Oboron Protocol Specification

oboron.org

version 1.0

Oboron is a string-in/string-out *authenticated* symmetric encryption protocol for UTF-8 text. This document is normative and specifies the authenticated schemes (the Oboron core). The unauthenticated and obfuscation schemes are specified separately in the Oboron Unauthenticated Layer (obu) Specification. The companion CLI contract is specified separately in the Oboron CLI Specification.

The key words MUST, MUST NOT, REQUIRED, SHALL, SHALL NOT, SHOULD, SHOULD NOT, RECOMMENDED, MAY, and OPTIONAL in this document are to be interpreted as described in BCP 14 (RFC 2119, RFC 8174) when, and only when, they appear in all capitals, as shown here.

1 Formats

An Oboron **format** is the full transformation of a plaintext string into encrypted text, in two stages:

1. *Encryption*: the plaintext string, taken as its UTF-8 bytes, is encrypted by the scheme's AEAD algorithm, producing **scheme output** bytes (an optional nonce, the encrypted plaintext, and an authentication tag, as fixed by the scheme).
2. *Encoding*: the scheme output bytes are encoded to a string — the **obtext**.

"Encoding" in this document always means this byte-to-text step; the UTF-8 interpretation of the plaintext is called out explicitly wherever it matters.

1.1 Scheme + Encoding = Format

A format combines a scheme (cryptographic algorithm) with an encoding (string representation):

- *Scheme*: cryptographic algorithm + mode + parameters (e.g. `dsiv`).
- *Encoding*: string representation method (e.g. `c32`).
- *Format*: scheme + encoding = the complete transformation (e.g. `dsiv.c32`).

Given an encryption key, the format thus uniquely specifies the complete transformation from a plaintext string to an encoded obtext string. Formats are written as identifiers:

- `ob:{scheme}.{encoding}` — prefixed-identifier syntax, e.g. `ob:dsiv.c32` (`ob:` is a bare prefix, not a registered URI scheme);

- `{scheme}.{encoding}` — when the context is clear.

The following requirements apply to Oboron APIs:

- The `ob:` namespace prefix is *not* used in the `oboron` API. Formats like `dsiv.c32` MUST be used directly.
- The public interface MUST use `enc` / `dec` names for methods and functions. The `enc` operation comprises the full process, including the encryption and encoding stages.

Format identifiers are lowercase ASCII and case-sensitive. Implementations MUST reject a malformed identifier: an unknown scheme or encoding, uppercase letters, whitespace, an empty component, an extra separator, or the `ob:` prefix where an unprefixed format is expected.

1.2 Encodings

All encodings MUST produce output with **no padding characters**.

- `b32` — standard base32 (RFC 4648 §6): **uppercase**, no padding.
  Alphabet: `ABCDEFGHIJKLMNOPQRSTUVWXYZ234567`.
- `c32` — Crockford base32: **lowercase**, no padding.
  Alphabet: `0123456789abcdefghjkmnpqrstvwxyz` (excludes `i`, `l`, `o`, `u`). This is the canonical Oboron profile of Crockford base32: no check symbols, no hyphens, and no ambiguous-character aliases. It uses the same 5-bit grouping as `b32` (RFC 4648 §6), differing only in alphabet.
- `b64` — URL-safe base64 (RFC 4648 §5): most compact, case-sensitive, no padding.
  Alphabet:
  `ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789-_`.
- `hex` — hexadecimal: **lowercase**, no padding.
  Alphabet: `0123456789abcdef`.
  Longest output (two characters per byte).

Decoding. Encoders MUST emit only the canonical form above. Decoders MUST reject any non-canonical input: no padding characters (including `=`), no whitespace or line breaks, no prefix or suffix, and no character outside the selected alphabet — which is uppercase-only for `b32`, lowercase-only for `c32` and `hex`, and the URL-safe alphabet for `b64`. Decoders MUST also reject inputs in which the unused trailing bits of the final `b32`, `c32`, or `b64` symbol are non-zero, and inputs whose length is impossible for the encoding: for `b32` and `c32` an unpadded length of 1, 3, or 6 modulo 8 is invalid; for `b64` a length of 1 modulo 4 is invalid; for `hex` the length MUST be even. An implementation MAY offer a separate, explicitly non-canonical compatibility mode, but the canonical decoder is strict.

1.3 Schemes

Schemes define the encryption algorithm and its properties. Every core scheme is **authenticated** — it MUST provide both confidentiality and integrity protection. (Unauthenticated and obfuscation schemes are specified separately in the Oboron Unauthenticated Layer (obu) Specification.)

A scheme ID has two parts: a one-letter property code followed by an algorithm code — e.g. `dsiv` is `d` (deterministic) + `siv` (AES-SIV).

Scheme Properties

The first letter of the scheme ID describes its properties:

- **`d` — deterministic.** For a fixed key and format, the same plaintext always produces the same obtext. Examples: `dsiv`, `dgcmsiv`.
- **`p` — probabilistic.** A fresh random nonce per encryption makes repeated encryptions of the same plaintext differ, except with negligible probability. Examples: `psiv`, `pgcmsiv`.

Informal note. Every Oboron scheme is also *avalanche*: the AEAD authentication tag is computed over the whole plaintext and seeds the keystream, so a one-byte plaintext change is expected to change the obtext broadly. Because this holds universally across the schemes, it is not a distinguishing axis and is not encoded in the scheme ID.

Scheme Algorithms

The algorithm code is the AEAD algorithm's standard name with the `AES-` prefix removed, hyphens dropped, and the result lowercased:

- `siv` = AES-SIV
- `gcmsiv` = AES-GCM-SIV

Summary Table

Scheme	Algorithm	Deterministic
`dsiv`	AES-SIV	Yes
`dgcmsiv`	AES-GCM-SIV	Yes
`psiv`	AES-SIV	No
`pgcmsiv`	AES-GCM-SIV	No

Choosing a Scheme

- Deterministic (`dsiv`, `dgcmsiv`): compact output, and usable as a stable lookup key or reference; note that, for a given key and format, they reveal when two plaintexts are equal.
- Probabilistic (`psiv`, `pgcmsiv`): repeated encryptions of the same plaintext under the same key and format produce independent obtexts that differ except with negligible probability (larger output, due to the nonce).

`dsiv`/`psiv` use AES-SIV; `dgcmsiv`/`pgcmsiv` are their AES-GCM-SIV counterparts. The two families differ in performance: AES-SIV is typically faster on short inputs, while AES-GCM-SIV scales better and is typically faster on larger inputs — in practice the crossover is around 256 bytes. Match the family to the expected input sizes.

Every scheme MUST use AES-256. The SIV schemes take a 64-byte AES-SIV key (two 256-bit subkeys; §3.1); the GCM-SIV schemes use AES-256-GCM-SIV with a 32-byte key-generating key.

2 Algorithm

Oboron combines encryption and text encoding in a single operation. The following terms are used throughout:

- **enc**: the combined encryption and encoding operation.
- **dec**: the combined decoding and decryption operation.
- **scheme output**: the exact byte string a scheme produces before text encoding — depending on the scheme, an optional nonce, the encrypted plaintext, and an authentication tag.
- **encrypted plaintext**: the encrypted bytes produced by the AEAD algorithm, excluding any nonce or authentication tag.
- **authentication tag**: the tag produced by the AEAD algorithm and verified by `dec`.
- **obtext**: the text encoding of the scheme output — the string returned by `enc`. The scheme output bytes are an internal detail and are *not* exposed in the public API.

The high-level process flow is:

```
enc:  plaintext (string) -> UTF-8 bytes  -> encryption
                         -> scheme output (bytes) -> encoding
                         -> obtext (string)

dec:  obtext (string) -> decoding -> scheme output (bytes)
                      -> decryption -> UTF-8 bytes -> plaintext (string)
```

The obtext is the text encoding of the scheme output and nothing more. Apart from its alphabet, its length, and — for deterministic schemes — equality with other obtexts, the scheme output is intended to be computationally indistinguishable from random to anyone without the key. The scheme is not carried in the obtext; it is supplied by the caller, who constructs the codec for a specific format (§1.1). A `dec` operation decrypts under its configured format; supplying the wrong scheme fails authentication except with negligible probability, rather than silently returning unauthenticated plaintext.

2.1 Padding

The schemes use CTR-based AEAD modes (AES-SIV and AES-GCM-SIV) and require no block padding. The custom CBC padding used by the unauthenticated layer is specified in the Oboron Unauthenticated Layer (obu) Specification.

2.2 Per-Scheme Output Layout

This section specifies the exact byte layout of each scheme's output — the bytes the obtext encodes directly. Implementations MUST conform to these layouts for cross-implementation

interoperability. During decryption, an implementation MUST parse the scheme output according to the configured scheme, supply any nonce to the AEAD operation exactly as specified below, and MUST reject any input shorter than the scheme's minimum layout length.

Deterministic schemes (no random nonce)

- **`dsiv` (AES-SIV):** AES-SIV (RFC 5297) with a *zero-element* associated-data vector (S2V is invoked with no associated-data components, not with one empty component) and no external nonce; the SIV value is the synthetic IV. Output layout: `16-byte SIV tag || encrypted plaintext`.
- **`dgcmsiv` (AES-GCM-SIV):** AES-GCM-SIV with a fixed all-zero 12-byte nonce (`[0x00; 12]`) and zero-length additional authenticated data (AAD). Output layout: `encrypted plaintext || 16-byte authentication tag`. The fixed nonce is *not* included in the output.

Probabilistic schemes (random nonce prepended to output)

- **`psiv` (AES-SIV):** AES-SIV with a fresh random 16-byte nonce per encryption. The associated-data vector contains *exactly one* element — the nonce bytes — and `dec` MUST supply that same element. Output layout: `16-byte random nonce || 16-byte SIV tag || encrypted plaintext`.
- **`pgcmsiv` (AES-GCM-SIV):** AES-GCM-SIV with a fresh random 12-byte nonce per encryption and zero-length AAD. Output layout: `12-byte random nonce || encrypted plaintext || 16-byte authentication tag`.

Summary:

Scheme	Algorithm	Nonce in output	Scheme output layout (bytes)
`dsiv`	AES-SIV	No	`16-byte SIV tag || encrypted plaintext`
`dgcmsiv`	AES-GCM-SIV	No (zero nonce fixed)	`encrypted plaintext || 16-byte tag`
`psiv`	AES-SIV	16-byte nonce (prepended)	`16-byte nonce || 16-byte SIV tag || encrypted plaintext`
`pgcmsiv`	AES-GCM-SIV	12-byte nonce (prepended)	`12-byte nonce || encrypted plaintext || 16-byte tag`

The minimum scheme-output lengths (before text encoding) follow from these layouts:

Scheme	Minimum output
`dsiv`	16 bytes
`dgcmsiv`	16 bytes
`psiv`	32 bytes

Scheme	Minimum output
`pgcmsiv`	28 bytes

These are layout minima only. A decoded scheme output of minimum length may authenticate and decrypt to zero plaintext bytes, in which case `dec` MUST still reject it, because the empty string is outside the Oboron plaintext domain.

3 Key Management

3.1 Single Master Key Model

Oboron uses a single 512-bit master key. Key material is obtained from it by scheme family:

Scheme	Key material
`dsiv`, `psiv`	Full 64-byte master key, used directly (no derivation)
`dgcmsiv`, `pgcmsiv`	32-byte key from `HKDF-Expand` over the master (shared by both)

SIV family (`dsiv`, `psiv`). The full 64-byte master key is passed directly to the AES-SIV primitive as the combined key per RFC 5297. AES-SIV internally splits it into two 256-bit subkeys: bytes 0–31 as the S2V (CMAC) authentication key and bytes 32–63 as the CTR-mode encryption key. The 64-byte key is interpreted as these two subkeys in the order AES-SIV requires. An implementation using a combined-key AES-SIV API MUST pass the 64 bytes in this order; one built on lower-level primitives MUST produce the same subkey interpretation. Both `dsiv` and `psiv` use this AES-SIV key, distinguished only by associated-data structure (`dsiv`: a zero-element vector; `psiv`: a one-element vector holding the random nonce). Sharing one key across the two is safe because AES-SIV is secure for arbitrary associated data under a single key; its nonce-misuse resistance further means `psiv` degrades only to equality leakage if a random nonce ever repeats.

GCM-SIV family (`dgcmsiv`, `pgcmsiv`). The 32-byte AES-256-GCM-SIV key-generating key is derived from the master with `HKDF-Expand` (HKDF using HMAC-SHA-256, RFC 5869):

```
key = HKDF-Expand(PRK = master, info = "gcmsiv", L = 32)
```

The HKDF-Extract step is omitted: the 512-bit master is already a uniformly random key and serves directly as the pseudorandom key (PRK). Implementations MUST compute `HKDF-Expand` only; they MUST NOT prepend an `HKDF-Extract` step or use an Extract-then-Expand convenience (a single `HKDF` call with a null or empty salt is *not* equivalent). Here `master` denotes the 64 decoded key bytes, not the 128 hex characters. The `info` value is the fixed ASCII string `gcmsiv` (the 6 bytes `67 63 6d 73 69 76`), shared by both GCM-SIV schemes, so `dgcmsiv` and `pgcmsiv` derive the *same* 32-byte key. As in the SIV family, sharing one key across the two is safe: AES-GCM-SIV is secure for arbitrary nonce selection under a single key, so a nonce collision degrades only to the equality leakage `dgcmsiv` already exposes by design.

Implementations MUST NOT place the master key in obtexts, error messages, logs, or test-vector output; provisioning and storage of the master key are the application's responsibility.

3.2 Key Format

The canonical key text encoding is **hexadecimal**: 128 lowercase hex characters for the 512-bit master key. Hex is the format accepted by codec constructors and produced by key generation utilities. It has no alphabet variants, no padding rules, and every 128-character canonical-hex string decodes to a valid 64-byte key (see §3.3). The key material is the 64 decoded bytes, not the 128 hex characters. Keys are typically read from an environment variable, so the string form is fed directly into the constructor. Hex is the only key text encoding; there is no base64 key form.

3.3 Valid Keys

A canonical Oboron key is a string of exactly 128 lowercase ASCII hexadecimal characters (`0`–`9`, `a`–`f`), encoding 64 bytes of key material. Implementations MUST reject any key string of a different length, and MUST reject characters outside this alphabet (including uppercase) unless a non-canonical compatibility mode is explicitly documented. Key generation MUST emit only canonical lowercase hex.

There are no weak-key exclusions: every one of the 2^512 possible 64-byte values is a valid master key, so no canonical key string is rejected for the *value* it encodes.

3.4 Alternative Key Interfaces

In addition to canonical hex key strings, implementations MUST support construction from exactly 64 raw key bytes, for programmatic use, and MUST reject raw keys of any other length.

4 Protocol API

All Oboron implementations MUST provide the following abstract interface.

4.1 Core Operations

- `enc(plaintext: string) -> obtext: string` — encrypts and encodes the plaintext using the configured format; returns an obtext string.
- `dec(obtext: string) -> plaintext: string` — decodes and decrypts the obtext; returns the original plaintext.

UTF-8 and normalization. Oboron operates on the UTF-8 bytes of the plaintext exactly as supplied; it performs no Unicode normalization, so callers that require it MUST normalize before calling `enc`. `enc` MUST reject input that is not valid UTF-8 (relevant where the host string type can hold arbitrary bytes). `dec` MUST validate that the decrypted bytes are valid UTF-8 and MUST return an error on invalid input, never returning an unchecked string.

Implementations MAY provide a non-core helper that returns the authenticated plaintext bytes before UTF-8 validation; such a helper is outside the normative `dec` interface and MUST NOT be used to treat non-UTF-8 plaintext as conforming Oboron input.

> **Format detection.** The scheme is not encoded in the obtext; `dec` operates on the caller's configured format. An implementation can offer a best-effort convenience that trial-decrypts a configured set of candidate formats and selects the one whose authentication tag verifies. Such format detection is outside the normative Oboron interface.

Empty plaintext. The empty string is outside the Oboron plaintext domain. `enc` MUST return an error when given an empty plaintext, and `dec` MUST return an error if a successfully authenticated decryption yields zero bytes.

4.2 Codec Construction

A codec MUST be constructible from a key and a format specifier. Implementations MUST support construction from a hex key string and from raw key bytes.

In Rust:

```
use oboron::DsivC32;

let ob = DsivC32::new(&env::var("OBORON_KEY")?)?;
```

In Python:

```
from oboron import DsivC32
import os

ob = DsivC32(os.getenv("OBORON_KEY"))
```

In Go:

```
import "oboron.org/go/oboron"

ob, err := oboron.NewDsivC32(os.Getenv("OBORON_KEY"))
```

4.3 Key Generation

All implementations MUST provide a key generation utility that outputs a valid 512-bit key as 128 lowercase hex characters. The 64 key bytes MUST be drawn from a cryptographically secure random number generator and sampled uniformly from the 2^512 possible values; if suitable randomness is unavailable, key generation MUST fail rather than output a predictable or low-entropy key.

In Rust:

```
let key = oboron::generate_key();

cargo run --bin keygen
```

In Python:

```
key = oboron.generate_key()

python -m oboron.keygen
```

In Go:

```
key := oboron.GenerateKey()
```

```
go run oboron.org/go/cmd/keygen
```

5 Security Considerations

Oboron targets confidentiality — scheme output computationally indistinguishable from random without the key — and integrity with negligible forgery probability, against an active adversary who observes obtexts and `dec` outcomes. Each codec operates under one key and format; key or endpoint compromise, traffic analysis, and denial of service are out of scope. The considerations below state the residual leakages and operational limits within that model.

Authentication. Every Oboron obtext is authenticated. `dec` MUST verify the authentication tag and MUST return an error on failure; a corrupted obtext, a wrong key, or a wrong scheme fails the tag check — except with negligible forgery probability — rather than yielding altered plaintext. The obtext carries no unauthenticated framing — it is the encoding of the scheme output and nothing more.

Fixed nonce. `dgcmsiv` encrypts with a fixed all-zero nonce, making it deterministic. This is sound only because AES-GCM-SIV is nonce-misuse-resistant (RFC 8452): nonce reuse does not cause the catastrophic two-time-pad failure of plain AES-GCM, and the only confidentiality loss is the equality leakage `dgcmsiv` already exposes by design. The binding limit is therefore on data volume rather than nonce reuse: security degrades only as the total data encrypted under one key approaches the AES-GCM-SIV birthday bound, which for Oboron's short-string workload is far out of practical reach. Because the codec is stateless, honoring that bound is a deployment responsibility; callers encrypting at high volume under one key SHOULD rotate the master key well before it. `dgcmsiv` and `pgcmsiv` share one derived key (§3.1), so their volumes draw on the same budget. The fixed-nonce construction MUST NOT be transplanted onto plain AES-GCM, where nonce reuse is catastrophic.

Key separation. The GCM-SIV family derives its key from the master via HKDF while the SIV family uses the master directly (§3.1). Each family shares one key across its deterministic and probabilistic schemes; both AEADs are nonce-misuse-resistant, so the sharing is safe. The cross-family separation is one-directional: a compromise of an SIV key recovers the master, and hence the GCM-SIV key, whereas a compromise of the GCM-SIV key reveals only that derived key.

Deterministic schemes. `dsiv` and `dgcmsiv` are deterministic: for a fixed key and format, identical plaintexts produce identical obtexts, revealing equality and, in low-cardinality domains (country codes, status flags, short identifiers), frequency patterns. Use the probabilistic schemes (`psiv`, `pgcmsiv`) where equality or frequency must not leak, unless deterministic lookup is a required property. This leakage is over the exact UTF-8 bytes supplied to `enc`: visually identical strings in different Unicode normalization forms are distinct plaintexts and do not collide.

Length leakage. No scheme hides plaintext length: the obtext length reveals the UTF-8 plaintext length plus the scheme's fixed overhead (16 bytes for `dsiv` and `dgcmsiv`, 32

for `psiv`, 28 for `pgcmsiv`), expanded by the encoding. Pad before `enc` if length must be concealed.

Randomness. Probabilistic schemes MUST draw each nonce from a cryptographically secure random number generator. If suitable randomness is unavailable, `enc` MUST fail rather than reuse or synthesize a nonce.

Uniform errors. All `dec` failures — decode errors, tag-verification failures, invalid UTF-8 — SHOULD be reported through a single uniform error, so that `dec` does not become a distinguishing oracle.

Key scope. An Oboron master key SHOULD be used only with Oboron; applications SHOULD NOT reuse the same key bytes with other protocols or cryptographic libraries.

Format migration. Obtexts are not self-describing — the format is supplied out of band. Applications that store obtexts across possible format changes SHOULD record the format identifier alongside each obtext, outside the obtext itself.

Context and replay. An obtext attests only that its plaintext was produced by a holder of the key under the given format; Oboron binds no recipient, purpose, or expiry to it and provides no replay protection. Applications that need freshness, audience, or context separation MUST encode those fields in the plaintext and verify them after `dec`.

Key handling. The master key's confinement is specified in §3.1; in addition, implementations SHOULD zeroize the master key and any derived subkeys once they are no longer needed.

6 Conformance

A conforming Oboron core implementation MUST implement every scheme in §1.3 and every encoding in §1.2, and MUST support every scheme–encoding combination as a format. It MUST reject unknown scheme identifiers, unknown encoding identifiers, and malformed format identifiers, and MUST NOT treat unauthenticated or obfuscation schemes as core schemes.

Conforming implementations MUST be mutually interoperable:

- For deterministic formats, they MUST produce identical obtexts for the same key, format, and plaintext.
- For probabilistic formats, they MUST decrypt obtexts produced by any other conforming implementation.

Conformance is checked against the common test vectors published at `https://gitlab.com/oboron/oboron-test-vectors`; implementations MUST pass the vector set carried by the release tag matching this specification's version. The vectors bind to a single fixed public test key and carry no per-vector key field. Where the repository and this specification disagree, the vector set pinned by this specification's release takes precedence; a vector that contradicts this document's normative text is an erratum to be corrected, not an override of it. The scheme set (§1.3) and encoding set (§1.2) are closed for version 1.0: new schemes or encodings require a new specification version, distinguished out of band since obtexts are not self-describing. The Rust, Python, and Go reference implementations are maintained for this specification and are expected to pass that vector set.

References

The following standards are normative for this specification:

- BCP 14 — RFC 2119, *Key words for use in RFCs to Indicate Requirement Levels*; RFC 8174, *Ambiguity of Uppercase vs. Lowercase in RFC 2119 Key Words*.
- RFC 4648 — *The Base16, Base32, and Base64 Data Encodings*.
- RFC 5297 — *Synthetic Initialization Vector (SIV) Authenticated Encryption Using the Advanced Encryption Standard (AES)*.
- RFC 8452 — *AES-GCM-SIV: Nonce Misuse-Resistant Authenticated Encryption*.
- RFC 5869 — *HMAC-based Extract-and-Expand Key Derivation Function (HKDF)*.
- FIPS 197 — *Advanced Encryption Standard (AES)*.
- NIST SP 800-38B — *Recommendation for Block Cipher Modes of Operation: the CMAC Mode for Authentication*.

Index