The SIV acts as a Message Authentication Code (MAC) computed over the component and all preceding components.

* Any modification to a component will cause the SIV verification to fail during decryption.

* The chained construction ensures that reordering, insertion, or deletion of components is detected.

* Authentication provides 128-bit security against forgery attempts.

7.3. Prefix-Preserving Property

URICrypt maintains a controlled information leakage pattern:

* URIs sharing a common prefix will produce ciphertexts with the same encrypted prefix.

* This property is deterministic and intentional, enabling systems to perform prefix-based operations.

* The leakage is limited to prefix structure only—no information about non-matching suffixes is revealed.

7.4. Domain Separation

The context parameter provides cryptographic domain separation:

* Different contexts with the same key produce completely independent ciphertexts.

* This prevents cross-context attacks where ciphertexts from one application could be used in another.

* Context binding is cryptographically enforced through the XOF initialization.

7.5. Key Commitment

URICrypt provides full key-commitment security.

The scheme is fully key-committing, meaning that a ciphertext can only be decrypted with the exact key that was used to encrypt it. It is computationally infeasible to find two different keys that successfully decrypt the same ciphertext to valid plaintexts.

7.6. Resistance to Common Attacks

URICrypt resists several categories of attacks:

Chosen-plaintext Attacks (CPA): While deterministic, URICrypt is CPA-secure for unique inputs. The determinism is a design requirement for prefix preservation.

Tampering Detection: Any bit flip, truncation, or modification in the ciphertext will be detected with overwhelming probability (1 - 2^-128).

Length-extension Attacks: The use of length-prefixed encoding and domain separation prevents length-extension attacks.

Replay Attacks: Within a single (key, context) pair, replay is possible due to determinism. Applications requiring replay protection should incorporate timestamps or nonces into the context.

Key Recovery: TurboSHAKE128's security properties ensure that observing ciphertexts does not leak information about the secret key.

7.7. Security Bounds

The security of URICrypt is bounded by the following:

* Key strength: Minimum 128-bit security with SIVLEN-byte keys
* Collision resistance: 2^64 birthday bound for SIV collisions
* Authentication security: 2^-128 probability of successful forgery
* Computational security: Based on TurboSHAKE128's proven security as an XOF

7.8. Limitations and Trade-offs

URICrypt makes specific security trade-offs for functionality, including the following:

* Deterministic encryption: Same inputs produce same outputs, enabling certain traffic analysis

* Partial length obfuscation: With PADBS=3, exact component lengths are partially hidden

* Prefix structure leakage: The hierarchical structure of URIs is preserved by design

* SIV length configuration: Implementations MAY adjust SIVLEN for different usage bounds. Larger values (24 or 32 bytes) increase birthday bound resistance at the cost of ciphertext expansion. However, 16 bytes is generally recommended as it provides practical collision resistance with acceptable overhead

* Padding block size configuration: The default PADBS=3 already provides partial length-hiding. Implementations MAY adjust PADBS to increase size obfuscation. Larger values create larger size buckets but increase ciphertext expansion. The value MUST remain a multiple of 3 to ensure efficient Base64url encoding without padding characters

These trade-offs are intentional and necessary for the prefix-preserving functionality. Applications requiring stronger privacy guarantees should evaluate whether URICrypt's properties align with their threat model.

8. Security Considerations

URICrypt provides confidentiality and integrity for URI paths while preserving prefix relationships. The encryption is fully reversible: encrypted URIs can be decrypted to recover the original plaintext URIs, but only with knowledge of the secret key. The security properties depend on:

* Key Secrecy: The security of URICrypt depends entirely on the secrecy of the secret key. Keys MUST be generated using a cryptographically secure random number generator [RFC4086] and stored securely.

* Deterministic Encryption: URICrypt is deterministic - identical inputs produce identical outputs. This allows observers to detect when the same URI is encrypted multiple times. Applications requiring unlinkability SHOULD incorporate additional entropy (e.g., via the context parameter).

* Prefix Preservation: While essential for functionality, prefix preservation leaks information about URI structure. Systems where this information is sensitive SHOULD consider alternative approaches.

* Context Separation: The context parameter prevents cross-context attacks. Applications MUST use distinct contexts for different purposes, even when sharing keys.

* Component Authentication: Each component is authenticated via the SIV mechanism. Any modification, reordering, or truncation of components will be detected during decryption.

* Length Obfuscation: The default PADBS=3 configuration provides partial length-hiding. Applications requiring stronger length-hiding SHOULD consider using larger PADBS values or padding components to fixed lengths.

* Key Reuse: Using the same key with different contexts is safe, but using the same (key, context) pair for different applications is NOT RECOMMENDED.

IANA Considerations

This document has no actions for IANA.

Acknowledgments

The author would like to thank Maciej Soltysiak for highlighting the importance of properly supporting query parameters and fragments in URI encryption.

Normative References

[I-D.draft-denis-ipcrypt]
Denis, F., "Methods for IP Address Encryption and Obfuscation", Work in Progress, Internet-Draft, draft-denis-ipcrypt-12, 19 September 2025, <https://datatracker.ietf.org/doc/html/draft-denis-ipcrypt-12>.

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997, <https://www.rfc-editor.org/rfc/rfc2119>.

[RFC3986] Berners-Lee, T., Fielding, R., and L. Masinter, "Uniform Resource Identifier (URI): Generic Syntax", STD 66, RFC 3986, DOI 10.17487/RFC3986, January 2005, <https://www.rfc-editor.org/rfc/rfc3986>.

[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>.

[RFC4648] Josefsson, S., "The Base16, Base32, and Base64 Data Encodings", RFC 4648, DOI 10.17487/RFC4648, October 2006, <https://www.rfc-editor.org/rfc/rfc4648>.

[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>.

[RFC9861] "*** BROKEN REFERENCE ***".

Appendix A. Pseudocode

A.1. URI Component Extraction

```
   function extract_components(uri_string):
     if uri_string contains "://":
        scheme = substring up to and including "://"
        path = substring after "://"
     else:
        scheme = ""
        path = uri_string

     components = []

     // For absolute paths, treat leading "/" as first component
     if path starts with "/":
        components.append("/")
        path = substring after first "/"

     while path is not empty:
        terminator_pos = find_next_terminator(path)
        if terminator_pos found:
           component = substring(path, 0, terminator_pos + 1)
           path = substring(path, terminator_pos + 1)
           components.append(component)
        else:
           components.append(path)
           path = ""

     return (scheme, components)

   function find_next_terminator(path):
     for i from 0 to length(path) - 1:
        if path[i] == '/' or path[i] == '?' or path[i] == '#':
           return i
     return not_found
```

A.2. XOF Initialization

```
   function initialize_xofs(secret_key, context):
     // Initialize base XOF
     base_xof = TurboSHAKE128(0x1F)

     // Absorb key and context with length prefixes
     base_xof.update(uint8(len(secret_key)))
     base_xof.update(secret_key)
     base_xof.update(uint8(len(context)))
     base_xof.update(context)

     // Create components XOF
     components_xof = base_xof.clone()
     components_xof.update("IV")

     // Create base keystream XOF
     base_keystream_xof = base_xof.clone()
     base_keystream_xof.update("KS")

     return (components_xof, base_keystream_xof)
```

A.3. Encryption Algorithm

```
   function uricrypt_encrypt(secret_key, context, uri_string):
     // Extract components
     (scheme, components) = extract_components(uri_string)

     // Initialize XOF instances with secret key and context
     (components_xof, base_keystream_xof) =
         initialize_xofs(secret_key, context)
     if error: return error

     encrypted_output = byte_array()

     // Process each component
     for component in components:
        // Update components XOF for SIV computation
        components_xof.update(component)

        // Generate SIVLEN-byte Synthetic Initialization Vector (SIV)
        siv = components_xof.squeeze(SIVLEN)

        // Create keystream XOF for this component
        keystream_xof = base_keystream_xof.clone()
        keystream_xof.update(siv)

        // Calculate padding for base64 encoding alignment
        // The total bytes (SIV + component) must be a multiple of PADBS
        // to produce clean base64 output without padding characters
```

```
        component_len = len(component)
        padding_len = (PADBS - (SIVLEN + component_len) % PADBS) % PADBS

        // Generate keystream
        keystream = keystream_xof.squeeze(component_len + padding_len)

        // Pad component to align with base64 encoding requirements
        padded_component = component + byte_array(padding_len)

        // Encrypt using XOR with keystream
        encrypted_part = xor_bytes(padded_component, keystream)

        // Append to output
        encrypted_output.extend(siv)
        encrypted_output.extend(encrypted_part)

     // Base64 encode with URL-safe characters and no padding
     base64_output = base64_urlsafe_no_pad_encode(encrypted_output)

     // Return with appropriate prefix
     if scheme != "":
        return scheme + base64_output
     else if uri_string starts with "/":
        return "/" + base64_output
     else:
        return base64_output
```

A.4. Decryption Algorithm

```
function uricrypt_decrypt(secret_key, context, encrypted_uri):
  // Split scheme and base64
  if encrypted_uri contains "://":
     scheme = substring up to and including "://"
     base64_part = substring after "://"
  else:
     scheme = ""
     base64_part = encrypted_uri

  // Decode base64
  try:
     decoded = base64_urlsafe_no_pad_decode(base64_part)
  catch:
     return error("Decryption failed")

  // Initialize XOF instances with secret key and context
  (components_xof, base_keystream_xof) =
      initialize_xofs(secret_key, context)
  if error: return error
```

```
  decrypted_components = []
  input_stream = ByteStream(decoded)

  // Process each component
  while not input_stream.empty():
     // Read SIV
     siv = input_stream.read(SIVLEN)
     if len(siv) != SIVLEN:
        return error("Decryption failed")

     // Create keystream XOF
     keystream_xof = base_keystream_xof.clone()
     keystream_xof.update(siv)

     // Determine component length by checking padding constraints
     remaining = input_stream.remaining()
     if remaining == 0:
        return error("Decryption failed")

     // Find valid component length by checking padding alignment
     component_data = None
     for possible_len in range(1, remaining + 1):
        total_len = SIVLEN + possible_len
        padding_len = (PADBS - total_len % PADBS) % PADBS
        if possible_len >= padding_len:
           component_data = input_stream.peek(possible_len)
           break

     if component_data is None:
        return error("Decryption failed")

     // Read encrypted data
     encrypted_part = input_stream.read(len(component_data))

     // Generate keystream and decrypt
     keystream = keystream_xof.squeeze(len(encrypted_part))
     padded_plaintext = xor_bytes(encrypted_part, keystream)

     // Remove padding bytes added for base64 alignment
     padding_len = (PADBS - (SIVLEN + len(encrypted_part)) % PADBS) % PADBS
     component = padded_plaintext[:-padding_len] if padding_len > 0 else padded_plaintext

     // Update XOF with plaintext
     components_xof.update(component)

     // Generate expected SIV
     expected_siv = components_xof.squeeze(SIVLEN)
```

```
     // Authenticate using constant-time comparison to prevent timing attacks
     if not constant_time_equal(siv, expected_siv):
        return error("Decryption failed")

     decrypted_components.append(component)

  // Reconstruct URI
  if scheme and decrypted_components:
     path = "".join(decrypted_components)
     return scheme + path
  elif decrypted_components:
     return "/" + "".join(decrypted_components)
  else:
     return ""
```

A.5. Padding and Encoding

```
function calculate_padding(component_len):
  // Calculate padding needed for base64 encoding alignment
  // The combined SIV (SIVLEN bytes) + component must be divisible by PADBS
  // for clean base64 encoding without '=' padding characters
  total_len = SIVLEN + component_len
  return (3 - total_len % 3) % 3

function base64_urlsafe_no_pad_encode(data):
  // Use standard base64 encoding
  encoded = standard_base64_encode(data)
  // Make URL-safe and remove padding for URI compatibility
  encoded = encoded.replace('+', '-')
                   .replace('/', '_')
                   .rstrip('=')
  return encoded

function base64_urlsafe_no_pad_decode(encoded):
  // Add padding if needed for standard decoder
  padding = (4 - len(encoded) % 4) % 4
  if padding > 0:
     encoded = encoded + ('=' * padding)
  // Make standard base64
  encoded = encoded.replace('-', '+')
                   .replace('_', '/')
  // Decode
  return standard_base64_decode(encoded)
```

Appendix B. Test Vectors

These test vectors were generated using the reference Rust implementation of URICrypt with TurboSHAKE128.

```
   Test Configuration:
   secret_key (hex): 0102030405060708090a0b0c0d0e0f10
   context: "test-context"
```

B.1. Test Vector 1: Full URI

```
  Input: "https://example.com/a/b/c"
  Output: "https://HOGo9vauZ3b3xsPNPQng5apSzL5V7QW94C7USgN8mHZJ337AKSWOu
           cUwMuD-uUfF95SsSHCNgBkXUnH1uGll_YtBltXSqKEHNcYJJwbdFdhfWz19"
```

B.2. Test Vector 2: Path-Only URI

```
  Input: "/a/b/c"
  Output: "/b9bCOhqZsvU9XxGOMk6d8QFQhTIdI_xYKpds2lWXpZCms5-az9wtfUft3rec
           3d9YkUo0N7VcxO5MXfxE5UobvgTJX8UpRdNN"
```

B.3. Test Vector 3: Multi-Component Path

```
 Input: "https://cdn.example.com/videos/2025/03/file.mp4"
 Output: "https://hxUM2N3txwYjGxjvCpWn30SznxR0v0fDbkSQgCTXCUu7Rq8iSbWP4
          0OvYxKs9zC3kw1JNzAc4Wuj7RZvRd0VUprJWLs5KJPnWsA9Kguxa_J7XviTS3G
          Tqf-XZdPxYyq1Y1MXVE9_4ojHwm6jBDUkVthAkuNe5Cqk_h6d"
```

B.4. Test Vector 4: Root with Scheme

```
   Input: "https://example.com/"
   Output: "https://HOGo9vauZ3b3xsPNPQng5apSzL5V7QW94C7USgN8"
```

B.5. Test Vector 5: Simple Path

```
  Input: "/path/to/resource"
  Output: "/b9bCOhqZsvU9XxGOMk6d8QFQPTuMlsQKDBhAbc77JvsdRj0kxiFipunATQmm
           CkNhAe0BPP2EqQoxORElY_ukfUYSrr9mIMfiO9joa3Kn5RS7eSKr"
```

B.6. Test Vector 6: URI with Query Parameters

```
  Input: "https://example.com/search?q=test&limit=10"
  Output: "https://HOGo9vauZ3b3xsPNPQng5apSzL5V7QW94C7USgN8cl2BBtuWmxTsI
           Ij59ka3KeDsaqXFGnKgW9aLLR36YvUf9ORkMnVE5PTR_3DiO43hL9WjdSu7L9
           FN"
```

B.7. Test Vector 7: URI with Fragment

```
  Input: "https://docs.example.com/guide#installation"
  Output: "https://ypHTiw0JUMcr4bUjQH9Dxo8wGWHyfFlLq8VrOE-zX6IbgLFxYX_Jm
           2hzivywvrpIBWa-9Jl6nSZLq2pd35QwkDsc1-_Kao2BvyBB19ndu1PpwQv1wy
           uA"
```

B.8. Test Vector 8: URI with Query and Fragment

```
  Input: "/api/v2/users?id=123#profile"
  Output: "/b9bCOhqZsvU9XxGOMk6d8QFQwcP2C3bJVNVZDge7zfub_ai4x6LaUlXp-XjZ
           XOgZlLloIbasK-JKlbeKeKV2rctq5bX9zQh1KogN2zaggTMZioUb4kwGIKp8Z
           y744xQwGDG64n6GhN56XEM8LvBfJuEj6ZgsjeLbTPIMbCmO0pJhzVSh"
```

Author's Address

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