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

N. Sullivan
Cryptography Consulting LLC

15 March 2026

SAFE: Sealed, Algorithm-Flexible Envelope
draft-sullivan-safe-01

Abstract

SAFE defines an encryption envelope that encrypts a payload once for multiple recipients. Decryption can require multiple credentials in sequence (public keys, passphrases, or other registered methods), so that no single compromise reveals content. The format targets large, writable files: it supports streaming decryption, random-access reads at block granularity, and selective re-encryption of modified blocks without re-keying. Per-block Authenticated Encryption with Associated Data (AEAD) provides confidentiality and integrity and detects reordering, truncation, and extension. SAFE accommodates post-quantum key encapsulation without format changes, provides algorithm agility through IANA registries, supports recipient privacy, and defines application profiles for common deployment scenarios.

Table of Contents

1. Introduction

SAFE is an encryption format for files and objects. A SAFE-encoded file contains an encrypted payload and one or more LOCK blocks. Each LOCK wraps a Content-Encryption Key (CEK) for one recipient; multiple LOCKs allow multiple recipients to decrypt the same payload without duplicating ciphertext. A LOCK can require several credentials in sequence (a passphrase AND a private key, for example), so neither factor alone suffices.

The payload is split into fixed-size blocks, each encrypted with Authenticated Encryption with Associated Data (AEAD) [RFC5116] and a per-block random nonce. Blocks can be decrypted individually or streamed sequentially. An aligned binary layout (Section 6.4.2) places each ciphertext block at a predictable offset, enabling O(1) random reads. Per-block random nonces (Section 5.7.5) allow individual blocks to be re-encrypted without re-wrapping the CEK or touching other blocks.

Existing formats address subsets of these capabilities; Section 2 surveys the differences. SAFE provides algorithm agility through IANA registries (Section 10) and accommodates post-quantum key encapsulation mechanisms without format changes. Recipient privacy modes (Section 8.6) allow Hybrid Public Key Encryption (HPKE) steps to omit key identifiers, preventing passive observers from linking files to recipients. Application profiles (Section 7.4) provide baseline algorithm guidance for common deployment scenarios.

For example, a deployment might require that documents be decryptable only with both a passphrase AND a recipient private key. The LOCK block for such a recipient would contain two Step lines:

```
Step: pass(kdf=argon2id, salt=...)
Step: hpke(kem=p-256, id=..., kemct=...)
```

Both steps are evaluated with the known passphrase and private key to derive the key that wraps the CEK. Neither factor alone suffices. See Section 5.7.1.1 for the cryptographic details.

The payload layer and LOCK layer serve distinct roles. The LOCK layer manages key encapsulation and multi-factor access control, wrapping the CEK independently for each recipient. The payload layer provides per-block authenticated encryption with support for random-access reads and selective re-encryption of modified blocks. A file-level commitment value precedes independently decryptable fixed-size blocks. Confidentiality and integrity are enforced per block. Block AAD authenticates both the block index and a final-block indicator, so a Decryptor can detect block substitution, reordering, truncation,

and extension (Section 5.7.9). A 32-octet commitment prefix derived from the CEK, the negotiated algorithm parameters, and a per-file salt provides uniform key commitment across all AEAD choices (Section 8.12). [FLOE] formalizes a similar set of properties under the term random-access authenticated encryption (raAE).

SAFE's registered AEADs fall into two usage classes. Nonce-Misuse-Resistant (NMR) and non-NMR suites. Non-NMR suites (AES-256-GCM, ChaCha20-Poly1305, AEGIS-256, AEGIS-256X2) use stored per-block random nonces; their security depends on nonce uniqueness and the total number of block encryptions under a given payload key. The NMR suite (AES-256-GCM-SIV, where SIV denotes Synthetic Initialization Vector) uses derived nonces and tolerates nonce reuse, degrading to deterministic encryption rather than permitting plaintext recovery. Because block rewrites consume additional encryptions under the same payload key, the relevant usage budget is not just the current file size but the total lifetime block encryptions, including all rewrites. See Section 8.13 for per-AEAD analysis.

2. Comparison with Related Formats

The following table compares SAFE with existing encryption formats on the capabilities most relevant to encrypted file storage. X indicates native support, P indicates that the capability is achievable but is not a design goal of the format, and - indicates no support.

Capability	SAFE	JWE	CMS	OpenPGP	S/MIME	SFrame	Age	AEA	FLOE	Chunked OHTTP
Large-file framing	X	P	P	P	P	-	X	X	X	X
Streaming decrypt	X	P	X	X	X	X	X	X	X	P
Random-access reads	X	-	-	-	-	P	-	X	X	-
Random-access writes	X	-	-	-	-	P	-	-	X	-
Random per-block nonces	X	-	-	-	-	-	-	-	X	-
Multi-recipient (single ciphertext)	X	X	X	X	X	-	P	-	-	-
Multi-factor per recipient	X	P	P	-	P	-	-	-	-	-
Algorithm agility	X	X	X	X	X	X	-	-	P	P
Restricts insecure configurations	X	P	P	P	P	P	X	X	X	-

Table 1

JWE ([RFC7516]) encrypts the entire plaintext as a single AEAD operation; its JSON Serialization wraps the Content-Encryption Key per recipient but defines no block structure for streaming or random access.

CMS ([RFC5652]) wraps a content-encryption key per recipient and supports streaming, but defines no fixed-size blocks for random access or selective rewrite.

OpenPGP ([RFC9580]) similarly wraps a session key per recipient via PKESK packets and supports streaming through partial body lengths. Like CMS, it provides no block structure for random access or selective rewrite. Both CMS and OpenPGP provide recipient-level composition (multiple recipients, each with one key), not per-recipient multi-factor.

SFrame ([RFC9605]) targets real-time media: low-latency per-frame AEAD for conferencing, not stored-object encryption.

Age [AGE] streams fixed-size chunks with counter-derived nonces and wraps keys per recipient but deliberately avoids algorithm agility (single cipher suite, no registries). Counter-based nonces prevent selective editing; modifying any block requires re-encrypting the entire payload. Age's payload AEAD (ChaCha20-Poly1305) is not key-committing; [AGE-COMMIT] demonstrates that a malicious encryptor can craft recipient stanzas that unwrap to different file keys while the same ciphertext decrypts under each, limiting multi-recipient robustness against targeted attacks.

AEA [AEA] encrypts segments (1 MB default, 256 per cluster) using hierarchical HKDF-SHA256 key derivation and AES-256-CTR with HMAC-SHA256 per segment (encrypt-then-MAC). Per-segment keys are derived deterministically from the main key and segment index, enabling parallel decryption and random-access reads without decrypting preceding segments. Key encapsulation uses Elliptic Curve Diffie-Hellman Ephemeral (ECDHE) P-256 for a single recipient or scrypt for passwords; there is no multi-recipient key wrapping and no algorithm agility (the cipher suite is fixed per profile).

FLOE [FLOE] defines a rewritable authenticated encryption scheme (raAE) with epoch-based key rotation for segment-level random access. SAFE adapts FLOE's epoch key rotation (Section 5.7.4) but adds multi-recipient key wrapping, multi-factor authentication, and algorithm agility.

Chunked OHTTP [I-D.ietf-ohai-chunked-ohttp] splits HTTP message bodies into individually encrypted chunks for incremental processing through an oblivious relay. It targets live HTTP streaming, not stored-object encryption, and provides no random access or multi-recipient support.

SAFE's block construction builds on the STREAM [STREAM] streaming AEAD pattern (truncation detection via a last-block indicator in Section 5.7.9) and extends it with per-block nonces (Section 5.7.5) so that individual blocks can be re-encrypted without re-wrapping the CEK. Non-NMR suites use stored random nonces, which are compatible with AEAD implementations that provide only random IV generation,

including FIPS-validated AES-GCM modules per [NIST-SP-800-38D] Section 8.2.2. The NMR suite uses deterministic per-block nonces derived from the key schedule and block index, tolerating nonce reuse during rewrites (Section 8.9.2).

3. Protocol Overview

This section summarizes the encryption and decryption procedures. Normative details appear in the referenced sections.

Given a plaintext and a set of recipients (each defined by one or more credentials), an Encryptor produces a SAFE object:

1. Select AEAD, Block-Size, and Hash (or use defaults).
2. Generate a random 32-octet CEK using SafeRandom (Section 5.4.2).
3. For each recipient, build a LOCK: generate step artifacts (salts, KEM ciphertexts), derive a KEK, and wrap the CEK.
4. Generate a 32-octet per-file salt: SafeRandom(32, "SAFE-SALT").
5. Derive payload_key and acc_key from CEK and salt (Section 5.7.3).
6. Split plaintext into blocks; encrypt each with a per-block nonce.
7. Compute the snapshot accumulator from block tags (Section 5.7.8).
8. Write the file: optional CONFIG, LOCK blocks, and payload (salt is the first 32 octets of the DATA block).

Given a SAFE object and the appropriate credentials (private keys, passphrases, or other step inputs), a Decryptor recovers the plaintext:

1. Parse CONFIG, LOCK, and DATA blocks.
2. Try each LOCK until one succeeds: evaluate its steps with the recipient’s credentials to derive a KEK, then unwrap the CEK.
3. Read the 32-octet salt from the start of the DATA block.
4. Verify the commitment prefix.
5. Derive payload_key and acc_key from CEK and salt (Section 5.7.3).

6. If all N block nonces and tags are available (i.e., not streaming or partial random-access reads), verify the snapshot accumulator (Section 5.7.8).

7. Decrypt requested blocks.

The CEK enables multi-recipient encryption (wrap once per recipient). The KEK binds each recipient's credentials to the CEK. The payload_key is derived from the CEK, the encryption_parameters (the AEAD, Block-Size, and Hash; Section 5.5), and the per-file salt, providing domain separation: distinct salts produce distinct payload keys even under the same CEK.

4. Conventions and Notation

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.

Header text is UTF-8. Base64 means [RFC4648] with padding equals signs and no line wrapping in the source value. When Base64 values appear in SAFE blocks, Encryptors SHOULD wrap lines at 64 characters; Decryptors MUST accept any line length and MUST ignore line breaks within Base64 values. LF denotes the newline U+000A; Encryptors MUST use LF, Decryptors MUST accept LF and MAY accept CRLF.

String constants used in Encode AAD labels are ASCII and begin with SAFE- (e.g., SAFE-DATA, SAFE-STEP). SafeDerive labels are ASCII (e.g., commit, kek); the protocol prefix SAFE-v1 is added automatically.

ABNF follows [RFC5234]. The following common ABNF rule is used throughout this document:

```
BASE64CHAR = ALPHA / DIGIT / "+" / "/" / "="
```

This rule is a loose character-class approximation. Implementations MUST validate Base64 encoding per [RFC4648] Section 4, including correct padding placement.

4.1. Notation

This document uses the following notation:

Symbol	Meaning
||	Octet string concatenation
XOR	Bitwise exclusive-or of equal-length octet strings
len(x)	Length of x in octets
x[i:j]	Slice of x from octet i (inclusive) to j (exclusive), zero-indexed
uint8(n)	8-bit unsigned integer n (single octet)
I2OSP(n, w)	w-octet big-endian encoding of non-negative integer n
lp16(x)	I2OSP(len(x), 2) || x — 2-octet length-prefixed encoding
Encode(x1, ..., xn)	lp16(x1) || ... || lp16(xn) — multi-value length-prefixed encoding
...x	List expansion: if x is a list, each element becomes a separate argument; if x is a single octet string, it is passed as one argument
uint64(n)	64-bit unsigned integer n in network byte order (big-endian)
floor(x)	Largest integer less than or equal to x
ceil(x)	Smallest integer greater than or equal to x

Table 2

All integers serialized in binary are unsigned and use network byte order (big-endian). Multi-byte integer fields are serialized most-significant byte first.

4.2. Text Encoding

SAFE header lines (fence markers, field names, field values) MUST contain only ASCII printable characters (0x20-0x7E) plus LF (0x0A). Derive info strings and AEAD AAD prefixes use ASCII. Decryptors MUST reject malformed UTF-8 in text fields.

SAFE uses standard Base64 per Section 4 of [RFC4648]. Padding is REQUIRED. Base64 values in headers MAY wrap across lines; continuation lines MUST begin with whitespace. Decryptors MUST strip leading whitespace from continuation lines before decoding. In armored encoding, the DATA block's Base64 MAY contain line breaks; Decryptors MUST ignore them.

Encryptors MUST use LF (0x0A) line terminators. Decryptors MUST accept LF and MAY accept CRLF. Decryptors MUST strip trailing whitespace from header lines.

Case sensitivity: All field names, identifiers, and fence markers are case-sensitive.

4.3. Terminology

CEK (Content-Encryption Key): A randomly generated 32-octet key used to derive the payload encryption key. The CEK is wrapped independently for each recipient.

KEK (Key-Encryption Key): A 32-octet key derived from a LOCK's step sequence. Used to wrap or unwrap the CEK.

Encryptor: The party that creates a SAFE-encoded file.

Decryptor: The party that recovers plaintext from a SAFE-encoded file using appropriate credentials.

5. Algorithms

5.1. Default Parameters

The following defaults apply whenever a CONFIG block is absent or when a field is omitted from the CONFIG block:

Field	Default Value
AEAD	aes-256-gcm
Block-Size	65536
Hash	sha-256
Key-Epoch	(absent)
Lock-Encoding	armored
Data-Encoding	armored

Table 3

Implementations MUST use these values for any omitted fields. CONFIG need only include fields that differ from the defaults; see Section 6.1.

5.2. Algorithm Summary Tables

This section provides a quick reference of all cryptographic algorithms and identifiers used in SAFE. Detailed specifications appear in later sections.

5.2.1. AEAD Algorithms

Algorithm	Identifier	Nk	Nn	NMR	Key-Epoch
AES-256-GCM	aes-256-gcm	32	12	No	Recommended
ChaCha20-Poly1305	chacha20-poly1305	32	12	No	Required
AES-256-GCM-SIV	aes-256-gcm-siv	32	12	Yes	Not Applicable
AEGIS-256	aegis-256	32	32	No	Not Recommended
AEGIS-256X2	aegis-256x2	32	32	No	Not Recommended

Table 4

Nk/Nn are key/nonce sizes in octets. "NMR" indicates nonce-misuse resistance (see Section 8.9). "Key-Epoch" indicates whether Encryptors should include Key-Epoch in CONFIG: Required means MUST, Recommended means SHOULD, Not Applicable means MUST NOT (NMR AEADs), Not Recommended means SHOULD NOT (256-bit nonce AEADs gain no practical benefit). See Section 6.1 and Section 5.7.4 for details. AEADs without NMR permit plaintext recovery under nonce reuse; Encryptors SHOULD select an NMR AEAD when nonce reuse cannot be ruled out. All AEADs provide [RFC5116] semantics with 16-octet tags. All SAFE DATA payloads begin with a 32-octet salt followed by a 32-octet commitment prefix derived per Section 5.7.3 that binds the ciphertext to the CEK, the negotiated algorithm parameters, and the per-file salt (see Section 8.12).

5.2.2. Key Encapsulation Mechanisms

KEM	Identifier	HPKE KEM ID	Encap Size	Auth
X25519	x25519	0x0020	32 octets	Yes
P-256	p-256	0x0010	65 octets	Yes
ML-KEM-768	ml-kem-768	0x0041	1088 octets	No

Table 5

HPKE KEM IDs are defined in Section 7.1 of [RFC9180] and the IANA HPKE KEM Identifiers registry. ML-KEM-768 enables hybrid post-quantum constructions via multi-step step sequences (see Section 5.6.3.2).

All KEMs use HPKE [RFC9180] in export-only mode (AEAD ID 0xFFFF). The encryptor calls SetupBaseS (or SetupAuthS when sid or shint is present) to produce a KEM ciphertext and an HPKE context, then calls Export on the context to derive the step secret (Section 5.6.3.4). When a sender parameter is present, HPKE Auth mode is used (see Section 5.6.3.1). The KEM identifier appears in hpke(...) step tokens (Section 5.6.3). SAFE maintains a registry mapping string identifiers to HPKE KEM IDs (Section 10.2).

5.2.3. Step Types

Step Type	Token Format	Parameters	Secret
Passphrase	pass(kdf=..., salt=...)	kdf, salt	32 octets
HPKE	hpke(kem=X, kemct=..., ...)	kem, kemct, id/ hint, sid/shint	32 octets

Table 6

Step types are composed via multiple Step lines within a LOCK block, with AND semantics: all steps are required to derive the KEK. Each step conforms to the interface defined in Section 5.6 and produces a 32-octet secret that contributes to KEK derivation (Section 5.7.1). Additional step types MAY be registered per Section 10.4.

5.3. Algorithm Requirements

AEADs used with SAFE MUST provide [RFC5116] semantics with a 16-octet authentication tag. KEMs MUST use HPKE export-only mode (AEAD ID 0xFFFF) as specified in Section 5.6.3. When sid or shint is present, HPKE Auth mode is used (Section 5.6.3.1).

5.4. Hash Function and KDF

SAFE piggybacks on the HPKE KDF interface. HPKE [RFC9180] and [I-D.ietf-hpke-pq] define two KDF classes:

Two-stage KDFs (e.g., HKDF-SHA256, KDF ID 0x0001):

* KDF.Nh: output size of Extract (32 for HKDF-SHA256)

* KDF.Extract(salt, ikm) -> prk (Nh octets)

* KDF.Expand(prk, info, L) -> okm (L octets)

Single-stage KDFs (e.g., TurboSHAKE256, [I-D.ietf-hpke-pq]):

* KDF.Derive(ikm, L) -> okm (L octets). For TurboSHAKE256: TurboSHAKE256(M=ikm, D=0x1F, L).

The Hash config parameter selects both the KDF and its class. Any KDF registered for SAFE MUST be registered in the HPKE KDF Identifiers registry [RFC9180] and MUST implement either the two-stage (Extract/Expand/Nh) or single-stage (Derive) interface.

5.4.1. SafeDerive

SafeDerive binds every derivation to the protocol version and a call-site label. The ikm and info parameters accept a single octet string or a list; list elements are individually lp16-encoded by Encode().

Callers that require suite binding include encryption_parameters in the info argument. Config-independent derivations such as key identifiers (Section 5.6.3.3) omit it.

Two-stage (HKDF):

```
SafeDerive(label, ikm, info, L):
    prk = Extract(
        "SAFE-v1",
        Encode("SAFE-v1", label, ...ikm))
    return Expand(prk,
        Encode("SAFE-v1", label, ...info,
               I2OSP(L, 2)), L)
```

Single-stage (Extendable Output Function (XOF)):

```
SafeDerive(label, ikm, info, L):
    return Derive(
        Encode("SAFE-v1", label, ...ikm,
               I2OSP(L, 2), ...info), L)
```

SafeDerive labels MUST be unique across all call sites within a single protocol version. The labels defined by this specification are reserved; future extensions and step type registrations MUST NOT reuse them. Similarly, SafeRandom labels (Section 5.4.2) MUST be unique and MUST NOT be reused by extensions.

5.4.2. Random Generation (SafeRandom)

SafeRandom is the random generation function used for all encryptor-generated random values in SAFE. It requires a cryptographically secure pseudorandom number generator (CSPRNG).

5.4.2.1. Base Construction

```
SafeRandom(n, label):
    return CSPRNG(n)
```

When no private key is available, SafeRandom returns raw CSPRNG output. The label parameter is reserved for use by the hedged construction (Section 5.4.2.2).

5.4.2.2. Hedged Construction

When a long-term private key sk is available, the Encryptor SHOULD mix it into SafeRandom to defend against a weak or attacker-influenced RNG. The construction follows [RFC8937]: hedge_key is a deterministic function of sk (replacing the signature in [RFC8937] with a KDF, since SAFE does not require a signature scheme), and SafeRandom combines it with CSPRNG output via SafeDerive.

```
hedge_key = SafeDerive("SAFE-HEDGE", sk, "", 32)

SafeRandom(n, label):
    return SafeDerive(label,
        [hedge_key, CSPRNG(n)], "", n)
```

An adversary who can predict CSPRNG output but does not know sk cannot predict the hedged values. Hedging does not prevent repeated output from RNG state duplication (VM snapshot restore, process fork without reseed); identical CSPRNG output produces identical hedged output regardless of the private key.

Suitable private keys include any long-term key held by the Encryptor (an HPKE sender private key, an application-provided signing key, or similar). The key need not correspond to any LOCK step.

Encryptors MUST use SafeRandom for all random values generated during SAFE encoding: CEK, payload salt, passphrase salt, lock_nonce, and block nonce base. HPKE internal randomness (Encap) is not hedged by default. Implementations whose HPKE library accepts an external randomness source SHOULD supply SafeRandom(Nrand, "SAFE-ENCAP") instead of raw CSPRNG output, where Nrand is the randomness length required by the KEM.

A functioning CSPRNG is REQUIRED when no private key is available. See Section 8.9 for the security analysis.

5.5. Encryption Parameters

The encryption_parameters is an ordered list of the effective parameters (defaults augmented by any config overrides):

```
encryption_parameters = [aead_id, block_size, hash_id]
```

When Key-Epoch is present in CONFIG, the list includes it as a fourth element:

```
encryption_parameters = [aead_id, block_size, hash_id,
                         key_epoch_str]
```

where key_epoch_str is the decimal ASCII representation of Key-Epoch with no leading zeros. When Key-Epoch is absent, encryption_parameters has three elements, preserving backward compatibility with files that predate this feature. Because Key-Epoch changes encryption_parameters, adding or removing it requires re-encryption of the payload and re-wrapping of the CEK in new LOCKs.

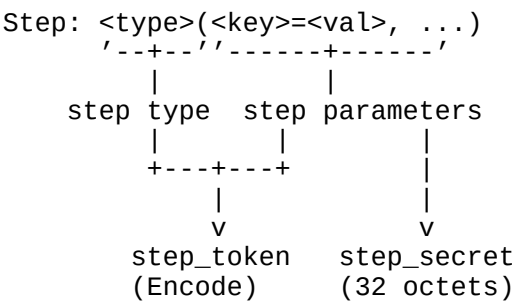
aead_id, block_size, and hash_id are the ASCII string forms of the AEAD, Block-Size, and Hash parameters respectively. SafeDerive splices this list via ...encryption_parameters and applies lp16 framing to each element (Section 5.4.1).

AEAD identifiers MUST be lowercase ASCII and match exactly the registered values in Section 10.1. Block-Size MUST be rendered as a decimal string with no leading zeros (except for the value 0 itself). Hash identifiers MUST be lowercase ASCII and match exactly the registered values in Section 10.3.

SafeDerive binds encryption_parameters throughout the key schedule: the KEK aggregator initialization and final derivation (Section 5.7.1), and each payload schedule call (Section 5.7.3). See Appendix G for a worked example.

5.6. Steps

Step terminology at a glance:



5.6.1. Step Interface

A step is a registered cryptographic operation that produces a 32-octet step secret from user-supplied credentials or cryptographic material. Each step type MUST define:

Step name: A unique ASCII identifier used in step tokens (e.g., "pass", "hpke").

Parameters: An ABNF grammar for step-specific parameters appearing in the token (e.g., kem=x25519, id=...).

Derivation: A deterministic algorithm that produces exactly 32 octets. The algorithm MUST be reproducible given the same inputs. The step definition MUST specify all required inputs for both encryption and decryption.

KEK schedule integration: The step secret and binding step_token feed into the KEK schedule (Section 5.7.1) via SafeDerive("kek_step", [agg, step_secret], step_token, 32). The step secret (ikm) enters Extract; the step token (info) enters Expand. The on-wire step token appears verbatim in the LOCK block. Each step type defines an Encode form: the canonical binary encoding of its cryptographically relevant fields via Encode() (Section 4.1). The Encode form serves as the binding step_token in the KEK schedule. Display-only fields (label, hint, shint) are excluded. The binding forms for the built-in step types are:

Step	Binding step_token
pass	Encode("pass", kdf, salt)
hpke	Encode("hpke", kem, kemct, id)
hpke (auth)	Encode("hpke", kem, kemct, id, "auth", sid)

Table 7

String values (kdf, kem) are UTF-8; binary values (salt, kemct, id, sid) are raw decoded octets, not Base64.

Fields computed for binding (hpke id, hpke sid, webauthn-prf rpid) may be omitted on-wire for privacy but are deterministically reconstructed during decryption and always appear in the binding form. For hpke, id is always present in the binding step_token

even when omitted from the on-wire token; it is computed during trial decryption. Similarly, sid is optional on-wire but always present in auth-mode binding; when omitted, it is computed from the candidate sender public key.

Registration: New step types are registered via the IANA SAFE Step Names registry (Section 10.4) with Specification Required policy. The registration MUST include: step name, parameters grammar, inputs, derivation algorithm, Encode binding form, and any step-specific parameter definitions. See Appendix L for an example.

label (OPTIONAL, any step): A human-readable display name intended to help users identify which passphrase, credential, or key to use during decryption (e.g., "Work laptop", "Recovery key"). The label is always excluded from the binding step_token and has no cryptographic effect. Encryptors MAY include a label in any step token; Decryptors MUST ignore it for binding purposes. The label value MUST match the grammar 1*(ALPHA / DIGIT / "-").

Steps are registered in the IANA SAFE Step Names registry (Section 10.4). The following subsections define the initial registered steps.

5.6.2. Passphrase Step

The passphrase step derives a 32-octet step secret from a user passphrase using a password-based KDF. The kdf parameter selects the algorithm:

KDF	Algorithm	Parameters
argon2id	Argon2id [RFC9106]	m=65536 KiB, t=2, p=1
pbkdf2	PBKDF2-HMAC-SHA-256 [RFC8018]	iter=600000

Table 8

The step token format is:

```
pass(kdf=<kdf>,salt=<Base64>)
pass(kdf=<kdf>,salt=<Base64>,label=<text>)
```

The kdf parameter is REQUIRED. The label parameter is OPTIONAL and is for display only; it is not included in the binding step_token (Section 5.6). Encryptors MUST generate a fresh 16-octet salt using SafeRandom(16, "SAFE-PASS-SALT") for each pass(...) step in a LOCK. Decryptors MUST reject pass(...) steps whose salt value does not decode to exactly 16 octets.

Grammar:

```
pass-step   = "pass(" pass-params ")"
pass-params = "kdf=" kdf-name "," "salt=" salt
              [ "," "label=" label-value ]
kdf-name    = "argon2id" / "pbkdf2"
salt        = 1*BASE64CHAR
label-value = 1*( ALPHA / DIGIT / "-" )
```

Encode form:

Encode("pass", kdf, salt)

Binding step_token: Encode("pass", kdf, salt).

The step secret is computed as follows:

For kdf=argon2id: Argon2id(passphrase, salt, m=65536, t=2, p=1, T=32) per Section 3.1 of [RFC9106].

For kdf=pbkdf2: PBKDF2(PRF=HMAC-SHA-256, Password=passphrase, Salt=salt, c=600000, dkLen=32).

In both cases, salt is the decoded value of the salt parameter.

Implementations SHOULD prefer argon2id for its memory-hardness properties. Implementations MAY support pbkdf2 for environments where Argon2id is not permitted by policy.

5.6.3. HPKE Step

The HPKE step token has three forms:

```
hpke(kem=x25519, kemct=<Base64>, id=<Base64>)    ; Identified mode
hpke(kem=x25519, kemct=<Base64>, hint=<digits>)  ; Hinted mode
hpke(kem=x25519, kemct=<Base64>)                 ; Anonymous mode
```

The parameters are:

kem (REQUIRED): The KEM algorithm. Supported values: x25519, p-256, ml-kem-768.

kemct (REQUIRED): The Base64-encoded HPKE KEM encapsulated key material (the KEM ciphertext). This value MUST decode to the encapsulated key length for the selected KEM (see Section 5.2.2). Decryptors MUST reject hpke steps whose decoded kemct length does not match the KEM's encapsulated key length.

id (OPTIONAL): The key identifier computed as SafeDerive("SAFE-SPKI-v1", spki_der, "", 32) using the configured Hash (default: sha-256). When present, Decryptors match this value against their local keys. When omitted, Decryptors perform trial decryption. See Section 5.6.3.3 and Section 8.6.

hint (OPTIONAL): A 4-digit decimal value (0000-9999) assigned by the recipient out-of-band; not solely dependent on the public key. When present, Decryptors filter candidate keys to those associated with this hint in their local key storage. Mutually exclusive with id.

Encryptors MUST include exactly one of: id, hint, or neither (but not both id and hint).

5.6.3.1. HPKE Auth Mode

In Base mode, any party who knows a recipient's public key can create a valid SAFE object for that recipient. Auth mode [RFC9180] uses SetupAuthS/SetupAuthR, which bind the HPKE context to the sender's private key so that the Decryptor can verify who produced the object. This is useful for offline encrypted file exchange where the recipient needs assurance of origin (for example, encrypted firmware images, signed-then-encrypted document workflows, or air-gapped key escrow) without requiring a separate signature layer.

The presence of sid or shint selects HPKE Auth mode (mode_auth) instead of Base mode. Auth mode MUST only be used with Diffie-Hellman KEM (DHKEM) based KEMs (x25519, p-256). Encryptors MUST NOT include sid or shint with ml-kem-768 or other non-DHKEM KEMs, because these KEMs do not define AuthEncap/AuthDecap.

sid (OPTIONAL): The sender's key identifier, computed as SafeDerive("SAFE-SPKI-v1", spki_der, "", 32) using the same Hash as id. When present with a Base64 value, Decryptors match it against known sender public keys. The special value anon indicates anonymous sender Auth mode: the sender's key is not identified, and Decryptors perform trial decryption across candidate sender keys. Mutually exclusive with shint.

shint (OPTIONAL): A 4-digit decimal value (0000-9999) assigned by

the sender out-of-band; parallels hint for recipient keys. When present, Decryptors filter candidate sender keys to those associated with this value. Mutually exclusive with sid.

Encryptors MUST include exactly one of sid or shint (but not both) when using Auth mode.

Auth mode token forms extend the base forms:

```
hpke(kem=x25519, kemct=<B64>, id=<B64>, sid=<B64>)
hpke(kem=x25519, kemct=<B64>, sid=<B64>)
hpke(kem=x25519, kemct=<B64>, sid=anon)
hpke(kem=x25519, kemct=<B64>, shint=1234)
```

All combinations of recipient identification (id, hint, or anonymous) and sender identification (sid, shint, or sid=anon) are valid.

The hpke step token refines the step-token grammar in Section 6.2.1.1. OWS (optional whitespace) is permitted after each comma separator, consistent with step-params:

```
hpke-step    = "hpke(" hpke-params ")"
hpke-params  = "kem=" kem-name "," OWS "kemct=" kemct
               [ "," OWS recipient-id ]
               [ "," OWS sender-id ]
kem-name     = 1*( ALPHA / DIGIT / "-" )
                 ; registered in SAFE KEM Identifiers
                 ; registry ({{iana-kem}})
kemct        = 1*BASE64CHAR
recipient-id = "id=" 1*BASE64CHAR / "hint=" hint-value
sender-id    = "sid=" ( 1*BASE64CHAR / "anon" )
             / "shint=" hint-value
hint-value   = 4DIGIT
```

Encryptors MUST NOT include both id and hint, and MUST NOT include both sid and shint.

Each HPKE step uses HPKE [RFC9180] in export-only mode with ciphersuite (KEM_ID, KDF_ID, 0xFFFF) constructed from the KEM's registered identifiers (Section 5.6.3.2). AEAD ID 0xFFFF disables Seal/Open; only Export is used.

For Base mode (default):

```
;; Encryptor
(kemct, ctx) = SetupBaseS(pkR, info="SAFE-v1")
step_secret  = ctx.Export(exporter_context, L=32)

;; Decryptor
ctx          = SetupBaseR(kemct, skR, info="SAFE-v1")
step_secret  = ctx.Export(exporter_context, L=32)
```

For Auth mode (sid or shint present):

```
;; Encryptor
(kemct, ctx) = SetupAuthS(pkR, skS, info="SAFE-v1")
step_secret  = ctx.Export(exporter_context, L=32)

;; Decryptor
ctx          = SetupAuthR(kemct, skR, pkS, info="SAFE-v1")
step_secret  = ctx.Export(exporter_context, L=32)
```

The kemct value is the KEM ciphertext (enc in HPKE terminology). The exporter_context is defined in Section 5.6.3.4.

This design uses HPKE's standardized key schedule and export interface for KDF agility, while SAFE's own SafeDerive function handles KEK aggregation, payload key derivation, and nonce constructions.

5.6.3.2. Supported KEMs

The following table lists the KEMs defined in the IANA HPKE KEM Identifiers registry [RFC9180] that are recognized by SAFE:

KEM	KEM ID	KDF ID	HPKE Ciphersuite	Key Encoding
x25519	0x0020	(see below)	(0x0020, KDF_ID, 0xFFFF)	[RFC8410]
p-256	0x0010	(see below)	(0x0010, KDF_ID, 0xFFFF)	[RFC5480]
ml-kem-768	0x0041	(see below)	(0x0041, KDF_ID, 0xFFFF)	(see below)

Table 9

The HPKE Ciphersuite column shows the (KEM_ID, KDF_ID, AEAD_ID) triple used with HPKE's Setup functions. AEAD ID 0xFFFF selects export-only mode per Section 5.3 of [RFC9180].

* All KEMs, including DHKEM-based KEMs (x25519, p-256), use the KDF selected by the Hash parameter, both for the HPKE key schedule and for DHKEM's internal operations per [I-D.ietf-hpke-hpke]: HKDF-SHA256 (KDF ID 0x0001) when Hash=sha-256, or TurboSHAKE256 (KDF ID 0x0013) when Hash=turboshake256.

* When Hash=turboshake256, the HPKE implementation MUST conform to the one-stage key schedule defined in [I-D.ietf-hpke-hpke].

ML-KEM-768 key encoding follows [I-D.ietf-lamps-kyber-certificates]. Auth mode requires AuthEncap/AuthDecap, which are defined only for DHKEM-based KEMs. ML-KEM-768 MUST NOT be used with Auth mode. Additional KEMs from the IANA HPKE KEM Identifiers registry MAY be supported following the process defined in Section 10.2.

5.6.3.3. Key Identifier Computation

The id parameter in hpke(...) steps identifies the intended recipient public key. Key identifiers hash the SubjectPublicKeyInfo (SPKI) Distinguished Encoding Rules (DER) encoding rather than raw key octets. This ensures key identifiers are consistent with certificate fingerprint practices and include the algorithm Object Identifier (OID), preventing collisions between keys of different types.

```
spki_der = DER-encode SubjectPublicKeyInfo for pk
           per the KEM's registered SPKI Encoding ({{iana-kem}})
           DER encoding MUST be canonical.

id = Base64( SafeDerive("SAFE-SPKI-v1",
    spki_der, "", 32) ) per {{RFC4648}}
```

The resulting Base64 string is the value of the id parameter (44 characters for the 32-octet output).

5.6.3.4. HPKE Step Secret Derivation

When the step sequence includes one or more hpke(...) steps, the LOCK MUST include a corresponding kemct parameter value for each HPKE step, in the same order as they appear in the step sequence. Encryptors MUST generate a fresh encapsulation per LOCK; reusing a prior encapsulation is prohibited.

For Auth mode, the Decryptor resolves the sender public key as follows:

* If sid is present: match against known sender public keys using the configured Hash.

* If shint is present: filter candidate sender keys by the hint value.

* If neither is present: try all locally known sender keys matching the kem type.

The step secret is derived via HPKE's Export interface:

```
exporter_context = SafeDerive("SAFE-STEP",
    step_token, "", 32)

step_secret = ctx.Export(exporter_context, L=32)
```

where ctx is the HPKE context returned by SetupBaseS/R (or SetupAuthS/R for Auth mode) as described in Section 5.6.3.2, and step_token is the binding form defined in Section 5.6.

When id or sid is omitted from the on-wire token, the Decryptor reconstructs it during trial decryption per Section 5.6.

The KEM binds the shared secret to the recipient key (and for Auth mode, the sender key). The exporter_context binds the step secret to the step token, preventing key-confusion attacks where an attacker substitutes one recipient's encapsulation for another's. The HPKE info parameter "SAFE-v1" binds the key schedule to this protocol, preventing cross-protocol reuse of the same KEM keys from producing valid SAFE step secrets. Suite binding is not needed here because the final KEK derivation commits to encryption_parameters (Section 5.7.1).

Encode form:

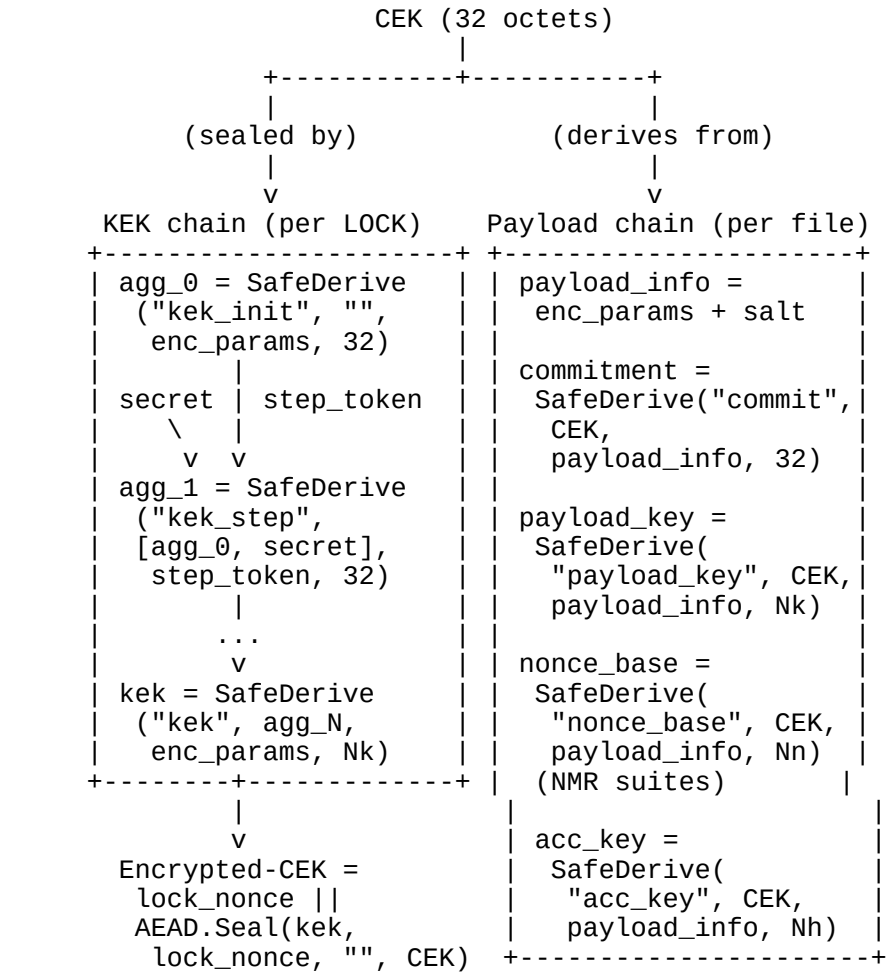
```
Encode("hpke", kem, kemct, id)
Encode("hpke", kem, kemct, id,
       "auth", sid)                    ; auth
```

Binding step_token: Encode("hpke", kem, kemct, id) for Base mode; Encode("hpke", kem, kemct, id, "auth", sid) for Auth mode. Display-only fields (hint, shint) are not included. The id and sid fields are reconstructed per Section 5.6 when omitted on-wire.

5.7. Key Schedules

SAFE uses two applications of SafeDerive (Section 5.4). The KEK derivation produces a LOCK-specific KEK from its ordered step secrets. The payload derivation produces the per-file payload key, commitment, accumulator key, and (for NMR AEADs) nonce base from the CEK, encryption parameters, and per-file salt.

The following diagram shows the two independent chains:



Payload encryption is performed once under the CEK and per-file salt and does not depend on lock structure. Locks are independent wrappers of the same CEK and can be added or removed without touching payload ciphertext.

5.7.1. KEK Schedule

The KEK schedule derives a KEK from an ordered sequence of step secrets using a running aggregator.

Algorithm:

```
agg = SafeDerive("kek_init", "",
          encryption_parameters, 32)

for each (step_token_i, step_secret_i) in order:
    agg = SafeDerive("kek_step",
              [agg, step_secret_i],
              step_token_i, 32)

derived_kek = SafeDerive("kek", agg,
                  encryption_parameters, Nk)
```

Each step folds the aggregator and step secret into ikm (Extract) as an Encode-framed array. The step token is placed in info (Expand). Suite binding enters at kek_init and again at the final kek derivation, where encryption_parameters commits the aggregator to the negotiated AEAD, Block-Size, and Hash.

5.7.1.1. Multi-Step Example

Consider a LOCK block with two steps requiring both a passphrase and a private key:

```
Step: pass(kdf=argon2id, salt=<Base64>)
Step: hpke(kem=p-256, id=<Base64>, kemct=<Base64>)
Encrypted-CEK: <Base64>
```

Evaluation proceeds per Section 5.7.1:

```
agg = SafeDerive("kek_init", "",
          encryption_parameters, 32)

// Step 1: passphrase
step_secret_1  = Argon2id(passphrase, salt)
agg = SafeDerive("kek_step",
          [agg, step_secret_1],
          step_token_1, 32)

// Step 2: HPKE (export-only mode)
ctx            = SetupBaseR(kemct, sk, info="SAFE-v1")
step_secret_2  = ctx.Export(
    exporter_context =
        SafeDerive("SAFE-STEP",
            step_token_2, "", 32),
    L = 32)
agg = SafeDerive("kek_step",
          [agg, step_secret_2],
          step_token_2, 32)

derived_kek = SafeDerive("kek", agg,
                  encryption_parameters, Nk)
```

The derived_kek depends on both factors. Each step is bound to its position via the aggregator chaining through ikm, preventing step reordering.

5.7.2. Sealing Encrypted-CEK

The per-step salt and kemct values MUST be unique per LOCK. Reusing these values with the same credentials produces the same step_secret, weakening KEK uniqueness.

With derived_kek computed per Section 5.7.1, the Encrypted-CEK field is:

```
Encrypted-CEK = lock_nonce || AEAD.Seal(
    key   = derived_kek,
    nonce = lock_nonce,
    aad   = "",
    pt    = CEK )
```

The AEAD algorithm is the one specified in the CONFIG block (or the default aes-256-gcm when CONFIG is absent); the same AEAD is used for both CEK wrapping and payload encryption. Empty AAD is sufficient because derived_kek already binds encryption_parameters (via kek_init and the final kek derivation, Section 5.7.1) and all step_tokens (via kek_step chaining); repeating this binding in the AEAD AAD would be redundant.

Encryptors MUST generate lock_nonce using SafeRandom(Nn, "SAFE-LOCK-NONCE"); each LOCK requires a fresh value.

5.7.3. Payload Schedule

The payload schedule derives the commitment, payload key, accumulator key, and (for NMR AEADs) nonce base from the CEK using SafeDerive. The salt is a 32-octet random value generated using SafeRandom(32, "SAFE-SALT") and stored at the start of the DATA block. It is appended to encryption_parameters in the info input:

```
payload_info = [...encryption_parameters, salt]

commitment  = SafeDerive("commit", CEK,
                  payload_info, 32)
payload_key = SafeDerive("payload_key", CEK,
                  payload_info, Nk)
acc_key     = SafeDerive("acc_key", CEK,
                  payload_info, Nh)
```

For NMR AEADs (NMR=Yes in Section 5.2.1), the payload schedule also derives nonce_base:

```
nonce_base  = SafeDerive("nonce_base", CEK,
                  payload_info, Nn)
```

For non-NMR AEADs, nonce_base is not derived. The accumulator key acc_key is always derived and is used for the snapshot accumulator (Section 5.7.8).

The commitment prefix is always 32 octets for all AEADs. Decryptors MUST verify the commitment before decrypting any block and MUST reject the file if the derived commitment does not match the stored value (see Section 8.12).

Each call independently derives from CEK with distinct labels. Using the wrong AEAD identifier or Block-Size when attempting decryption produces different SafeDerive outputs, causing AEAD verification to fail on every block. The per-file salt ensures that even when the same CEK is reused across files, each file produces unique derived keys. This binding provides implicit integrity assurance that the decryptor is using the correct encryption parameters.

5.7.4. Epoch Key Derivation

When Key-Epoch is present in CONFIG, per-block encryption uses an epoch key derived from the payload key. Key-Epoch is a non-negative integer r; each epoch spans 2^r consecutive blocks.

```
block_key(i):
  if key_epoch is absent:
    return payload_key
  epoch_index = i >> key_epoch
  return SafeDerive("epoch_key", payload_key,
             [I2OSP(epoch_index, 8)], Nk)
```

Blocks 0 through 2^r - 1 share epoch 0, blocks 2^r through 2^(r+1) - 1 share epoch 1, and so on. When Key-Epoch is 0, every block has its own key (epoch size 1).

Since Key-Epoch is included in encryption_parameters when present (Section 5.5), the commitment and all payload schedule outputs change when the epoch configuration changes. Implementations SHOULD cache epoch keys across consecutive blocks in the same epoch.

This construction is inspired by [FLOE], adapted to SAFE's key schedule.

5.7.5. Per-block Nonces

Nonces are unique per block; their size Nn is determined by the AEAD algorithm (see Section 10.1). Encryptors MUST ensure nonce uniqueness within each block_key(i)'s lifetime (Section 5.7.4).

The per-block encryption produces:

```
(ciphertext_i, tag_i) = AEAD.Seal(key   = block_key(i),
                                  nonce = nonce_i,
                                  aad   = aad_i,
                                  pt    = plaintext_block_i)
```

See Section 5.7.4 for block_key(i). AEAD.Seal returns (ciphertext, tag) where tag is the final 16 octets of the combined AEAD output per [RFC5116]. Implementations receiving a single output MUST split at offset len(output) - 16.

For non-NMR AEADs, Encryptors MUST use one of the nonce constructions defined in Section 5.7.6. Each block's nonce and tag are stored:

```
block_metadata_i = nonce_i || tag_i      (Nn + 16 octets)
```

Where nonce_i is Nn random octets and aad_i is defined below. In armored mode, blocks are stored as nonce_i || ciphertext_i || tag_i. In binary mode, the block metadata (nonce + tag) is stored separately from the ciphertext to enable single-seek block access.

For NMR AEADs, per-block nonces are derived deterministically from nonce_base (Section 5.7.3):

```
nonce_i = nonce_base XOR uint64(i)
```

The XOR is applied to the last 8 octets of nonce_base; leading octets are unchanged. Nonces are not stored; only the authentication tag is kept:

```
block_metadata_i = tag_i                 (16 octets)
```

See Section 8.9 for security rationale.

This design enables re-encryption of the payload without re-wrapping the CEK for each recipient, and supports selective editing of individual blocks. For non-NMR AEADs, per-block random nonces allow CEK reuse across payload revisions while maintaining AEAD security.

5.7.6. Nonce Constructions

Two nonce constructions are defined for non-NMR block encryption. Both produce Nn-octet nonces suitable for use with the configured AEAD. Decryptors read the stored nonce from each block and do not need to know which construction was used.

5.7.6.1. Base-XOR Construction

```
base    = SafeRandom(Nn, "SAFE-NONCE")
nonce_i = base XOR uint64(i)
```

The XOR is applied to the last 8 octets of base; leading octets are unchanged. Within a single file, the XOR with distinct block indices guarantees nonce uniqueness.

This construction is one-pass and supports parallel block encryption.

5.7.6.2. Plaintext-Bound Construction

Encryptors SHOULD incorporate plaintext into per-block nonce derivation for SIV-like nonce-misuse resistance when RNG state duplication is a concern (e.g., VM snapshots, process forks without reseed). This requires two passes over each block (hash then encrypt).

The construction proceeds in two steps:

```
;; Pass 1: hash the plaintext block
pt_hash_i = SafeDerive("SAFE-NONCE",
    plaintext_i, encryption_parameters, 32)

;; Pass 2: derive the nonce
nonce_ctx = Encode("SAFE-NONCE",
    I2OSP(i, 8), pt_hash_i)
nonce_i   = SafeDerive("nonce",
    [SafeRandom(Nn, "SAFE-NONCE"),
     payload_key],
    [...encryption_parameters, nonce_ctx],
    Nn)
```

Pass 1 commits to the plaintext via a SafeDerive hash bound to the suite. Pass 2 mixes fresh randomness, the payload key, and the plaintext hash into the final nonce. Under RNG state duplication, this construction produces different nonces when plaintext differs, limiting exposure to leaking equality of identical blocks at the same index.

5.7.7. Block Rewrite Rules

When re-encrypting a modified block, the procedure depends on the AEAD's nonce-misuse resistance property.

For non-NMR AEADs (stored nonce required):

```
Rewriting block i:
  old_tag  = stored tag_i
  nonce_i  = SafeRandom(Nn, "SAFE-NONCE")
  (ct, new_tag) = AEAD.Seal(block_key(i), nonce_i,
                      data_aad(i, is_final),
                      new_plaintext)
  Update stored nonce_i and tag_i
  Update accumulator ({{snapshot-accumulator}})
```

For NMR AEADs (derived nonce; Key-Epoch MUST be absent):

```
Rewriting block i:
  old_tag  = stored tag_i
  nonce_i  = nonce_base XOR uint64(i)
  (ct, new_tag) = AEAD.Seal(block_key(i), nonce_i,
                      data_aad(i, is_final),
                      new_plaintext)
  Update stored tag_i
  Update accumulator ({{snapshot-accumulator}})
```

NMR rewrites reuse the derived nonce for that block index. Because the AEAD is nonce-misuse resistant, this degrades to deterministic encryption: identical plaintext at the same index produces identical ciphertext, leaking equality but not content. See Section 8.9.2.

5.7.8. Snapshot Accumulator

The snapshot accumulator binds all block authentication tags into a single value, providing file-level integrity without decrypting every block. It uses SafeDerive and XOR; no additional primitives are needed.

For each block i with authentication tag tag_i, the per-block contribution is:

```
contrib_i = SafeDerive("acc_contrib", acc_key,
                [uint64(i), tag_i], Nh)
```

The accumulator is the XOR of all contributions:

```
accumulator = contrib_0 XOR contrib_1
              XOR ... XOR contrib_{N-1}
```

XOR is commutative and associative: blocks may be processed in any order, and a single contribution can be replaced without reprocessing other blocks. The result is Nh octets (32 for all registered Hash algorithms).

When re-encrypting block i with new tag new_tag_i (replacing old tag old_tag_i):

```
old_c = SafeDerive("acc_contrib", acc_key,
            [uint64(i), old_tag_i], Nh)
new_c = SafeDerive("acc_contrib", acc_key,
            [uint64(i), new_tag_i], Nh)
accumulator = accumulator XOR old_c XOR new_c
```

This update costs two SafeDerive calls and two XOR operations, independent of the total block count.

Verification:

1. Derive acc_key from the CEK and salt (Section 5.7.3).

2. Read all stored tags from block metadata.

3. Compute the accumulator by XOR-combining per-block contributions.

4. Compare the result to the stored accumulator. If they differ, at least one block has been modified, replaced, or is missing.

The accumulator is a PRF-based XOR-checksum over the set of (index, tag) pairs. Each contribution is a PRF output under acc_key with unique inputs; an adversary who does not know acc_key cannot predict any individual contribution and therefore cannot forge a valid accumulator value. Decryptors MUST verify the accumulator before per-block decryption when the full set of block tags is available. Streaming Decryptors that process blocks incrementally MUST verify the accumulator once all blocks have been processed.

5.7.9. Block AAD

For block index i in the range 0 <= i < N, the associated data binds each block to its position and indicates whether it is the final block:

```
data_aad(i, is_final) = Encode("SAFE-DATA",
    I2OSP(i, 8), I2OSP(is_final, 1))
```

Where I2OSP(i, 8) is the block index in network byte order and is_final is 0x01 for the last block (index N-1) and 0x00 for all preceding blocks. Encode provides prefix-free framing via lp16 (Section 4.1). No KDF call is needed: payload_key already depends on encryption_parameters (Section 5.7.3), and the AEAD tag authenticates the AAD under that key.

The is_final flag provides truncation detection: if a Decryptor decrypts block i with is_final=0 and no subsequent block exists, truncation has occurred. Decryptors MUST abort when truncation is detected (is_final=0 with no successor block). For random-access reads, Decryptors SHOULD verify that the block count N in the metadata matches the actual number of blocks. The flag also prevents extension attacks: appending blocks after a block marked is_final=1 will fail AEAD verification.

For streaming writes where the total block count is unknown, Encryptors buffer the last block until more data arrives or the stream ends. All emitted blocks use is_final=0; only when the stream closes does the Encryptor encrypt the final block with is_final=1.

Encryptors MUST ensure block indices remain below 2^64. Encryptors SHOULD limit i to at most 2^48 to avoid Base64 strings exceeding typical filesystem or object store limits; this is a practical recommendation, not a protocol limit. Decryptors MUST reject block indices i where i >= 2^64.

6. File Layout

A SAFE encoding MUST consist of an optional config header (Section 6.1), followed by one or more LOCK blocks (Section 6.2), followed by exactly one DATA block (Section 6.4). Blocks MUST appear in this order; Decryptors MUST reject encodings that violate it. There is no version marker in the fences. Multiple LOCK blocks provide multi-recipient encryption; the DATA block is shared.

6.1. SAFE CONFIG

The config header may be omitted when all defaults apply. When present, it lists only non-default parameters. The config does not need to be parsed before attempting decryption if the Decryptor already knows or can infer the default parameters.

The header is:

```
-----BEGIN SAFE CONFIG-----
AEAD: aes-256-gcm | chacha20-poly1305 | ...
Block-Size: 16384 | 65536
Hash: sha-256 | turboshake256
Key-Epoch: <non-negative integer>
Lock-Encoding: armored | readable
Data-Encoding: armored | binary | binary-linear
-----END SAFE CONFIG-----
```

All CONFIG fields are optional. Omitted fields use default values (Section 5.1): AEAD defaults to aes-256-gcm, Block-Size to 65536, Hash to sha-256, Lock-Encoding to armored, and Data-Encoding to armored. Hash selects the hash function used throughout the protocol (Section 5.4); any identifier in the SAFE KDF Identifiers registry (Section 10.3) is valid. Lock-Encoding specifies the LOCK block representation (Section 6.2). Data-Encoding specifies the payload format (Section 6.3): "armored" uses Base64 within DATA fence markers, "binary" uses block-aligned raw binary after the last LOCK, and "binary-linear" uses sequential raw binary after the last LOCK. The encoding fields are presentational choices that do not affect cryptographic operations.

Key-Epoch enables per-epoch key derivation (Section 5.7.4). When absent, all blocks use the payload key directly. When present, it is a non-negative integer r where r MUST be less than 64; Decryptors MUST reject files where Key-Epoch is 64 or greater. Block i uses an epoch key derived from the payload key with epoch index i >> r, giving each epoch 2^r blocks and limiting AEAD invocations per key (Section 8.13). Key-Epoch MUST NOT be present for NMR AEADs; Decryptors MUST reject files where Key-Epoch is present and the AEAD has NMR=Yes in Section 5.2.1. Encryptors using chacha20-poly1305 MUST include Key-Epoch; Section 4 of [RFC8439] requires nonce uniqueness per key, and epoch keys limit the collision domain to rewrites within each epoch. Encryptors using aes-256-gcm SHOULD include Key-Epoch for rewritable files to stay within the per-key invocation limits of [NIST-SP-800-38D]; see Section 8.13.3 for budget calculations. Key-Epoch: 0 is RECOMMENDED for maximum security; Key-Epoch: 5 is a practical alternative when key-derivation overhead matters.

NMR AEADs (NMR=Yes in Section 5.2.1) implicitly use derived nonces: per-block nonces are computed from the key schedule and block index, and the format nonce length is 0 (nonces are not serialized). Non-NMR AEADs use stored random nonces. See Section 5.7.5 for details.

Implementations MUST support Lock-Encoding: armored and Data-Encoding: armored. Support for Lock-Encoding: readable and Data-Encoding: binary or binary-linear is OPTIONAL.

A conforming SAFE file MAY omit the SAFE CONFIG block entirely; parsers MUST treat this identically to a CONFIG block with all defaults. When CONFIG is present, it MAY contain any subset of fields. Implementations MUST construct encryption_parameters using defaults for any omitted fields.

Field names within the SAFE CONFIG are case-sensitive. Encryptors MUST NOT include duplicate field names; Decryptors MUST reject SAFE config blocks containing duplicate fields. Decryptors MUST reject SAFE CONFIG blocks containing field names not listed in the SAFE Config Options registry (Section 10.5).

The order of fields within a CONFIG block is not significant. Encryptors MAY emit fields in any order; Decryptors MUST accept fields in any order.

All header lines MUST contain only ASCII characters (octets 0x20-0x7E and LF). Encryptors MUST NOT include non-ASCII characters in field names or values. Decryptors MUST reject SAFE CONFIG blocks containing non-ASCII octets or malformed UTF-8 sequences.

Implementations SHOULD bound SAFE CONFIG size; Decryptors MAY reject SAFE CONFIG headers exceeding 64 KiB.

Field values MAY wrap across multiple lines using the same rules as LOCK blocks (Section 6.2.1.2): continuation lines MUST be indented with at least two spaces, and Decryptors MUST concatenate continuation lines (stripping leading whitespace) before processing.

6.1.1. Block-Size Selection

SAFE defines two Block-Size values: 16384 and 65536. Only these values are valid; Decryptors MUST reject any other Block-Size. Adding new Block-Size values requires Standards Action.

65536 (default): Larger blocks amortize per-block AEAD overhead and reduce I/O syscalls, yielding higher throughput for sequential encrypt and decrypt. This value is appropriate when the payload will be decrypted in full or streamed sequentially.

16384: Smaller blocks reduce the cost of partial updates. Re-encrypting a modified block requires reading and rewriting only that block; at 16384 octets the I/O cost per edit is one quarter of the default. Applications that perform random-access writes to encrypted data SHOULD use Block-Size 16384.

Both values align to hardware page boundaries. Block-Size 16384 is one page on systems with 16 KiB pages (e.g., Apple Silicon) and four pages on systems with 4 KiB pages (e.g., x86-64). Block-Size 65536 is four pages and sixteen pages, respectively. This alignment avoids page-crossing penalties with direct I/O or memory-mapped access.

The Encryptor selects Block-Size at encryption time. The value is recorded in the SAFE CONFIG and applies to all recipients. There is no mechanism to change Block-Size after encryption without re-encrypting the entire payload.

6.2. SAFE LOCK

A LOCK block defines the unlock steps for a single recipient and carries the artifacts needed to recover the CEK. Each LOCK contains one or more steps and exactly one Encrypted-CEK.

Steps are evaluated in the order they appear. Step-specific inputs are carried as parameters (e.g., salt= for pass, kemct= for hpke). See Section 5.6.2 and Section 5.6.3 for step-specific requirements.

The Encrypted-CEK is the concatenation of lock_nonce and the AEAD ciphertext of the CEK under the derived KEK with empty associated data (Section 5.7.2). The lock_nonce length is the AEAD’s nonce size (Nn) as specified in Section 10.1. Encryptors MUST generate a fresh lock_nonce per LOCK using SafeRandom(Nn, "SAFE-LOCK-NONCE") (Section 5.4.2). Decryptors MUST reject Encrypted-CEK values that do not decode to exactly Nn + 32 + 16 octets (lock_nonce, CEK ciphertext, AEAD tag).

Decryptors MUST skip LOCK blocks containing unknown KEM identifiers or unknown step types, and attempt other LOCKs (if available). Decryptors MUST reject step tokens containing parameter names not defined for the step type in its registration.

Implementations SHOULD bound the number of LOCK blocks; Decryptors MAY reject files containing more than 1024 LOCK blocks to prevent resource exhaustion.

Two Lock-Encoding values are defined: readable (text) and armored (binary). Both produce the same binding step_tokens for the KEK schedule (Section 5.6).

6.2.1. Readable Format

The readable format uses text step tokens and colon-delimited fields:

```
-----BEGIN SAFE LOCK-----
Step: pass(kdf=argon2id, salt=<Base64>)
Step: hpke(kem=x25519, id=<Base64>, kemct=<Base64>)
Encrypted-CEK: <Base64>
-----END SAFE LOCK-----
```

Field names are case-sensitive. Encryptors MUST NOT include fields other than Step and Encrypted-CEK. Encryptors MUST include at least one Step line and exactly one Encrypted-CEK line. Decryptors MUST reject LOCK blocks containing multiple Encrypted-CEK lines or unknown field names. Optional whitespace (OWS) after colons and commas is permitted for readability.

6.2.1.1. Step Syntax

Each Step line declares a single cryptographic step. Multiple steps form an ordered sequence with AND semantics: all steps MUST be satisfied to derive the KEK. The syntax follows this ABNF, which applies after Decryptors perform line unfolding (concatenating continuation lines and stripping leading whitespace per Section 6.2.1.2):

```
step-line       = "Step:" OWS step-token LF
step-token      = step-name "(" step-params ")"
step-name       = 1*( ALPHA / DIGIT / "-" )
step-params     = param *( "," OWS param )
param           = param-name "=" param-value
param-name      = 1*( ALPHA / DIGIT / "-" )
param-value     = 1*PCHAR
PCHAR           = %x21-28 / %x2A-2B / %x2D-7E
                    ; VCHAR except SP, ")", ","
OWS             = *( SP / HTAB )
```

The binding step_token used in the KEK schedule (Section 5.7.1) is derived per Section 5.6: extract the binding fields and encode them with Encode().

Each Step line contains exactly one step token. LOCK blocks with multiple steps use multiple Step lines. Step-specific inputs are carried as step token parameters (e.g., salt= for pass, kemct= for hpke).

The param-value production forbids spaces (SP, 0x20) and tabs (HTAB, 0x09). Percent-encoding is not supported; all parameter values MUST be literal UTF-8 printable characters excluding whitespace.

Encryptors MUST emit parameters in the order specified by the step definition. Decryptors MUST reject step tokens whose parameters are not in the specified order. Decryptors MUST reject step tokens containing duplicate parameter names within a single step.

See Section 5.7 for how step secrets are combined to derive the KEK. See Section 5.6.3 for the HPKE step format and Section 5.6.2 for the passphrase step format.

6.2.1.2. Line Wrapping

Field values MAY wrap across multiple lines. Continuation lines MUST be indented with at least two spaces. Decryptors MUST unfold wrapped values by concatenating continuation lines and stripping leading whitespace.

* Step tokens MAY wrap at parameter boundaries (after commas). Encryptors SHOULD insert a space after the comma at each wrap point. Continuation lines use 4-space indent.

* Encrypted-CEK values use 2-space indent for continuation lines.

Encryptors SHOULD wrap lines at 64 characters. Decryptors MUST accept any line length.

A field value extends from immediately after the colon (and any following whitespace) until one of:

1. A line starting with "Step:" (unindented)

2. A line starting with "Encrypted-CEK:" (unindented)

3. A fence line "-----END SAFE LOCK-----"

Trailing whitespace on individual lines SHOULD be avoided; Decryptors MUST strip trailing spaces and tabs from each line before concatenation.

Example with wrapped HPKE step token:

```
Step: hpke(kem=ml-kem-768, hint=4217,
    kemct=bWxrZW03NjhrZW1jaXBoZXJ0ZXh0
    ZXh0cmVtZWx5bG9uZ2JhY2U2NGVuY29kZW
    RkYXRhYXBwcm94aW1hdGVseTEwODhvY3Rl
    dHNmb3JwcXNlY3VyaXR5)
```

Decryptors parse the text, extract field values, and produce the step Encode form for binding.

6.2.2. Armored Format

When Lock-Encoding is armored, the SAFE LOCK block contains a single Base64 value. The value is the Base64 encoding of the Encode-serialized LOCK:

```
armored_lock = Base64(
    Encode(step_1, step_2, ..., encrypted_cek))
```

Each step_i is the binding Encode form (excluding display-only fields) as defined in the step's specification. The encrypted_cek is the raw Encrypted-CEK octets (lock_nonce || ciphertext).

Decryptors decode the Base64, split the outer Encode into lp16-framed elements. The last element is the Encrypted-CEK; preceding elements are step tokens. Each step token is itself an Encode whose first element is the step name.

Encryptors MUST use Base64 with padding per [RFC4648]. The Base64 value MAY wrap across multiple lines; continuation lines MUST be indented with at least two spaces. Decryptors MUST concatenate continuation lines (stripping leading whitespace) before decoding.

6.2.3. LOCK Selection

A decryptor determines candidate LOCK blocks without touching the SAFE data block.

6.2.3.1. Candidate Selection

For each LOCK block, determine candidacy as follows:

1. Parse the Step tokens. If any token references an unsupported step type, the LOCK is not a candidate.

2. For each hpke(...) step, determine candidate keys based on mode:

   Identified (id present): Compute the key identifier for locally available public keys using SafeDerive("SAFE-SPKI-v1", spki_der, "", 32) with the configured Hash (default: sha-256). Keys whose identifier matches the id parameter are candidates.

   Hinted (hint present): Look up keys in local storage associated with this hint value. Keys with matching hint are candidates.

   Anonymous (neither id nor hint): All local keys matching the kem type are candidates.

   If no local recipient keys are candidates, the LOCK is not a candidate.

   For auth-mode hpke(...) steps (sid or shint present), also determine candidate sender keys:

   Identified (sid with Base64 value): Keys whose identifier matches sid are candidates.

Hinted (shint present): Keys with matching shint are candidates.

Anonymous (sid=anon): All locally known sender keys matching the kem type are candidates.

If no sender keys are candidates for an auth-mode step, the LOCK is not a candidate.

3. For pass(...) steps: the LOCK is a candidate if the implementation supports passphrase entry.

4. For other registered steps: the LOCK is a candidate if the implementation supports them and local policy permits.

6.2.3.2. Attempt Order

Among remaining candidates, Decryptors SHOULD attempt LOCKs in order of confidence:

1. LOCKs where all hpke steps are identified or hinted; the decryptor has confirmed it holds matching keys.

2. LOCKs with anonymous hpke steps; requires trial decryption across all keys of the matching KEM type.

3. LOCKs with pass steps; may require user interaction, so defer until key-only LOCKs are exhausted.

Encryptors MAY include multiple pass(...)-only LOCK blocks if they use different KDF variants (e.g., one pass(kdf=argon2id, ...) and one pass(kdf=pbkdf2, ...) for the same passphrase). This enables interoperability between implementations with different passphrase KDF support. Encryptors MUST NOT include duplicate pass(...)-only LOCKs with the same KDF variant. Decryptors MUST stop at the first successful CEK recovery. Decryptors MAY attempt multiple candidates in parallel.

6.2.3.3. Trial Decryption

For hinted or anonymous step sequences, Decryptors iterate through candidate key combinations. For composable step sequences (multiple hpke steps with AND semantics), trial decryption MUST consider the combinatorial product of candidates for each step. For auth-mode steps, the product includes sender key candidates in addition to recipient key candidates. For each combination:

1. Establish an HPKE context for each hpke(...) step via SetupBaseR (or SetupAuthR with the candidate sender key for auth-mode steps) and call Export per Section 5.6.3.4

2. Derive the KEK by aggregating step secrets per Section 5.7.1

3. Attempt to open Encrypted-CEK with the derived KEK

A candidate succeeds when AEAD tag verification passes on Encrypted-CEK. If the KEK is wrong, the tag will not verify.

6.3. Data Encoding

The Data-Encoding CONFIG field specifies how the payload is represented. SAFE defines two payload layouts: linear concatenates encrypted blocks sequentially, while aligned adds padding for block-aligned random access. Three encoding values are defined:

Value	Layout	Representation
armored	linear	Base64 within fence markers
binary	aligned	Raw binary, block-aligned
binary-linear	linear	Raw binary, sequential

Table 10

The default is "armored" when Data-Encoding is omitted.

6.3.1. Armored Encoding

Armored encoding wraps the linear layout (Section 6.4.1) in Base64 with fence markers:

```
-----BEGIN SAFE DATA-----
<Base64: linear_payload>
-----END SAFE DATA-----
```

The Base64 string MAY be wrapped across multiple lines for readability. When wrapped, each line break MUST be a single LF character. For length calculations and random-access arithmetic, Decryptors MUST first remove all line breaks (LF and CRLF) and CR octets (0x0D), then strip trailing whitespace from the result. The normalized string length determines padding and block offset computations. Decryptors MUST ignore these characters during Base64 decoding and concatenate all lines before decoding.

Encryptors SHOULD wrap Base64 lines at 64 characters. Decryptors MUST accept any line length.

Implementations MAY enforce an upper bound on payload size to prevent over-allocation; Decryptors MAY reject payloads exceeding 64 TiB of ciphertext.

Armored data arithmetic (computing block count, byte-to-Base64 offsets, and per-block decryption) is detailed in Appendix D.

6.3.2. Binary Encoding

Binary encoding omits fence markers. Binary data begins at the octet immediately following the LF (0x0A) that terminates the last -----END SAFE ..------ line (typically the final LOCK block). Binary data ends at EOF.

Two variants exist:

Data-Encoding: binary Uses the aligned layout (Section 6.4.2). Optimized for random access to large files via memory-mapped I/O or O_DIRECT.

Data-Encoding: binary-linear Uses the linear layout (Section 6.4.1). Suitable for streaming or simple implementations that do not require block-aligned random access.

Implementations that do not support binary encoding MUST fail when encountering Data-Encoding: binary or Data-Encoding: binary-linear, consistent with the handling of unknown AEAD or Hash values.

Encryptors SHOULD prefer armored encoding for maximum compatibility. Binary encoding is intended for performance-critical applications or programmatic access where human readability is not required.

6.4. Payload Layouts

SAFE defines two payload layouts that describe how encrypted blocks are structured.

6.4.1. Linear Layout

The linear layout concatenates encrypted blocks sequentially with no padding or alignment constraints:

[salt] [commitment] [accumulator] [eb_0] [eb_1] ... [eb_{N-1}]

For non-NMR AEADs:

eb_i = nonce_i || ciphertext_i || tag_i (Nn + len(pt_i) + 16)

For NMR AEADs:

eb_i = ciphertext_i || tag_i (len(pt_i) + 16)

The payload begins with a 32-octet salt, a 32-octet commitment (both derived per Section 5.7.3), and the Nh-octet accumulator (Section 5.7.8), followed by encrypted blocks.

For non-NMR AEADs, each encrypted block (eb_i) is Nn + len(plaintext_i) + 16 octets. For NMR AEADs, each encrypted block is len(plaintext_i) + 16 octets (nonces are derived, not stored).

Decryptors MUST verify the commitment before decryption and MUST reject the file if the derived commitment does not match the stored value. See Section 8.12.

All blocks except the final block contain Block-Size octets of plaintext. The final block MAY be smaller. For non-NMR AEADs, each encrypted block consists of a nonce (Nn octets), ciphertext (same length as the plaintext), and authentication tag (16 octets). For NMR AEADs, each encrypted block consists of ciphertext and authentication tag only.

Zero-length plaintexts are allowed. A zero-length plaintext produces N = 1, L_final = 0. For non-NMR AEADs with Nn = 12, the minimum payload is 124 octets (32-octet salt + 32-octet commitment + 32-octet accumulator + 28-octet encrypted block). For NMR AEADs, the minimum is 112 octets (32 + 32 + 32 + 16).

Decryptors MUST reject payloads with unexpected structure: incorrect commitment length, missing or invalid accumulator, or block boundaries that do not align with expected sizes.

In binary-linear encoding the block count N is not stored explicitly. Streaming readers determine N by reading blocks sequentially until EOF: each non-final block is exactly C octets (Nn + B + 16 for non-NMR AEADs; B + 16 for NMR AEADs). The final block is smaller than C.

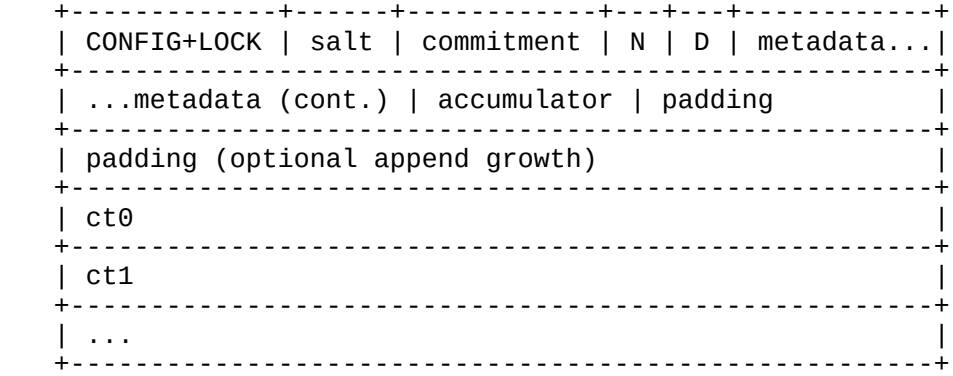
6.4.2. Aligned Layout

The aligned layout structures the file so that every ciphertext block begins at an offset that is an exact multiple of the Block-Size B. This alignment enables efficient memory-mapped I/O and O_DIRECT access, since the operating system can read any block without copying data across page boundaries.

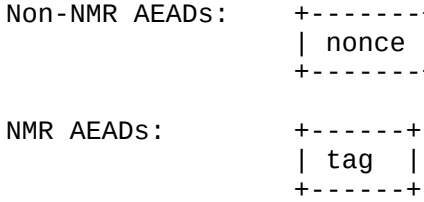
The file begins with a header section containing the text headers (CONFIG and LOCK blocks) followed by binary fields: salt, commitment, block count N, first ciphertext index D, per-block metadata (nonces and tags), and the snapshot accumulator. The header is padded with zeros to a block boundary, followed by zero or more padding blocks for append growth, then ciphertext blocks.

Let B denote the Block-Size in octets (16384 or 65536). Let Nn be the nonce size (12 for AES-GCM and ChaCha20, 32 for AEGIS-256 and AEGIS-256X2). N is the block count (uint32), and D is the first ciphertext block index (uint32). Let meta_len be the per-block metadata size: Nn + 16 for non-NMR AEADs, or 16 for NMR AEADs.

Each row below represents one Block-Size:



Per-block metadata entry:



The binary portion immediately follows the text headers:

* salt (32 octets)

* commitment (32 octets)

* N: block count (uint32, big-endian)

* D: first ciphertext block index (uint32, big-endian)

* metadata: N entries, each meta_len octets

* accumulator: Nh octets (Section 5.7.8)

* padding to block boundary, then zero or more padding blocks

* ciphertext blocks starting at offset (D × B)

The uint32 block count limits aligned-layout files to 2^32 - 1 blocks. Files exceeding this count MUST use linear layout.

6.4.2.1. Writing

To write an aligned-layout file, the Encryptor computes D as follows. Let H be the total header octet count: all CONFIG and LOCK text (including fence markers and newlines), plus 32 (salt) + 32 (commitment) + 4 (N) + 4 (D) + N * meta_len (metadata) + Nh (accumulator). Then:

D = ceil(H / B)

Padding blocks for append growth MAY be added by increasing D beyond the minimum value. The encryptor writes the header, pads to D * B octets, then writes ciphertext blocks at offsets D * B, (D+1) * B, and so on.

The aligned layout requires a seekable output. For streaming writes where N is unknown at the start, the Encryptor estimates a maximum block count, computes D from that estimate, reserves space for D, N, and metadata in the header, then seeks back to fill the actual values once the stream closes.

6.4.2.2. Reading

To read an aligned-layout file:

1. Parse CONFIG and LOCK text to determine AEAD and Block-Size.

2. Read the 32-octet salt, 32-octet commitment, N, and D.

3. Read N metadata entries, each meta_len octets.

4. Read the Nh-octet accumulator.

5. Verify the accumulator (Section 5.7.8).

6. Ciphertext block i is at offset (D + i) × B.

To read only block i: for NMR AEADs, compute nonce_i from nonce_base and the block index; for non-NMR AEADs, read the nonce from the metadata entry. Read the tag from the metadata entry in both cases. Then read B octets (or fewer for the final block) at offset (D + i) x B. The final block's ciphertext length is file_size - (D + N - 1) * B octets.

7. Compatibility and Migration

7.1. Handling Unknown Elements

Decryptors processing SAFE-encoded data MUST:

* Fail if they encounter an unrecognized or unimplemented value for any CONFIG field (AEAD, Hash, Data-Encoding, Lock-Encoding, Block-Size). Implementations MUST NOT silently ignore CONFIG values they do not support.

* Reject Block-Size values other than 16384 or 65536.

* Skip LOCKs containing unknown field names, KEM identifiers, or step types and attempt other LOCKs (if available).

* Fail if a CONFIG block contains field names not listed in the SAFE Config Options registry (Section 10.5).

* Fail if the payload has unexpected structure (wrong commitment length, trailing octets, misaligned block boundaries).

* Skip unknown block types if the IANA SAFE Block Types registry (Section 10.6) marks them as Ignorable; otherwise fail.

7.2. Versioning

This document defines SAFE version 1, identified by fence markers ("-----BEGIN SAFE CONFIG-----", etc.). Future incompatible versions would use different fence markers or a new media type. New features SHOULD be added through IANA registries rather than format version changes.

7.3. Extension Points

SAFE provides IANA registries for AEADs (Section 10.1), KEMs (Section 10.2), step types (Section 10.4), and block types (Section 10.6).

Unknown block types are critical by default: Decryptors MUST fail if they encounter an unrecognized block. The IANA SAFE Block Types registry (Section 10.6) MAY mark specific block types as Ignorable, enabling forward-compatible optional extensions such as metadata or signatures that older implementations can safely skip.

7.4. Application Profiles

This section is informative. It describes three parameter combinations for common deployment scenarios. These profiles compose the CONFIG fields defined in Section 6.1; they do not introduce new protocol elements.

7.4.1. Objects

Applications that prioritize text-safe output and maximum interoperability SHOULD use the default parameters (Section 5.1). No CONFIG block is required. The resulting SAFE object is entirely printable ASCII and can be embedded in email, JSON, YAML, or version-controlled files. AES-256-GCM is the default AEAD (Section 5.1).

7.4.2. Streaming

Applications that process data sequentially at high throughput SHOULD consider:

```
-----BEGIN SAFE CONFIG-----
AEAD: aegis-256
Data-Encoding: binary-linear
-----END SAFE CONFIG-----
```

AEGIS-256 offers high throughput and a 32-octet nonce that simplifies nonce management. Combined with binary-linear encoding (Section 6.4.1), this yields minimal framing overhead and sequential I/O without alignment padding. Encryptors using this profile SHOULD apply the hedged nonce construction (Section 5.4.2.2) or plaintext-bound nonce construction (Section 5.7.6.2) per Section 8.9.

Encryptors targeting broad interoperability SHOULD verify recipient support before selecting this profile.

7.4.3. Edit

Applications that perform random-access reads and writes on encrypted data SHOULD consider:

```
-----BEGIN SAFE CONFIG-----
AEAD: aes-256-gcm-siv
Block-Size: 16384
Data-Encoding: binary
-----END SAFE CONFIG-----
```

AES-256-GCM-SIV is nonce-misuse resistant (Section 5.2.1), so per-block nonces are derived rather than stored (Section 5.7.5). This reduces per-block metadata from Nn + 16 octets to 16 octets. Block-Size 16384 aligns each block to a single page on 16 KiB-page systems, minimizing page faults per edit (Section 6.1.1). Binary aligned encoding (Section 6.4.2) enables O(1) random access to any block via memory-mapped I/O.

Re-encrypting a modified block reuses the derived nonce for that block index. Because AES-256-GCM-SIV is nonce-misuse resistant, this degrades to deterministic encryption for unchanged blocks rather than catastrophic nonce reuse (Section 8.9.2).

7.4.4. FIPS Edit

Applications that require FIPS 140-validated algorithms for random-access editing SHOULD consider:

```
-----BEGIN SAFE CONFIG-----
AEAD: aes-256-gcm
Data-Encoding: binary
Key-Epoch: 5
-----END SAFE CONFIG-----
```

AES-256-GCM is FIPS 140-validated; ChaCha20-Poly1305, AES-256-GCM-SIV, AEGIS-256, and AEGIS-256X2 are not currently covered by a FIPS 140 validation program. Key-Epoch: 5 limits each epoch key to 32 blocks, giving 2^27 - 1 (over 134 million) rewrites per block before exhausting the per-key nonce budget (Section 8.13.3). This is sufficient for virtually all editing workloads while adding negligible key-derivation overhead. The default Block-Size of 65536 reduces the number of epoch key derivations relative to smaller block sizes.

Unlike the Edit profile (Section 7.4.3), this profile stores random per-block nonces (Nn + 16 octets per block vs. 16 octets for GCM-SIV). Applications that can tolerate block-level equality leakage on rewrite SHOULD prefer the Edit profile for its lower per-block overhead.

8. Security Considerations

SAFE provides:

Confidentiality: Indistinguishability under adaptive chosen-ciphertext attack (IND-CCA2) security for the payload, assuming IND-CCA2-secure AEAD. This follows from the standard reduction: SafeDerive is a PRF, so payload_key is indistinguishable from random; under a random key, the registered AEADs provide IND-CCA2 per their specifications.

Authentication: Each LOCK's Encrypted-CEK is authenticated via AEAD under derived_kek, which binds step tokens and step secrets through the KEK schedule. Block AEAD with index-bound AAD (Section 8.3) prevents reordering, modification, and splicing. The snapshot accumulator (Section 5.7.8) provides file-level integrity over all block tags.

Binding: The KEK schedule binds encryption_parameters at initialization and final derivation (Section 5.7.1). Step tokens and per-step secrets are folded into the aggregator via sequential SafeDerive chaining. Payload keys inherit suite binding from their own SafeDerive calls (Section 5.7.3).

SAFE does not provide:

Encryptor authentication (Base mode): Without a sender parameter (sid or shint), any party with recipient public keys can create SAFE-encoded data. See Section 8.2 for Auth mode authentication properties.

Forward secrecy: CEK compromise exposes all recipients' copies. This is inherent to stored-object encryption, which has no interactive key exchange.

Unlinkability: Key identifiers enable linking SAFE-encoded data to the same recipient. See Section 8.6 for privacy modes.

SAFE assumes secure key storage, side-channel resistant implementations, and trusted cryptographic primitives. A functioning CSPRNG is REQUIRED. See Section 8.9 for defenses against RNG weakness or state duplication.

8.1. Threat Model

SAFE defends against:

Compromised storage provider (confidentiality): An adversary with read access to stored SAFE-encoded data cannot decrypt without valid credentials (passphrase or private key) for at least one LOCK. The adversary can observe approximate file size, recipient count, and key identifier linkability. SAFE detects block corruption (AEAD failure) and truncation (is_final mismatch), but does not detect LOCK removal, LOCK addition, or whole-file replacement by an adversary with write access.

SAFE does not defend against:

Compromised recipient: If a recipient's credentials (passphrase or private key) are compromised, the adversary can decrypt the payload. All recipients share the same CEK; compromise of one recipient's KEK does not expose other recipients' KEKs, but does expose the shared CEK and payload. The weakest LOCK determines the effective security of the entire file: an attacker who can satisfy any single LOCK recovers the CEK.

Active attacker with key compromise: If an attacker compromises a recipient's private key and can modify files, they can create valid SAFE-encoded data for that recipient (in Base mode). Auth mode (Section 5.6.3.1) mitigates this for steps where the attacker does not also hold the sender's private key; see Section 8.2.

Side-channel attacks: SAFE assumes implementations do not intentionally leak secrets. Timing attacks on Argon2id, HPKE, or AEAD operations are out of scope for this document.

Malicious Encryptor: Any party with a recipient's public key can create valid SAFE-encoded data for that recipient. SAFE does not constrain what an Encryptor can encrypt or for whom. Applications MUST validate decrypted content independently of the encryption envelope.

8.2. Sender Authentication Properties

When sid or shint is present in an hpke(...) step, SAFE uses HPKE Auth mode (mode_auth, [RFC9180]). Auth mode defends against forgery by parties who do not hold the sender's private key: a decryptor who successfully processes an auth-mode step is assured that the encapsulation was produced by a holder of skS. This closes the "encryptor authentication" gap identified above for Base mode, within the following limits:

Non-repudiation: Auth mode authenticates the sender only to the holder of the recipient's private key. The recipient cannot prove to a third party that the sender created the SAFE object. Applications requiring non-repudiation MUST use external signatures.

Sender identity confidentiality: The sid parameter (when a Base64 value) reveals the sender's key identifier to any observer. shint narrows the sender's identity. Anonymous Auth mode (sid=anon) avoids explicit sender identification, at the cost of trial decryption across all candidate sender keys.

Sender key trust: SAFE does not define a trust model for sender public keys. Decryptors MUST independently verify that a sender's public key is authentic (e.g., via a certificate, trust on first use (TOFU), or out-of-band verification) before relying on auth-mode authentication.

8.3. Integrity and Authenticity

The KEK schedule binds encryption_parameters at initialization and final derivation, with step tokens and per-step secrets folded via sequential SafeDerive chaining. The payload schedule binds payload_key to encryption_parameters independently, tying block encryption to the negotiated AEAD and Block-Size. Payload AEAD authenticates each block with index-bound AAD, preventing reordering and cross-file splicing. SAFE detects truncation and extension at block boundaries via is_final in block AAD (Section 5.7.9).

The snapshot accumulator (Section 5.7.8) provides file-level integrity: a single Nh-octet value that binds all block tags under acc_key. Decryptors can verify whole-file consistency from metadata alone, without decrypting every block. The accumulator updates in O(1) when individual blocks are rewritten.

Applications requiring third-party verifiability (e.g., signatures) MUST use external signatures.

8.4. Implementation Considerations

The step sequence has AND semantics: an attacker must break every step to recover the CEK, so security is at least that of the strongest step. Compromising one step secret (e.g., a passphrase) allows the attacker to compute intermediate aggregator values up to that step; this is inherent to sequential chaining and does not weaken subsequent steps. Nonces MUST be unique: fresh lock_nonce per LOCK, and fresh random nonce per block. Implementations MUST zeroize sensitive values (CEK, KEK, PRKs, payload_key, acc_key, nonce_base)

immediately after use. Long-lived processes that retain a CEK (e.g., for incremental writes) SHOULD store it in swap-protected memory (e.g., mlock). To prevent error oracles, implementations exposing decryption to untrusted callers (e.g., network services, APIs) MUST return a single generic "decryption failed" error rather than distinguishing between wrong passphrase, wrong key, commitment mismatch, or AEAD failure. Local tools (e.g., CLI applications, test harnesses) MAY use the detailed error codes in Appendix C for diagnostics. Implementations MUST use constant-time AEAD, KEM, KDF, commitment comparison, and accumulator verification operations. This extends to passphrase KDF evaluation and Base64 decoding of secret material (Encrypted-CEK, step parameters carrying key material). Commitment and accumulator comparisons MUST use a constant-time equality function (e.g., CRYPTO_memcmp or equivalent). Trial decryption loops (Section 8.6.3) MUST NOT leak timing information about which candidate key succeeded.

8.5. Passphrase KDF Selection

SAFE supports two passphrase KDF variants with different security properties:

pass(kdf=argon2id, ...): Memory-hard function that resists GPU and ASIC attacks. The default parameters (64 MiB memory, 2 iterations) provide strong resistance to offline attacks. Recommended for most deployments.

pass(kdf=pbkdf2, ...): Widely deployed function using PBKDF2-HMAC-SHA-256. Lacks memory-hardness, making it more vulnerable to GPU and ASIC attacks than Argon2id. The 600,000 iteration count provides equivalent CPU-based attack resistance but does not mitigate hardware-based attacks. Use only when policy prohibits memory-hard KDFs.

Encryptors targeting Decryptors with mixed policy constraints MAY include two pass(...) LOCK blocks: one with pass(kdf=argon2id, ...) and one with pass(kdf=pbkdf2, ...), using the same passphrase but fresh salts for each.

Applications SHOULD enforce a minimum passphrase complexity policy (e.g., at least 20 characters or equivalent entropy). For high-value data, Encryptors SHOULD combine a pass(...) step with an hpke(...) step in the same LOCK, so that compromise of the passphrase alone is insufficient.

Multiple LOCK blocks allow observers to infer shared payload access. HPKE key identifiers link files to the same recipient across objects. The Base64 length reveals approximate payload size; LOCK count reveals recipient count. Applications concerned about traffic analysis SHOULD pad payloads.

8.6. Recipient Anonymity and Trial Decryption

SAFE supports three levels of recipient identification for hpke(...) steps:

Identified mode: The id parameter uniquely identifies the recipient's public key. Observers can link SAFE-encoded data encrypted to the same recipient.

Hinted mode: The hint parameter is a recipient-assigned value (not cryptographically derived). It filters candidates locally while revealing nothing about the key itself. Multiple keys may share the same hint.

Anonymous mode: No identifier is present. Decryptors MUST trial-decrypt against all local keys matching the kem type. Provides maximum privacy at the cost of increased decryptor computation.

8.6.1. Privacy Benefits

Omitting or replacing the key identifier with a hint prevents passive observers from mapping SAFE-encoded data to specific public keys. This is valuable when file-recipient associations are sensitive metadata.

8.6.2. Sender Anonymity

Auth-mode hpke(...) steps support the same three levels of sender identification via sid and shint:

Identified (sid present): The sid parameter identifies the sender's public key using the same Hash as id. Observers can link SAFE objects to the same sender across files.

Hinted (shint present): The shint parameter is a sender-assigned value that filters candidates locally. It reveals less than sid but still narrows the sender's identity.

Anonymous (neither sid nor shint): Decryptors trial-decrypt against all locally known sender keys matching the kem type. Provides sender privacy at the cost of increased trial decryption (see Section 8.6.3).

Encryptors SHOULD prefer sid unless sender privacy is required. The same trade-offs between identification, hinting, and anonymity apply to sender keys as to recipient keys.

8.6.3. Trial Complexity

Anonymous mode with composable step sequences (multiple hpke steps) requires combinatorial trial decryption. For a step sequence with two anonymous hpke(...) steps, where the Decryptor holds K1 keys for step 1 and K2 keys for step 2, up to K1 x K2 combinations may be attempted. Auth-mode steps add a further multiplicative factor: if an auth-mode step has no sid or shint, the Decryptor MUST try all S candidate sender keys, multiplying the search space by S.

Implementations MUST set a MaxTrialAttempts limit to bound computation and MUST reject LOCK blocks that would exceed this limit. A value of 1024 is RECOMMENDED; implementations MAY adjust based on deployment constraints.

8.7. Denial of Service Considerations

An attacker can craft SAFE-encoded data with many anonymous LOCK blocks to force Decryptors into expensive cryptographic operations. Implementations MUST:

* Limit the number of LOCK blocks processed per object
* Prioritize identified blocks over hinted blocks over anonymous blocks
* Abort early when resource limits are exceeded

Implementations MUST also limit the number of steps per LOCK block. A limit of 16 steps per LOCK is RECOMMENDED; this prevents an attacker from crafting a single LOCK block that forces evaluation of an excessive number of passphrase KDF computations. Implementations SHOULD limit the total number of passphrase KDF evaluations to 8 per file; an attacker who crafts multiple LOCKs with pass(...) steps can otherwise force expensive Argon2id computations proportional to the LOCK count.

ML-KEM decapsulation is significantly more expensive than X25519; anonymous ML-KEM steps amplify the DoS potential.

8.8. Hint Assignment

The hint is a 4-digit decimal value (0000-9999) assigned by the recipient; it is not solely dependent on the public key. Recipients communicate their hint to Encryptors out-of-band. Multiple keys MAY share the same hint.

Encryptors MUST NOT assume the hint uniquely identifies a key. Decryptors MAY reassign hints at any time; Encryptors SHOULD refresh hint values periodically through out-of-band communication.

Encryptors SHOULD prefer identified mode unless recipient privacy is required.

8.9. Nonce Generation and CEK Reuse

Encryptors SHOULD use the hedged construction (Section 5.4.2.2) when a private key is available, the plaintext-bound nonce construction (Section 5.7.6.2) when RNG state duplication is a concern, and an NMR AEAD (Section 5.2.1) for additional protection. The following cases describe the resulting security properties.

With private key, working RNG: Full protection. The block nonce base is derived from both fresh randomness and the hedge key. Nonces are unique across files and within files.

With private key, duplicated RNG state: Deterministic encryption per Encryptor. Different Encryptors (with different private keys) produce different hedge keys and therefore different CEKs and nonce bases. Within a single file, block indices guarantee nonce uniqueness. Across files from the same encryptor, the CEK and salt repeat, producing identical payload keys and ciphertext for identical plaintext blocks at the same index. This leaks equality but not content. The plaintext-bound nonce construction (Section 5.7.6.2) further limits exposure: nonces differ when plaintext differs, even across files.

Without private key, working RNG: Full protection. SafeRandom returns raw CSPRNG output. Nonce uniqueness depends on the RNG.

Without private key, duplicated RNG state: No defense. CEKs and

salts repeat across files, producing identical payload keys and nonce bases. Within a file, block indices still provide distinct nonces. Across files, nonce reuse under distinct plaintext permits key recovery attacks against AES-GCM, ChaCha20-Poly1305, AEGIS-256, and AEGIS-256X2. AES-256-GCM-SIV limits the damage to deterministic encryption (leaks equality). The plaintext-bound nonce construction also limits nonce reuse to identical blocks at the same index.

A functioning CSPRNG is REQUIRED when no private key is available.

Encryptors operating in environments where RNG state duplication is possible (VM snapshots, process forks without reseed, container cloning) SHOULD use the plaintext-bound nonce construction (Section 5.7.6.2). Because the plaintext-bound construction incorporates a plaintext-dependent derivation via SafeDerive("SAFE-NONCE", plaintext_i, encryption_parameters, 32) into nonce derivation, two instances that share identical key material still produce distinct nonces whenever plaintext differs. The two-pass cost of this construction is justified by the defense it provides against state duplication.

Within a single file, block indices are bounded by the plaintext length and Block-Size. Encryptors MUST ensure block indices remain below 2^64. Practical implementations SHOULD enforce a lower bound; for example, rejecting plaintexts exceeding 2^48 blocks (approximately 16 EiB at the default Block-Size of 65536 octets) provides a conservative margin while supporting files far larger than current storage systems.

8.9.1. File Extension

Appending data to an existing SAFE file requires re-encrypting the old final block (which had is_final=1) with is_final=0, then encrypting the new blocks. Encryptors MUST NOT generate a new CEK or salt; the existing LOCKs and salt are reused. Encryptors MUST verify the snapshot accumulator before extending the file; extending a corrupted file propagates undetected damage. The procedure is:

1. Decrypt the current final block (index N-1) and verify is_final=1.

2. Re-encrypt block N-1 with is_final=0. For non-NMR AEADs, Encryptors MUST generate a fresh nonce using SafeRandom (Section 5.4.2). For NMR AEADs, re-encrypting block N-1 at the same index produces the same nonce (nonce_base XOR uint64(N-1)). The AAD differs because is_final changes from 1 to 0; nonce-misuse resistance ensures security despite the repeated nonce. See Section 8.9.2.

3. Encrypt new blocks N through N+K-1 with is_final=0.

4. Encrypt block N+K (the new final block) with is_final=1.

5. Update the metadata (nonces, tags) and block count N.

6. Recompute the snapshot accumulator.

For NMR AEADs, re-encryption of block N-1 at the same index reuses the same derived nonce. Because the AAD differs (is_final changed from 1 to 0), the NMR AEAD produces different ciphertext. This is the standard NMR equality-leakage property: an observer can detect that block N-1 was re-encrypted, but content is not revealed.

8.9.2. Derived Nonces

For NMR AEADs, per-block nonces are derived deterministically from nonce_base and the block index rather than generated randomly and stored. This is restricted to NMR AEADs for the following reasons:

Uniqueness: The SafeDerive output is unique per CEK (each CEK produces a distinct nonce_base). XOR with distinct block indices yields distinct nonces for all i < 2^64.

Nonce reuse tolerance: Re-encrypting block i with the same CEK reuses nonce_i. NMR AEADs degrade gracefully to deterministic encryption: identical plaintext at the same index produces identical ciphertext, but no additional information is leaked. Non-NMR AEADs would suffer catastrophic nonce reuse, which is why derived nonces are not used with them.

8.10. Selective Editing Security

Per-block random nonces and the is_final flag enable selective editing: individual blocks can be re-encrypted without affecting other blocks or LOCK blocks. When editing:

* Generate a fresh random nonce for any re-encrypted block

* Update the is_final flag if the last block changes

* Update the snapshot accumulator (Section 5.7.8)

* Blocks not being edited retain their original nonces and ciphertexts

The is_final flag prevents truncation and extension attacks:

* Truncation: decrypting a block with is_final=0 when no successor exists indicates malicious or accidental truncation

* Extension: appending blocks after a block with is_final=1 will fail AEAD verification because the original final block's AAD included is_final=1

8.11. Key Identifier Collisions

Key identifiers are 32-octet hashes of SPKI DER encodings (Section 5.6.3.3). Both registered Hash algorithms (sha-256 and turboshake256) produce 32-octet output, giving a birthday bound on collision probability of approximately N^2 / 2^257 for a deployment with N keys. This is negligible for any practical key population.

Implementers MUST NOT rely solely on key identifier matching for authorization; successful HPKE decapsulation and AEAD verification of Encrypted-CEK are required.

8.12. Key Commitment

SAFE supports multiple LOCK blocks that can be added or removed independently without re-encrypting the payload, because each LOCK wraps the same CEK. However, removing a LOCK does not revoke access: any party who previously decrypted the CEK retains the ability to decrypt the payload. Applications requiring revocation MUST generate a new CEK and re-encrypt.

Without key commitment, an adversary could craft LOCK blocks that decrypt to different CEKs and exploit AEAD malleability to create payload ciphertext valid under multiple keys:

1. Creates LOCK that wraps CEK for recipient A

2. Creates LOCK that wraps CEK for recipient B

3. Crafts payload ciphertext C that decrypts to plaintext P under payload_key (derived from CEK) and to P under payload_key (derived from CEK)

None of the AEADs registered for SAFE provide key commitment from the AEAD mechanism alone at the 128-bit security level (AES-GCM and ChaCha20-Poly1305 are well-known to lack this property; AEGIS-256 with 128-bit tags provides only birthday-bound commitment at approximately 2^64). SAFE's external 32-octet commitment prefix (Section 5.7.3) supersedes the AEAD's native commitment properties, providing uniform 2^128 key-commitment security across all registered suites.

The commitment prefix in the SAFE DATA block (Section 6.4.1) provides uniform key commitment for all AEAD choices. The commitment is always 32 octets, derived via SafeDerive("commit", CEK, payload_info, 32) where payload_info = [...encryption_parameters, salt], per Section 5.7.3. Recipients verify that the derived commitment equals the prefix before block decryption. This binds the ciphertext to the CEK, the negotiated algorithm parameters, and the per-file salt, providing 2^128 key-commitment security and preventing cross-algorithm commitment collisions. The security relies on collision resistance of SafeDerive: the Encode() framing ensures unambiguous parsing of all inputs, so distinct (encryption_parameters, salt, CEK) tuples cannot produce the same commitment. Formally, for any two distinct input tuples the probability of a commitment collision is at most 2^(-128), under the assumption that the underlying KDF is a PRF.

8.13. AEAD Usage Bounds

The security properties described in this section address distinct threats and are provided by separate mechanisms. Key commitment (Section 8.12) prevents an adversary from crafting ciphertext that decrypts to different plaintexts under different keys; the 32-octet commitment prefix solves this uniformly for all registered AEADs and is independent of nonce discipline. Block-level integrity (Section 5.7.9) prevents reordering, truncation, and extension; the block index and final-block indicator in each block's AAD solve this independently of nonce collision risk.

Nonce collision risk is a separate concern. For non-NMR AEADs with stored random nonces, the primary practical constraint is the total number of block encryptions performed under one payload key over the file's lifetime. This total includes both initial writes and all subsequent rewrites of individual blocks. See Section 8.9 for defenses against RNG weakness and state duplication, including the hedged nonce construction (Section 5.4.2.2, [RFC8937]).

8.13.1. Lifetime Encryption Budget

Let q denote the total number of block encryptions under one payload key, counting every initial block write and every block rewrite. Because all recipients of a SAFE object share one CEK and therefore one payload key, the per-key analysis applies regardless of recipient count; adding recipients does not increase q. For AEADs with Nn-octet random nonces, the probability of at least one nonce collision among q encryptions is approximately:

P_collision q^2 / 2^(8*Nn + 1)

For 96-bit nonces (Nn = 12), this simplifies to approximately q^2 / 2^97. This is a standard birthday-bound approximation, not a formal proof of the AEAD's concrete multi-user security.

The following table illustrates q for representative workloads at the default Block-Size of 65536 octets:

Total encrypted data	Approximate q
1 TiB	2^24
16 TiB	2^28
256 TiB	2^32

Table 11

The same values of q can be reached by smaller files that are rewritten many times. A 1 GiB file contains 2^14 blocks at the default block size; rewriting the entire file 1024 times produces approximately 2^24 total block encryptions, the same budget as encrypting 1 TiB once. The relevant quantity is lifetime encrypted blocks per payload key, not current file size.

When Key-Epoch is present (Section 8.13.3), each epoch key covers at most 2^r blocks and their rewrites. The birthday bound then applies per epoch key rather than file-wide, so the effective q per key is bounded by the epoch size plus rewrites, not total file size. See Section 8.13.3 for per-epoch budget calculations.

In a multi-user setting where an attacker targets any one of U independently keyed SAFE files, the effective collision probability is approximately U * q^2 / 2^(8*Nn + 1). For 96-bit nonces with U = 2^20 files each at q = 2^32, this is approximately 2^(-13), which is

not negligible. Key-Epoch (Section 8.13.3) mitigates this by reducing the per-key q to the per-epoch budget; with Key-Epoch: 0 each key encrypts a single block, making per-key q equal to the number of rewrites of that block. At more typical workloads without epochs (q = 2^24 per file), the multi-user bound is approximately 2^(-29). The 256-bit nonce AEADs (AEGIS-256, AEGIS-256X2) render multi-user bounds operationally irrelevant (approximately 2^(-173) at the same parameters).

8.13.2. Per-AEAD Analysis

AES-256-GCM: AES-256-GCM [NIST-SP-800-38D] [RFC5116] is nonce-respecting with 96-bit nonces. SAFE uses a fresh random nonce per block for this suite. With stored random nonces, the practical limit is birthday-bound nonce collision under a single payload key: approximately q^2 / 2^97. At q = 2^32 (approximately 256 TiB at the default block size, or equivalent rewrite volume), the collision probability is approximately 2^(-33), which is negligible for most applications. Nonce reuse under AES-GCM permits authentication-key recovery and full plaintext recovery for the affected blocks. Encryptors SHOULD include Key-Epoch (Section 8.13.3) for rewritable files to limit per-key invocations within the bounds of [NIST-SP-800-38D]. Key-Epoch: 0 is RECOMMENDED; Key-Epoch: 5 is a practical alternative when key-derivation overhead matters (see Section 8.13.3 for the tradeoff). Alternatively, implementations MAY generate a fresh CEK and re-wrap it in new LOCKs.

ChaCha20-Poly1305: ChaCha20-Poly1305 [RFC8439] is nonce-respecting with 96-bit nonces. The ChaCha20-Poly1305 specification (Section 4 of [RFC8439]) requires nonce uniqueness per key and notes the collision risk of random nonces; SAFE’s use of stored random nonces gives the same birthday-style collision accounting as AES-256-GCM (approximately q^2 / 2^97). Nonce reuse under ChaCha20-Poly1305 permits XOR of plaintexts for the affected blocks and compromises Poly1305 authentication. Block rewrites are especially relevant because the same logical block may be encrypted multiple times under the same payload key, each time consuming budget. Because [RFC8439] requires nonce uniqueness per key, Encryptors MUST include Key-Epoch when using chacha20-poly1305 (Section 6.1). Key-Epoch: 0 is RECOMMENDED; Key-Epoch: 5 is a practical alternative when key-derivation overhead matters. Key-Epoch (Section 8.13.3) confines the collision domain to rewrites within each epoch.

AES-256-GCM-SIV: AES-256-GCM-SIV [RFC8452] is nonce-misuse resistant

(NMR). Under nonce reuse, it degrades to deterministic encryption: identical plaintext at the same block index produces identical ciphertext, leaking equality but not content. The collision analysis is therefore qualitatively different from AES-256-GCM and ChaCha20-Poly1305: the practical question is whether leaking block-level equality across rewrites is acceptable for a given application, not whether a nonce collision permits plaintext recovery. SAFE uses derived nonces for NMR AEADs (Section 5.7.5), which are deterministic per block index by design. Key-Epoch MUST NOT be present for this suite (Section 6.1). Unique nonces remain preferred to avoid equality leakage; misuse resistance does not provide unlimited safety. The concrete security bound for AES-256-GCM-SIV (Section 6 of [RFC8452]) includes a message-length-dependent term: the distinguishing advantage is bounded by approximately (q * l)^2 / 2^128, where q is the number of queries and l is the maximum message length in 128-bit blocks. At SAFE's default Block-Size of 65536 (l = 4096 blocks), this term is negligible for practical query counts.

AEGIS-256: AEGIS-256 [I-D.irtf-cfrg-aegis-aead] is nonce-respecting with 256-bit nonces. The birthday bound for 256-bit random nonces is approximately q^2 / 2^257. At q = 2^48 (approximately 16 EiB at the default block size), the collision probability is approximately 2^(-161), which is negligible for any foreseeable SAFE workload. The larger nonce space makes random-nonce collision operationally irrelevant at SAFE scale. Key-Epoch adds no practical benefit for this suite (Section 8.13.3). Implementations MUST still generate fresh random nonces for each block encryption.

AEGIS-256X2: AEGIS-256X2 shares the 256-bit nonce of AEGIS-256. The same analysis applies: the birthday bound of q^2 / 2^257 makes nonce collision negligible for any practical SAFE deployment. As with AEGIS-256, Key-Epoch adds no practical benefit. AEGIS-256X2 provides higher throughput on wide-vector hardware; the nonce discipline requirements are identical to AEGIS-256.

8.13.3. Epoch Key Rotation

Key-Epoch (Section 5.7.4) limits AEAD invocations per key. With Key-Epoch = r, each epoch key covers at most 2^r blocks (plus their rewrites). The birthday bound applies per epoch key, not file-wide.

For 96-bit nonce AEADs, each epoch key's budget is 2^32 encryptions (P < 2^(-33)). The total rewrite budget per epoch key is 2^32 - 2^r encryptions, shared across all 2^r blocks in the epoch. Assuming uniform rewrite distribution:

avg_rewrites_per_block = 2^(32-r) - 1

Key-Epoch	Epoch size	Avg rewrites/block
0	1	2^32 - 1
5	32	2^27 (134M)
8	256	2^24 (16M)
16	65536	2^16 (65K)

Table 12

In a multi-user setting with U files, the per-epoch multi-user bound is U * q_epoch^2 / 2^(8*Nn + 1), where q_epoch is the total encryptions under one epoch key (at most 2^r initial blocks plus their rewrites). With Key-Epoch: 0 (r=0) and W rewrites per block, q_epoch = 1 + W; even at U = 2^20 files and W = 2^20 rewrites, the bound is approximately 2^20 * 2^40 / 2^97 = 2^(-37).

If rewrites concentrate on a single block within an epoch, that block can consume the full epoch budget. With r=0 each key encrypts a single block, so the cross-block collision risk is eliminated and the only collision risk is across rewrites of that block — over 2^32 rewrites to reach the threshold. [FLOE] Section 8 benchmarks the overhead of epoch key derivation across segment sizes and rotation masks; at r=0 the overhead is noticeable for small segments but decreases rapidly with segment size, and by r=5 it is dominated by the cost of processing the plaintext. Each epoch key at r=5 still permits over 134 million rewrites per block.

Implementations SHOULD set Key-Epoch: 0 for maximum security. When throughput is critical (e.g., Hardware Security Module (HSM)-backed key derivation or constrained devices), Key-Epoch: 5 provides a practical alternative with strong security margins.

For 256-bit nonce AEADs (AEGIS-256, AEGIS-256X2), epoch rotation adds no practical benefit. Implementations SHOULD NOT include Key-Epoch for these suites.

Epoch keys are derived, not stored, so no LOCK re-wrapping is needed. This construction adapts the epoch-based key rotation of [FLOE] to SAFE’s key schedule.

8.13.4. Relationship to Key Commitment

The nonce-collision analysis above is orthogonal to key commitment. As noted in Section 8.12, none of the AEADs registered for SAFE provide key commitment from the AEAD mechanism alone at the target security level. SAFE therefore provides a uniform file-level commitment prefix that is verified before block decryption. Commitment addresses wrong-key and cross-parameter ambiguity; the nonce analysis in this section addresses per-key AEAD lifetime and rewrite safety. The choice of layout (linear versus aligned, Section 6.4) does not change the cryptographic analysis; it affects only framing and on-disk storage layout.

8.14. Algorithm Agility and Post-Quantum Support

SAFE accommodates post-quantum KEMs without format changes. ML-KEM-768 (HPKE KEM ID 0x0041) is registered and MAY be used as kem=ml-kem-768.

Hybrid post-quantum constructions require no protocol extensions. An encryptor lists both a classical and a post-quantum hpke(...) step in the same step sequence:

```
Step: hpke(kem=x25519, kemct=<Base64>, id=<classical-id>)
Step: hpke(kem=ml-kem-768, kemct=<Base64>, id=<pq-id>)
```

The KEK schedule (Section 5.7.1) folds each step’s secret into the aggregator in order, so the derived KEK depends on both the X25519 and ML-KEM-768 shared secrets. An attacker must break both KEMs to recover the KEK. Because the KEK schedule folds each step secret sequentially, the derived KEK’s security is conjunctive: an attacker must break every step. Hybrid post-quantum protection follows naturally from combining classical and post-quantum steps.

The decryptor evaluates both decapsulations during CEK recovery. Each KEM ciphertext is carried in the kemct parameter of its corresponding hpke(...) step token. See Appendix A for a complete example.

Implementations planning PQ migration SHOULD ensure kemct parsing does not impose unnecessary length limits (ML-KEM-768 ciphertexts are 1088 octets).

Auth mode (Section 5.6.3.1) relies on DHKEM AuthEncap/AuthDecap, which requires a CDH-hard group. No post-quantum KEM currently supports Auth mode; ML-KEM-768 MUST NOT be used with Auth mode (Section 5.6.3.2). Applications requiring post-quantum sender authentication MUST use external signatures.

8.15. Security Level and Design Notes

SAFE derives all symmetric keys at 256 bits. The AEAD tag length (128 bits), accumulator contribution length (Nh = 256 bits), and commitment length (256 bits) provide at least 128-bit security against forgery and collision attacks.

The KEK schedule initializer kek_init = SafeDerive("kek_init", "", encryption_parameters, 32) uses an empty ikm. This is a public derivation: its output is deterministic and computable by anyone who knows the encryption parameters. Security of the KEK relies on the step secrets folded in subsequent aggregator rounds, not on kek_init being secret.

The Encode function is injective for a fixed number of arguments: distinct input tuples produce distinct outputs because each field is length-prefixed. This ensures that binding step_tokens are unambiguous.

Per-block nonces are unique within a file by construction: for non-NMR AEADs, each nonce is generated via SafeRandom; for NMR AEADs, nonce_base XOR block_index is unique because block indices are unique. The relevant birthday bound for nonce collisions is across files or across block rewrites under the same payload key; see Section 8.13 for per-AEAD analysis.

8.16. Downgrade Resistance

SAFE has no algorithm negotiation: the Encryptor selects encryption_parameters unilaterally, and the Decryptor either accepts them or fails. An active attacker who modifies encryption_parameters (e.g., substituting a weaker AEAD) changes the derived KEK (Section 5.7.1) and payload keys (Section 5.7.3), causing CEK unwrapping or block decryption to fail. Forging a valid SAFE object with altered parameters requires the attacker to also hold valid credentials for the target LOCK.

Decryptors SHOULD enforce a locally configured allowlist of acceptable encryption parameters. Rejecting algorithms outside the allowlist limits the attack surface to supported primitives.

The symmetric components of SAFE (HPKE export-only key schedules, SafeDerive-based KEK aggregation, commitment prefixes, and AEAD encryption) are not vulnerable to quantum attacks. Grover’s algorithm provides at most a quadratic speedup against symmetric primitives, leaving all symmetric operations at or above 128-bit security. The post-quantum migration surface is limited to KEM selection.

A deployment using Hash=turboshake256 eliminates all SHA-256 dependencies: per [I-D.ietf-hpke-hpke], all KEMs use the Hash-selected KDF internally.

9. Privacy Considerations

This section addresses the privacy properties of SAFE per [RFC6973]. SAFE is primarily a data-at-rest format; it does not define a transport protocol, so many [RFC6973] considerations (correlation by IP, traffic analysis of flows) do not apply directly.

SAFE-encoded data reveals the following metadata to passive observers: approximate payload size (from Base64 or binary length), recipient count (from the number of LOCK blocks), and — when key identifiers are present — linkability across files encrypted to the same recipient (Section 8.6). Hinted mode (Section 8.6.1) reduces linkability; anonymous mode eliminates key-identifier-based linking at the cost of increased trial decryption. Auth-mode sender identifiers (sid, shint) create analogous sender linkability (Section 8.6.2). Applications concerned about metadata leakage SHOULD pad payloads and SHOULD prefer hinted or anonymous modes.

Beyond the metadata listed above, the step types and KEM identifiers in each LOCK are visible in cleartext, revealing the authentication factors required (e.g., passphrase, X25519, ML-KEM-768). In multi-recipient files, the set of LOCK blocks reveals co-recipient relationships. For NMR AEADs, block rewrites at the same index produce identical ciphertext when the plaintext is unchanged, leaking equality (Section 8.9.2). A comprehensive [RFC6973] privacy analysis is deferred to a future revision.

10. IANA Considerations

10.1. SAFE AEAD Identifiers Registry

IANA is requested to create a SAFE AEAD Identifiers registry. Registration policy is Specification Required. Designated Experts should verify that proposed AEADs provide [RFC5116] semantics with a 16-octet authentication tag and Nk of 32 octets (SAFE derives all keys at 256 bits). Experts MUST reject registrations where Nk is not 32. Identifiers MUST consist of lowercase ASCII letters, digits, and hyphens, and MUST NOT exceed 255 octets. The algorithm MUST be appropriate for general-purpose use in encrypted data formats.

Initial entries:

Identifier	Nk	Nn	NMR	Key-Epoch	Reference	Change Controller
aes-256-gcm	32	12	No	Recommended	[NIST-SP-800-38D]	IETF
chacha20-poly1305	32	12	No	Required	[RFC8439]	IETF
aes-256-gcm-siv	32	12	Yes	Not Applicable	[RFC8452]	IETF
aegis-256	32	32	No	Not Recommended	[I-D.irtf-cfrg-aegis-aead]	IETF
aegis-256x2	32	32	No	Not Recommended	[I-D.irtf-cfrg-aegis-aead]	IETF

Table 13

Nk/Nn are key/nonce sizes in octets. Nn is required to compute block boundaries (Section 5.7.5). "NMR" indicates nonce-misuse resistance. "Key-Epoch" indicates whether Encryptors should include Key-Epoch in CONFIG; see Section 6.1 for normative requirements and Section 5.7.4 for details.

10.2. SAFE KEM Identifiers Registry

IANA is requested to create a SAFE KEM Identifiers registry. This registry maps SAFE's string identifiers (used in kem= parameters) to HPKE KEM IDs. Registration policy is Specification Required. Designated Experts should verify that the KEM is registered in the IANA HPKE KEM Identifiers registry, is compatible with HPKE export-only mode (AEAD ID 0xFFFF) as specified in Section 5.6.3, and that a specification for SPKI encoding of public keys is provided. The Auth column indicates whether the KEM supports AuthEncap/AuthDecap for HPKE Auth mode. Encryptors MUST NOT include sid or shint with KEMs where Auth=No.

Initial entries:

Identifier	HPKE KEM ID	Encap Size	Auth	SPKI Encoding	Change Controller
x25519	0x0020	32 octets	Yes	[RFC8410]	IETF
p-256	0x0010	65 octets	Yes	[RFC5480]	IETF
ml-kem-768	0x0041	1088 octets	No	[I-D.ietf-lamps-kyber-certificates]	IETF

Table 14

HPKE KEM IDs are defined in the IANA HPKE KEM Identifiers registry established by [RFC9180]. Identifiers MUST consist of lowercase ASCII letters, digits, and hyphens, and MUST NOT exceed 255 octets.

10.3. SAFE KDF Identifiers Registry

IANA is requested to create a SAFE KDF Identifiers registry. Registration policy is Specification Required. Designated Experts should verify that identifiers consist of lowercase ASCII letters, digits, and hyphens, do not exceed 255 octets, that the underlying KDF provides at least 128-bit security, and that the entry references a KDF registered in the HPKE KDF Identifiers registry (Section 7.2 of [RFC9180]).

Each entry MUST reference a KDF registered in the HPKE KDF Identifiers registry. Two-stage KDFs provide Extract(salt, ikm), Expand(prk, info, L), and Nh. Single-stage KDFs ([I-D.ietf-hpke-pq]) provide Derive(ikm, L). The Nh column specifies the output size in octets used for accumulator contributions and commitment derivation; Nh MUST be 32 for all registrations. The Class column determines which SafeDerive instantiation is used (see Section 5.4).

The CONFIG field name remains "Hash" for wire compatibility; the IANA registry is named "SAFE KDF Identifiers" because the identifiers select a KDF (and its underlying hash function) rather than a bare hash algorithm.

All conforming implementations MUST implement sha-256, which is the default when Hash is omitted from the SAFE CONFIG. Implementations MAY implement turboshake256.

Initial entries:

Identifier	HPKE KDF ID	Class	Nh	Reference	Change Controller
sha-256	0x0001	two-stage	32	[RFC5869]	IETF
turboshake256	TBD	single-stage	32	[I-D.ietf-hpke-pq]	IETF

Table 15

[RFC Editor: The HPKE KDF ID for turboshake256 is pending allocation in the HPKE KDF Identifiers registry per [I-D.ietf-hpke-pq]. Replace "TBD" with the assigned value before publication.]

10.4. SAFE Step Names Registry

IANA is requested to create a SAFE Step Names registry. Each registration defines a step type conforming to the interface in Section 5.6. The registry has the following columns:

Step Name: Unique ASCII identifier for the step type (e.g., "pass", "hpke").

Parameters Grammar: ABNF grammar for step-specific parameters in the token, or "None" if no parameters.

Inputs: Description of required inputs (e.g., "user passphrase, salt", "recipient private key, kemct").

Secret Length: MUST be 32 octets for all registered steps.

Reference: Document specifying the step’s derivation algorithm.

Registration policy is Specification Required. Designated Experts MUST verify:

* The derivation algorithm is deterministic and produces exactly 32 octets

* Parameter names do not conflict with existing registrations

* The specification provides complete implementation guidance including the Encode binding form

Step names MUST match the grammar 1*(ALPHA / DIGIT / "-") and MUST NOT exceed 255 octets.

Initial entries:

Step Name	Parameters Grammar	Inputs	Secret Length	Reference	Change Controller
pass	kdf, salt, (t, p, m for argon2id; c for pbkdf2)	passphrase, salt	32 octets	Section 5.6.2	IETF
hpke	kem, kemct, id, (sid, shint for auth)	private key, kemct	32 octets	Section 5.6.3	IETF

Table 16

The pass step's algorithm variant (argon2id or pbkdf2) is specified in the step token's kdf parameter. The default KDF is argon2id; pbkdf2 is available for environments where policy prohibits Argon2id.

The hpke step additionally requires the step token itself as input. When sid or shint is present, the sender's private key (for encryption) or public key (for decryption) is also required. Supported kem values are defined in Section 5.6.3.2; key identifier computation is defined in Section 5.6.3. hpke(...) with sid or shint is OPTIONAL and limited to DHKEM-based KEMs (x25519, p-256).

Future registrations MAY define additional step types (e.g., hardware token, Oblivious Pseudorandom Function (OPRF)) or variant algorithms for existing step names (subject to Designated Expert review for interoperability impact). A registration request MUST include:

* Step Name and Parameters Grammar (ABNF)

* Complete list of Inputs with their sources

* Derivation algorithm producing exactly 32 octets

* Definition of any step-specific parameters (name, encoding, semantics)

* Security considerations for the step type

See Appendix L for an illustrative example.

10.5. SAFE Config Options Registry

IANA is requested to create a SAFE Config Options registry. Each registration defines a CONFIG field name and the registry or value set that defines its legal values. The registry has the following columns:

Field Name: Case-sensitive ASCII field name used in SAFE CONFIG.

Value Definition: Registry or fixed set of values allowed for the field.

Reference: Document specifying the field and its semantics.

Registration policy is Specification Required. Designated Experts MUST verify that the field name does not conflict with existing registrations, that the specification defines default behavior when the field is absent, and that the value definition is unambiguous.

Initial entries:

Field Name	Value Definition	Reference	Change Controller
AEAD	SAFE AEAD Identifiers registry	Section 10.1	IETF
Block-Size	16384, 65536	Section 6.1	IETF
Hash	SAFE KDF Identifiers registry	Section 10.3	IETF
Key-Epoch	Non-negative integer; absent when disabled	Section 5.7.4	IETF
Lock-Encoding	armored (default), readable	Section 6.2	IETF
Data-Encoding	armored (default), binary, binary-linear	Section 6.3	IETF

Table 17

The default encoding is "armored" when the Data-Encoding field is omitted. The default Lock-Encoding is "armored" when omitted. Key-Epoch is absent (disabled) when omitted. Block-Size values are fixed by this specification; adding new values requires a Standards Track document update.

10.6. SAFE Block Types Registry

IANA is requested to create a SAFE Block Types registry. This registry lists the block types that may appear in SAFE-encoded data, identified by their fence markers. The registry has the following columns:

Block Type: The block type name as it appears in fence markers (e.g., "CONFIG", "LOCK", "DATA").

Fence Marker: The opening fence marker string (e.g., "-----BEGIN SAFE CONFIG-----").

Ignorable: "Yes" if Decryptors MAY skip unrecognized instances of this block type without failing; "No" if Decryptors MUST fail when encountering an unrecognized block of this type.

Reference: Document defining the block’s semantics.

Registration policy is Specification Required. Designated Experts MUST verify:

* The block type name does not conflict with existing registrations

* The specification clearly defines the block’s syntax and semantics

* Blocks marked Ignorable=Yes do not affect security or correctness if omitted

Block type names MUST consist of uppercase ASCII letters and hyphens and MUST NOT exceed 32 octets.

Initial entries:

Block Type	Fence Marker	Ignorable	Reference	Change Controller
CONFIG	-----BEGIN SAFE CONFIG-----	No	Section 6.1	IETF
LOCK	-----BEGIN SAFE LOCK-----	No	Section 6.2	IETF
DATA	-----BEGIN SAFE DATA-----	No	Section 6.3.1	IETF

Table 18

All initial block types are critical (Ignorable=No). Future extensions MAY register new block types with Ignorable=Yes for optional features such as detached signatures, metadata, or recipient hints.

10.7. Media Type Registration

IANA is requested to register the following media type per [RFC6838]:

Type name: application

Subtype name: safe

Required parameters: None

Optional parameters: None

Encoding considerations: Binary or 7bit. Armored-encoded SAFE data (the default) consists of ASCII printable characters and line feeds, with Base64 encoding for payload data. Binary-encoded SAFE data have ASCII headers followed by raw binary payload data.

Security considerations: SAFE-encoded data contain encrypted content. See Section 8 of this document.

Interoperability considerations: SAFE-encoded data are ASCII-armored with PEM-style fence markers. Line wrapping of Base64 content is permitted; Decryptors MUST accept any line length.

Published specification: This document

Applications that use this media type: File encryption, secure file

sharing, encrypted backups

Fragment identifier considerations: None

Additional information: Deprecated alias names for this type: None

Magic number(s): Files begin with "-----BEGIN SAFE" (ASCII)

File extension(s): .safe

Macintosh file type code(s): None

Person & email address to contact for further information: N. Sullivan (nicholas.sullivan+ietf@gmail.com)

Intended usage: COMMON

Restrictions on usage: None

Author: See Authors' Addresses section

Change controller: IETF

11. References

11.1. Normative References

[I-D.ietf-hpke-hpke]
Barnes, R., Bhargavan, K., Lipp, B., and C. A. Wood, "Hybrid Public Key Encryption", Work in Progress, Internet-Draft, draft-ietf-hpke-hpke-03, 2 March 2026, <https://datatracker.ietf.org/doc/html/draft-ietf-hpke-hpke-03>.

[I-D.ietf-hpke-pq]
Barnes, R. and D. Connolly, "Post-Quantum and Post-Quantum/Traditional Hybrid Algorithms for HPKE", Work in Progress, Internet-Draft, draft-ietf-hpke-pq-04, 2 March 2026, <https://datatracker.ietf.org/doc/html/draft-ietf-hpke-pq-04>.

[I-D.ietf-lamps-kyber-certificates]
Turner, S., Kampanakis, P., Massimo, J., and B. Westerbaan, "Internet X.509 Public Key Infrastructure - Algorithm Identifiers for the Module-Lattice-Based Key-Encapsulation Mechanism (ML-KEM)", Work in Progress, Internet-Draft, draft-ietf-lamps-kyber-certificates-11, 22 July 2025, <https://datatracker.ietf.org/doc/html/draft-ietf-lamps-kyber-certificates-11>.

[I-D.irtf-cfrg-aegis-aead]
Denis, F. and S. Lucas, "The AEGIS Family of Authenticated Encryption Algorithms", Work in Progress, Internet-Draft, draft-irtf-cfrg-aegis-aead-18, 5 October 2025, <https://datatracker.ietf.org/doc/html/draft-irtf-cfrg-aegis-aead-18>.

[NIST-SP-800-38D]
Dworkin, M., "Recommendation for Block Cipher Modes of Operation: Galois/Counter Mode (GCM) and GMAC", NIST Special Publication 800-38D, November 2007, <https://nvlpubs.nist.gov/nistpubs/Legacy/SP/nistspecialpublication800-38d.pdf>.

[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>.

[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>.

[RFC5116] McGrew, D., "An Interface and Algorithms for Authenticated Encryption", RFC 5116, DOI 10.17487/RFC5116, January 2008, <https://www.rfc-editor.org/rfc/rfc5116>.

[RFC5234] Crocker, D., Ed. and P. Overell, "Augmented BNF for Syntax Specifications: ABNF", STD 68, RFC 5234, DOI 10.17487/RFC5234, January 2008, <https://www.rfc-editor.org/rfc/rfc5234>.

[RFC5480] Turner, S., Brown, D., Yiu, K., Housley, R., and T. Polk, "Elliptic Curve Cryptography Subject Public Key Information", RFC 5480, DOI 10.17487/RFC5480, March 2009, <https://www.rfc-editor.org/rfc/rfc5480>.

[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>.

[RFC6838] Freed, N., Klensin, J., and T. Hansen, "Media Type Specifications and Registration Procedures", BCP 13, RFC 6838, DOI 10.17487/RFC6838, January 2013, <https://www.rfc-editor.org/rfc/rfc6838>.

[RFC8018] Moriarty, K., Ed., Kaliski, B., and A. Rusch, "PKCS #5: Password-Based Cryptography Specification Version 2.1", RFC 8018, DOI 10.17487/RFC8018, January 2017, <https://www.rfc-editor.org/rfc/rfc8018>.

[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>.

[RFC8410] Josefsson, S. and J. Schaad, "Algorithm Identifiers for Ed25519, Ed448, X25519, and X448 for Use in the Internet X.509 Public Key Infrastructure", RFC 8410, DOI 10.17487/RFC8410, August 2018, <https://www.rfc-editor.org/rfc/rfc8410>.

[RFC8439] Nir, Y. and A. Langley, "ChaCha20 and Poly1305 for IETF Protocols", RFC 8439, DOI 10.17487/RFC8439, June 2018, <https://www.rfc-editor.org/rfc/rfc8439>.

[RFC8452] Gueron, S., Langley, A., and Y. Lindell, "AES-GCM-SIV: Nonce Misuse-Resistant Authenticated Encryption", RFC 8452, DOI 10.17487/RFC8452, April 2019, <https://www.rfc-editor.org/rfc/rfc8452>.

[RFC9106] Biryukov, A., Dinu, D., Khovratovich, D., and S. Josefsson, "Argon2 Memory-Hard Function for Password Hashing and Proof-of-Work Applications", RFC 9106, DOI 10.17487/RFC9106, September 2021, <https://www.rfc-editor.org/rfc/rfc9106>.

[RFC9180] Barnes, R., Bhargavan, K., Lipp, B., and C. Wood, "Hybrid Public Key Encryption", RFC 9180, DOI 10.17487/RFC9180, February 2022, <https://www.rfc-editor.org/rfc/rfc9180>.

11.2. Informative References

[AEA] Apple Inc., "Apple Encrypted Archive (AEA)", 2024, <https://support.apple.com/guide/security/protecting-app-access-to-user-data-secddd150c21/web>.

[AGE] Valsorda, F., "The age file encryption format", 2025, <https://age-encryption.org/v1>.

[AGE-COMMIT] Stuble, M., "Actually Good Encryption? Confusing Users by Changing Nonces", ETH Zrich Semester Project, 2022, <https://ethz.ch/content/dam/ethz/special-interest/infk/inst-infsec/appliedcrypto/education/theses/semester-project_mirco-stauble.pdf>.

[FLOE] Fbrega, A., Len, J., Ristenpart, T., and G. Rubin, "Random-Access AEAD for Fast Lightweight Online Encryption", IACR ePrint 2025/2275, 2025, <https://eprint.iacr.org/2025/2275>.

[I-D.ietf-ohai-chunked-ohttp] Pauly, T. and M. Thomson, "Chunked Oblivious HTTP Messages", Work in Progress, Internet-Draft, draft-ietf-ohai-chunked-ohttp-08, 18 February 2026, <https://datatracker.ietf.org/doc/html/draft-ietf-ohai-chunked-ohttp-08>.

[RFC5652] Housley, R., "Cryptographic Message Syntax (CMS)", STD 70, RFC 5652, DOI 10.17487/RFC5652, September 2009, <https://www.rfc-editor.org/rfc/rfc5652>.

[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>.

[RFC7516] Jones, M. and J. Hildebrand, "JSON Web Encryption (JWE)", RFC 7516, DOI 10.17487/RFC7516, May 2015, <https://www.rfc-editor.org/rfc/rfc7516>.

[RFC8937] Cremers, C., Garratt, L., Smyshlyaev, S., Sullivan, N., and C. Wood, "Randomness Improvements for Security Protocols", RFC 8937, DOI 10.17487/RFC8937, October 2020, <https://www.rfc-editor.org/rfc/rfc8937>.

[RFC9497] Davidson, A., Faz-Hernandez, A., Sullivan, N., and C. A. Wood, "Oblivious Pseudorandom Functions (OPRFs) Using Prime-Order Groups", RFC 9497, DOI 10.17487/RFC9497, December 2023, <https://www.rfc-editor.org/rfc/rfc9497>.

[RFC9578] Celi, S., Davidson, A., Valdez, S., and C. A. Wood, "Privacy Pass Issuance Protocols", RFC 9578, DOI 10.17487/RFC9578, June 2024, <https://www.rfc-editor.org/rfc/rfc9578>.

[RFC9580] Wouters, P., Ed., Huigens, D., Winter, J., and Y. Niibe, "OpenPGP", RFC 9580, DOI 10.17487/RFC9580, July 2024, <https://www.rfc-editor.org/rfc/rfc9580>.

[RFC9605] Omara, E., Uberti, J., Murillo, S. G., Barnes, R., Ed., and Y. Fablet, "Secure Frame (SFrame): Lightweight Authenticated Encryption for Real-Time Media", RFC 9605, DOI 10.17487/RFC9605, August 2024, <https://www.rfc-editor.org/rfc/rfc9605>.

[STREAM] Hoang, V. T., Reyhanitabar, R., Rogaway, P., and D. Vizr, "Online Authenticated-Encryption and its Nonce-Reuse Misuse-Resistance", IACR ePrint 2015/189, 2015, <https://eprint.iacr.org/2015/189>.

[W3C.webauthn-3] "Web Authentication: An API for accessing Public Key Credentials - Level 3", W3C WD webauthn-3, W3C webauthn-3, <https://www.w3.org/TR/webauthn-3/>.

Appendix A. Examples

This appendix is informative.

Note: The examples in this section illustrate structure and formatting only. The Base64 values are placeholders and do not represent valid cryptographic outputs. Implementers requiring test vectors with known inputs and outputs should consult the Test Vectors appendix (Appendix G).

A minimal object with readable LOCK format (Lock-Encoding set to readable; aes-256-gcm, commitment prefix):

```
-----BEGIN SAFE CONFIG-----
Lock-Encoding: readable
-----END SAFE CONFIG-----
-----BEGIN SAFE LOCK-----
Step: pass(kdf=argon2id, salt=c2FsdHZhbHVlMTIzNDU2Nzg=)
Encrypted-CEK:
  MTIzNDU2Nzg5MDEyM0FCQ0RFRkdISUpLTE1OT1BRUlNU
  VVZXWFlaYWJjZGVmZ2hpamtsbW5vcHFyc3R1dnd4eXowMTIzNA==
-----END SAFE LOCK-----
-----BEGIN SAFE DATA-----
cGxhY2Vob2xkZXJjZWtjb21taXRtZW50aGFzaDEyMzQ1Njc4YWJjZGVmZ2hpamts
bW5vcHFyc3R1dnd4eXpBQkNERUZHSElKS0xNTk9QUVJTVFVWV1hZWjAxMjM0NTY3
ODkrLz09
-----END SAFE DATA-----

A LOCK block encoded as armored (default Lock-Encoding):

-----BEGIN SAFE LOCK-----
U3RlcDpwYXNzKGtkZj1hcmdvbjJpZCxzYWx0PWMyRnNkSFpoYkhWbE1USXpORFUy
TnpnPSlFbmNyeXB0ZWQtQ0VLOk1USXpORFUyTnpnNU1ERXlNMEZDUTBSRlJrZElT
VXBMVEUxT1QxQlJVbE5VVlZaWFdGbGFZV0pqWkdWbVoyaHBhbXRzYlc1dmNIRnlj
M1IxZG5kNGVYb3dNVEl6TkE9PQ==
-----END SAFE LOCK-----
-----BEGIN SAFE DATA-----
YmFzZTY0bG9ja2V4YW1wbGVkYXRhY2h1bmtwbGFjZWhvbG
RlcmV4YW1wbGVkYXRhMTIzNDU2
-----END SAFE DATA-----

HPKE recipient in armored format (derived from the readable example
above):

-----BEGIN SAFE CONFIG-----
Block-Size: 16384
-----END SAFE CONFIG-----
-----BEGIN SAFE LOCK-----
U3RlcDpocGtlKGtlbT14MjU1MTksaWQ9Wmk2bVFuWTVqOHBJWXEzbzZyV2dOdz09
LGtlbWN0PVlXSmpaR1ZtWjJocGFtdHNiVzV2Y0hGeWMzUjFkbmQ0ZVhveE1qTTBO
VFkzT0RrdylFbmNyeXB0ZWQtQ0VLOmNHRnpjM2R2Y21ReE1qTTBOVFkzT0Rrd1lX
SmpaR1ZtWjJocGFtdHNiVzV2Y0hGeWMzUjFkbmQ0ZVhveE1qTTBOVFkzT0Rrd1lX
SmpaR1ZtWjJocGFnPT0=
-----END SAFE LOCK-----
-----BEGIN SAFE DATA-----
aHBrZWV4YW1wbGVjZWtjb21taXRtZW50aGFzaHZhbHVlMTJWR2hwY3lCcGN5QmhJ
SE5oYlhCc1pTQmxibU55ZVhCMFpXUWdjR0Y1Ykc5aFpDQjNhWFJvSUcxMWJIUnBj
R3hsSUdOb2RXNXJjdz09
-----END SAFE DATA-----
```

A HPKE recipient with non-default Block-Size (AEAD omitted, uses default aes-256-gcm with commitment prefix):

```
-----BEGIN SAFE CONFIG-----
Block-Size: 16384
Lock-Encoding: readable
-----END SAFE CONFIG-----
-----BEGIN SAFE LOCK-----
Step: hpke(kem=x25519, id=Zi6mQnY5j8pIYq3o6rWgNw==,
  kemct=YWJjZGVmZ2hpamtsbW5vcHFyc3R1dnd4eXoxMjM0NTY3ODkw)
Encrypted-CEK:
  cGFzc3dvcmQxMjM0NTY3ODkwYWJjZGVmZ2hpamtsbW5vcHFy
  c3R1dnd4eXoxMjM0NTY3ODkwYWJjZGVmZ2hpag==
-----END SAFE LOCK-----
-----BEGIN SAFE DATA-----
aHBrZWV4YW1wbGVjZWtjb21taXRtZW50aGFzaHZhbHVlMTJWR2hwY3lCcGN5QmhJ
SE5oYlhCc1pTQmxibU55ZVhCMFpXUWdjR0Y1Ykc5aFpDQjNhWFJvSUcxMWJIUnBj
R3hsSUdOb2RXNXJjdz09
-----END SAFE DATA-----
```

HPKE recipient with Hash: turboshake256 (32-octet key identifier, default aes-256-gcm with commitment prefix):

```
-----BEGIN SAFE CONFIG-----
Hash: turboshake256
Lock-Encoding: readable
-----END SAFE CONFIG-----
-----BEGIN SAFE LOCK-----
Step: hpke(kem=x25519,
  id=dHVyYm9zaGFrZTI1NmV4YW1wbGVoYXNoMzJvY3RldHM=,
  kemct=dHVyYm9zaGFrZWtlbWNpcGhlcnRleHRleGFtcGxlMTIz)
Encrypted-CEK:
  dHVyYm9ub25jZTEyMzQ1Njc4OWFiY2RlZmdoaWprbG1u
  b3BxcnN0dXZ3eHl6MDEyMzQ1Njc4OTBhYmNkZWZnaGk=
-----END SAFE LOCK-----
-----BEGIN SAFE DATA-----
dHVyYm9zaGFrZWNla2NvbW1pdG1lbnRleGFtcGxlaGFzaDFSeGhiWEJzWlNCMWMy
bHVaeUJ6Y0d0cExYUjFjbUp2YzJoaGEyVXlOVFlnYTJWNUlHbGtaVzUwYVdacFpY
SWdkMmwwYUNBek1pMXZZM1JsZENBPT0=
-----END SAFE DATA-----
```

AEGIS-256 with TurboShake key identifier (commitment prefix in DATA):

```
-----BEGIN SAFE CONFIG-----
AEAD: aegis-256
Hash: turboshake256
Lock-Encoding: readable
-----END SAFE CONFIG-----
-----BEGIN SAFE LOCK-----
Step: hpke(kem=x25519,
  id=YWVnaXN0dXJib3NoYWtlaWRlbnRpZmllcjMyYnl0ZXM=,
  kemct=YWVnaXNrZW1jaXBoZXJ0ZXh0ZXhhbXBsZTEyMzQ1Njc4)
Encrypted-CEK:
  YWVnaXNub25jZTEyMzQ1Njc4OTAxMjM0NTY3ODkwMTIz
  YWJjZGVmZ2hpamtsbW5vcHFyc3R1dnd4eXowMTIzNDU2
  Nzg5MGFiY2RlZmdoaWprbG1ub3BxcnN0dXZ3eHl6
-----END SAFE LOCK-----
-----BEGIN SAFE DATA-----
YWVnaXNub2NvbW1pdG1lbnRjaHVua2NpcGhlcnRleHRvbmx5ZXhhbXBsZWRhdGE=
-----END SAFE DATA-----

Anonymous X25519 recipient (trial decryption required):

-----BEGIN SAFE CONFIG-----
Lock-Encoding: readable
-----END SAFE CONFIG-----
-----BEGIN SAFE LOCK-----
Step: hpke(kem=x25519,
  kemct=YW5vbnltb3VzZW5jYXBzdWxhdGVka2V5bWF0ZXJpYWwx
  MjM0NTY3ODkwYWJjZGVmZ2hpamtsbW5vcHFyc3R1dg==)
Encrypted-CEK:
  YW5vbnltb3VzZW5jcnlwdGVkY2VrZGF0YTEyMzQ1Njc4
  OTBhYmNkZWZnaGlqa2xtbm9wcXJzdHV2d3h5ejAxMg==
-----END SAFE LOCK-----
-----BEGIN SAFE DATA-----
YW5vbnltb3VzY2VrY29tbWl0bWVudGhhc2h2YWx1ZXBheWxvYWRjaHVua2RhdGE=
-----END SAFE DATA-----

Hinted ML-KEM-768 recipient (4-digit hint for candidate filtering):
```

```
-----BEGIN SAFE CONFIG-----
Lock-Encoding: readable
-----END SAFE CONFIG-----
-----BEGIN SAFE LOCK-----
Step: hpke(kem=ml-kem-768, hint=4217,
  kemct=bWxrZW1jaXBoZXJ0ZXh0d2l0aGhpbnRlZHJlY2lwaWVu
  dGZpbHRlcmluZ2V4YW1wbGVkYXRhMTIzNDU2Nzg5MA==)
Encrypted-CEK:
  aGludGVkZW5jcnlwdGVkY2VrZGF0YWV4YW1wbGUxMjM0
  NTY3ODkwYWJjZGVmZ2hpamtsbW5vcHFyc3R1dnd4eXo=
-----END SAFE LOCK-----
-----BEGIN SAFE DATA-----
aGludGVkY2VrY29tbWl0bWVudGhhc2h2YWx1ZXBheWxvYWRjaHVua2RhdGFoZXJl
-----END SAFE DATA-----
```

Two recipients, one passphrase-only and one HPKE (default aes-256-gcm):

```
-----BEGIN SAFE CONFIG-----
Lock-Encoding: readable
-----END SAFE CONFIG-----
-----BEGIN SAFE LOCK-----
Step: pass(kdf=argon2id, salt=cHdkc2FsdDEyMzQ1Njc4OTA=)
Encrypted-CEK:
  cHdkbm9uY2UxMjM0NTY3ODkwYWJjZGVmZ2hpamtsbW5vcHFy
  c3R1dnd4eXowMTIzNDU2Nzg5MGFiY2RlZmdoaWo=
-----END SAFE LOCK-----
-----BEGIN SAFE LOCK-----
Step: hpke(kem=p-256, id=Z1d0u6QG0cB2a4nM3Kp2Ww==,
  kemct=QUJDREVGR0hJSktMTU5PUFFSU1RVVldYWVphYmNkZWZnaGlq
  a2xtbm9wcXJzdHV2d3h5ejAxMjM0NTY3ODkwYWJjZGVmZ2hp
  amtsbW5vcHFyc3R1dnd4eXo=)
Encrypted-CEK:
  aHBrZW5vbmNlMTIzNDU2Nzg5MGFiY2RlZmdoaWprbG1ub3Bx
  cnN0dXZ3eHl6MDEyMzQ1Njc4OTBhYmNkZWZnaGlq
-----END SAFE LOCK-----
-----BEGIN SAFE DATA-----
bXVsdGlyZWNpcGllbnRjZWtjb21taXRtZW50aGFzaHZhbHVlVFhWc2RHa3RjbVZq
YVhCcFpXNTBJR1Y0WVcxd2JHVWdkMmwwYUNCemFHRnlaV1FnY0dGNWJHOWhaQ0Js
Ym1OeWVYQjBaV1FnYjI1alpRPT0=
-----END SAFE DATA-----
```

Multi-step sequence with AND semantics (passphrase AND HPKE, chacha20-poly1305 with commitment prefix):

```
-----BEGIN SAFE CONFIG-----
AEAD: chacha20-poly1305
Block-Size: 16384
Key-Epoch: 0
Lock-Encoding: readable
-----END SAFE CONFIG-----
-----BEGIN SAFE LOCK-----
Step: pass(kdf=argon2id, salt=bXVsdGlzdGVwc2FsdDEyMzQ=)
Step: hpke(kem=x25519, id=eEF3bXlyT3BWbXpLUjRCdz09,
  kemct=bXVsdGlzdGVwZXhhbXBsZWtlbWNpcGhlcnRleHQxMjM0NTY3)
Encrypted-CEK:
  bXVsdGlzdGVwbm9uY2UxMjM0NTY3ODkwYWJjZGVmZ2hpamts
  bW5vcHFyc3R1dnd4eXowMTIzNDU2Nzg5MGFiYw==
-----END SAFE LOCK-----
-----BEGIN SAFE DATA-----
bXVsdGlzdGVwY2VrY29tbWl0bWVudGhhc2h2YWx1ZTEyMzQ1VW1WeGRXbHlaWE1n
WW05MGFDQndZWE56ZDI5eVpDQkJUa1FnV0RJMU5URTVJSEJ5YVhaaGRHVWdhMlY1
SUhSdklHUmxZM0o1Y0hRPQ==
-----END SAFE DATA-----
```

Same multi-step example in armored format (default Lock-Encoding):

```
-----BEGIN SAFE CONFIG-----
AEAD: chacha20-poly1305
Block-Size: 16384
Key-Epoch: 0
-----END SAFE CONFIG-----
-----BEGIN SAFE LOCK-----
U3RlcDpwYXNzKGtkZj1hcmdvbjJpZCxzYWx0PWJYVnNkR2x6ZEdWd2MyRnNkREV5
TXpRPSlTdGVwOmhwa2Uoa2VtPXgyNTUxOSxpZD1lRUYzYlhseVQzQldiWHBMVWpS
Q2R6MDksa2VtY3Q9YlhWc2RHbHpkR1Z3WlhoaGJYQnNaV3RsYldOcGNHaGxjblJs
ZUhReE1qTTBOVFkzKUVuY3J5cHRlZC1DRUs6YlhWc2RHbHpkR1Z3Ym05dVkyVXhN
ak0wTlRZM09Ea3dZV0pqWkdWbVoyaHBhbXRzYlc1dmNIRnljM1IxZG5kNGVYb3dN
VEl6TkRVMk56ZzVNR0ZpWXc9PQ==
-----END SAFE LOCK-----
-----BEGIN SAFE DATA-----
bXVsdGlzdGVwY2VrY29tbWl0bWVudGhhc2h2YWx1ZTEyMzQ1VW1WeGRXbHlaWE1n
WW05MGFDQndZWE56ZDI5eVpDQkJUa1FnV0RJMU5URTVJSEJ5YVhaaGRHVWdhMlY1
SUhSdklHUmxZM0o1Y0hRPQ==
-----END SAFE DATA-----
```

Note: The armored LOCK examples above use placeholder Base64 values. The armored payload is Base64(Encode(step_1, ..., encrypted_cek)) per Section 6.2.2.

Two-passphrase step sequence (requires both passphrases to decrypt):

```
-----BEGIN SAFE CONFIG-----
Lock-Encoding: readable
-----END SAFE CONFIG-----
-----BEGIN SAFE LOCK-----
Step: pass(kdf=argon2id, salt=Zmlyc3RwYXNzd29yZHNhbHQ=)
Step: pass(kdf=pbkdf2, salt=c2Vjb25kcGFzc3dvcmRzYWx0)
Encrypted-CEK:
  dHdvcGFzc3dvcmRub25jZTEyMzQ1Njc4OTBhYmNkZWZnaGlq
  a2xtbm9wcXJzdHV2d3h5ejAxMjM0NTY3ODkwYWI=
-----END SAFE LOCK-----
```

Hybrid post-quantum step sequence (X25519 AND ML-KEM-768, default aes-256-gcm with commitment prefix):

```
-----BEGIN SAFE CONFIG-----
Lock-Encoding: readable
-----END SAFE CONFIG-----
-----BEGIN SAFE LOCK-----
Step: hpke(kem=x25519, id=cHF6RmlucHJpbnQxMjM0NTY3,
  kemct=eDI1NTE5a2VtY2lwaGVydGV4dDEyMzQ1Njc4OTBhYmNkZWY=)
Step: hpke(kem=ml-kem-768, id=bWxrZW1maW5nZXJwcmludDEyMw==,
  kemct=bWxrZW03NjhrZW1jaXBoZXJ0ZXh0ZXh0cmVtZWx5bG9uZ2Jh
  c2U2NGVuY29kZWRkYXRhYXBwcm94aW1hdGVseTEwODhvY3Rl
  dHNmb3JwcXNlY3VyaXR5dGhpc2lzZHVtbXlkYXRhZm9yZGVt
  ...
  NzY4a2VtY2lwaGVydGV4dGVuY2Fwc3VsYXRpb25kYXRh)
Encrypted-CEK:
  aHlicmlkbm9uY2UxMjM0NTY3ODkwYWJjZGVmZ2hpamtsbW5v
  cHFyc3R1dnd4eXowMTIzNDU2Nzg5MGFiY2RlZmc=
-----END SAFE LOCK-----
-----BEGIN SAFE DATA-----
aHlicmlkcHFjZWtjb21taXRtZW50aGFzaHZhbHVlMTIzNDU2VUc5emRDMXhkV0Z1
ZEhWdElHaDVZbkpwWkNCbGVHRnRjR3hsSUdSbGJXOXVjM1J5WVhScGJtY2dZMjl0
WW1sdVpXUWdXREkxTlRFNUlHRnVaQ0JOVEMxTFJVMHQ=
-----END SAFE DATA-----
```

Appendix B. Implementation Guide

This appendix is informative. For a concise summary of the encryptor and decryptor flows, see Section 3.

B.1. Encryptor Processing

Encryptor processing proceeds in three phases: setup, recipient key wrapping, and content encryption.

During setup, the Encryptor:

1. Selects an AEAD algorithm and Block-Size (or accepts defaults)

2. Emits a SAFE CONFIG block if using non-default values

3. Generates a random 32-octet CEK: SafeRandom(32, "SAFE-CEK")

4. Generates a 32-octet salt: SafeRandom(32, "SAFE-SALT")

Encryptors should generate a fresh CEK for each file. When the same CEK is reused across files, the per-file salt ensures payload keys are unique: each salt produces distinct commitment, payload_key, acc_key, and nonce_base values via Section 5.7.3.

For each recipient, the Encryptor wraps the CEK by:

1. Emitting Step lines with required step parameters (salt for pass steps, kemct for hpke steps)

2. Generating a fresh lock_nonce: SafeRandom(Nn, "SAFE-LOCK-NONCE")

3. Deriving step secrets and computing the KEK per Section 5.7.1

4. Emitting the Encrypted-CEK field

To encrypt content, the Encryptor:

1. Derives payload_key from the CEK, encryption_parameters, and salt per Section 5.7.3

2. Splits the plaintext into Block-Size blocks (N blocks total)

3. For NMR AEADs, derives nonce_base per Section 5.7.3 and computes nonce_i = nonce_base XOR uint64(i) for each block. For non-NMR AEADs, generates per-block nonces using one of the constructions in Section 5.7.6.

4. For each block i: a. Computes nonce_i per the chosen construction b. Determines is_final = 1 if i == N - 1, else 0 c. Constructs data_aad(i, is_final) per Section 5.7.9 d. Computes key_i = block_key(i) (Section 5.7.4). e. For non-NMR AEADs: emits nonce_i || AEAD.Seal(key_i, nonce_i, aad, block). For NMR AEADs: emits AEAD.Seal(key_i, nonce_i, aad, block) (ciphertext and tag only; nonce is not stored).

B.2. Decryptor Processing

Decryptor processing proceeds in three phases: configuration, CEK recovery, and content decryption.

During configuration, the Decryptor reads the SAFE CONFIG (if present) to learn non-default values and constructs encryption_parameters from AEAD, Block-Size, and Hash.

To recover the CEK, the Decryptor:

1. Selects a LOCK block per Section 6.2.3

2. Parses lock_nonce (first Nn octets) from the Encrypted-CEK field

3. Evaluates steps to derive the KEK per Section 5.7.1

4. Decrypts Encrypted-CEK to recover the 32-octet CEK

Encrypted-CEK size validation is specified in Section 6.2.

To decrypt content, the Decryptor reads the 32-octet salt from the start of the DATA block, then derives payload_key from the CEK, encryption_parameters, and salt per Section 5.7.3. For each block i, the Decryptor computes key_i = block_key(i) (Section 5.7.4). For NMR AEADs, the Decryptor derives nonce_base and computes nonce_i = nonce_base XOR uint64(i). For non-NMR AEADs, the Decryptor reads each nonce from the stored metadata. Each block is decrypted using key_i, its nonce, ciphertext, tag, and data_aad(i, is_final) per Section 5.7.9. The location of these components depends on the layout:

* Linear layout (Section 6.4.1): for non-NMR AEADs, each encrypted block contains nonce, ciphertext, and tag concatenated; for NMR AEADs, ciphertext and tag only.

* Aligned layout (Section 6.4.2): for non-NMR AEADs, nonces and tags are in the header metadata array; for NMR AEADs, only tags are stored. Ciphertext blocks are at aligned offsets.

See Appendix E for offset calculations.

Appendix C. Error Codes for Testing

This appendix is informative.

For interoperability testing, implementations MAY use the following error identifiers to categorize failures:

Error Code	Description	When to Emit
ERR_UNSUPPORTED_AEAD	Unknown AEAD algorithm	Parsing SAFE CONFIG
ERR_UNSUPPORTED_KEM	Unknown KEM identifier	Parsing hpke(...) step
ERR_INVALID_BLOCK_SIZE	Invalid Block-Size value	Parsing SAFE CONFIG
ERR_HPKE_NO_MATCH	No matching private key	Recipient discovery
ERR_HPKE_DECAP_FAILED	HPKE decapsulation error	CEK recovery
ERR_LOCK_AEAD_FAILED	Encrypted-CEK decryption failed	CEK recovery
ERR_PAYLOAD_AEAD_FAILED	Block decryption failed	Content decryption
ERR_BLOCK_OUT_OF_RANGE	Block index invalid	Content decryption
ERR_MALFORMED_BASE64	Base64 decoding error	Any Base64 field
ERR_DUPLICATE_FIELD	Repeated field name	Parsing SAFE CONFIG
ERR_DUPLICATE_PARAM	Repeated step parameter	Parsing Step lines
ERR_MISSING_SALT	pass(...) without salt	Parsing LOCK
ERR_MISSING_KEMCT	hpke(...) without kemct	Parsing LOCK
ERR_MULTIPLE_PASS_ONLY_LOCK	Multiple same-variant pass-only LOCKs	Discovery

ERR_NON_ASCII_HEADER	Non-ASCII in header	Parsing any header
ERR_RESOURCE_LIMIT	Size/count limit exceeded	Parsing any block
ERR_INVALID_SALT_LENGTH	salt not exactly 16 octets	Parsing salt
ERR_COMMITMENT_MISMATCH	Commitment verification failed	Key schedule
ERR_ACCUMULATOR_MISMATCH	Accumulator verification failed	Integrity check
ERR_TRUNCATION	Missing final block or block with is_final=0 at EOF	Content decryption

Table 19

Appendix D. Armored Data Arithmetic

This appendix is informative.

In armored mode, Decryptors compute the block count N and final block size from the Base64 payload length. Let S_b64 be the payload string between the fences, len_b64 its length in characters, pad the number of trailing = signs (0, 1, or 2), and len_bin_total = 3 * floor(len_b64 / 4) - pad. The header region is 32 (salt) + 32 (commitment) + Nh (accumulator) octets. The encrypted block region is len_bin_ciphertext = len_bin_total - 64 - Nh. Let B = Block-Size and Nn = AEAD nonce length. For NMR AEADs, nonces are derived (not stored), so the per-block overhead excludes Nn. Define:

```
if NMR:
    C      = B + 16
    C_min  = 16           # minimum: tag only
else:
    C      = Nn + B + 16
    C_min  = Nn + 16      # minimum: nonce + tag

N_nonfinal = floor( len_bin_ciphertext / C )
rem        = len_bin_ciphertext - N_nonfinal * C
if rem == 0:
    N       = N_nonfinal
    C_final = C
else if rem < C_min:
    reject as malformed
else:
    N       = N_nonfinal + 1
    C_final = rem

if NMR:
    L_final = C_final - 16
else:
    L_final = C_final - Nn - 16
```

A decryptor decrypting block index i computes octet offsets relative to the start of the encrypted block region (after salt, commitment, and accumulator):

```
block_byte_start = i * C
block_byte_len   = C if i < N - 1 else C_final
byte_start       = 64 + Nh + block_byte_start
byte_len         = block_byte_len
char_start       = 4 * floor( byte_start / 3 )
char_end         = 4 * ceil( (byte_start + byte_len) / 3 )
```

For each block index i, the Decryptor:

1. Extracts S_b64[char_start:char_end]

2. Base64-decodes to a temporary buffer tmp

3. Computes skip = byte_start mod 3

4. Selects encrypted_block = tmp[skip : skip + byte_len]

5. For non-NMR AEADs: parses nonce_i = encrypted_block[0:Nn] and ciphertext_i = encrypted_block[Nn:]. For NMR AEADs: ciphertext_i = encrypted_block and nonce_i = nonce_base XOR uint64(i) per Section 5.7.5

6. Determines is_final = 1 if i == N - 1 else 0

7. Constructs data_aad(i, is_final) per Section 5.7.9

8. Computes key_i = block_key(i) (Section 5.7.4)

9. AEAD-opens ciphertext_i under key_i with nonce_i and data_aad

Appendix E. Selective Decryption

This appendix is informative.

To decrypt block index i from a long object, a Decryptor first selects a candidate LOCK per Section 6.2.3. The decryptor constructs encryption_parameters from the SAFE CONFIG or from defaults, parses lock_nonce from Encrypted-CEK, evaluates the step sequence to derive the KEK, and opens Encrypted-CEK to recover the CEK. The decryptor reads the 32-octet salt from the start of the DATA block, then derives payload_key and acc_key from the CEK, encryption_parameters, and salt. It then locates block i in the payload. No other blocks need to be read or decoded.

For binary encoding, read N and D from the header, then compute block i’s ciphertext offset as (D + i) × B. The nonce and tag are at offset header_len + 64 + 8 + i × meta_len, where meta_len is Nn + 16 for non-NMR AEADs or 16 for NMR AEADs (Section 6.4.2).

For armored encoding, the Decryptor must compute the Base64 character window covering block i and decode only that window, as described below.

E.1. Example: Armored Selective Block Decryption

This example uses a non-NMR AEAD (AES-256-GCM, Nn=12). For NMR AEADs, omit Nn from per-block sizes (C = B + 16 instead of Nn + B + 16).

Consider Block-Size=16384, Nn=12, and three blocks: two full blocks plus a 5000-octet final block.

```
C       = Nn + B + 16 = 12 + 16384 + 16 = 16412 octets (full block)
C_final = Nn + 5000 + 16 = 12 + 5000 + 16 = 5028 octets (final block)
total_binary = 32 + 32 + 16412 + 16412 + 5028 + 32 = 37948 octets
Base64 len   = ceil(37948 / 3) * 4 = 50600 characters
```

To decrypt block i=0, compute octet and character offsets (salt, commitment, and accumulator occupy the first 96 octets):

```
byte_start  = 96 + (0 * C) = 96
byte_len    = C = 16412
char_start  = floor(byte_start / 3) * 4 = 128
char_end    = ceil((byte_start + byte_len) / 3) * 4 = 22012
skip        = byte_start mod 3 = 0
```

Extract S_b64[128:22012], Base64-decode, then take 16412 octets as the encrypted block (skip = 0, no leading octets to discard). Parse the first Nn octets (12 for AES-256-GCM) as the stored nonce, compute block_key(i) (Section 5.7.4), and AEAD-open the remaining octets under block_key(i) with the extracted nonce and block AAD. For NMR AEADs, derive the nonce from nonce_base instead (Section 5.7.5).

Appendix F. Design Rationale

This appendix is informative.

SAFE's design choices reflect trade-offs between flexibility, performance, and simplicity. This section explains the rationale behind key architectural decisions.

F.1. Two-Tier Key Hierarchy

SAFE separates the Content-Encryption Key (CEK) from the Key-Encryption Key (KEK) to enable multi-recipient encryption without duplicating payload ciphertexts.

F.1.1. Benefits

A single CEK is generated once and used to encrypt the payload; each recipient's LOCK derives a KEK that wraps the same CEK. This design offers several advantages:

1. Storage and bandwidth efficiency: Adding recipients requires only adding LOCK blocks (typically < 1 KB each), not duplicating the entire payload. For large files, this is critical.

2. Key rotation: Recipients can be added or removed by re-wrapping the CEK under new KEKs without re-encrypting the payload.

3. Operational flexibility: The CEK remains constant while KEKs rotate, simplifying key management.

F.1.2. Trade-offs

This design implies that all recipients share the same payload_key. Encryptors who require per-recipient payload keys (e.g., for fine-grained access control that survives CEK compromise) would need to encrypt multiple independent payloads.

If recipients directly decrypted the payload with their KEK, each recipient would require a distinct copy of the ciphertext, multiplying storage and bandwidth costs.

F.2. Minimal Block AAD

Block associated data is defined as Encode("SAFE-DATA", I2OSP(i, 8), I2OSP(is_final, 1)), binding each block to its position and finality. Suite parameters and LOCK-specific data are excluded from block AAD.

F.2.1. Rationale

This choice prioritizes simplicity and O(1) random access:

1. Selective decryption: payload_key already depends on encryption_parameters, so block AAD need not repeat them. This avoids requiring every block decryption to reference the SAFE CONFIG.

2. Multi-recipient caching: Including LOCK-specific data (Step lines, kemct) would couple block decryption to a specific LOCK, preventing efficient caching of payload_key across multiple recipients.

F.2.2. Security Properties

Suite and LOCK binding is indirect through the key hierarchy:

* The KEK schedule binds encryption_parameters at initialization and final derivation, with all step_tokens folded between; Encrypted-CEK AEAD authenticates the CEK under this KEK.

* The payload schedule binds payload_key to encryption_parameters via SafeDerive.

* Block AAD includes the block index and finality flag, preventing reordering, splicing, truncation, and extension within a file.

* The snapshot accumulator (Section 5.7.8) binds all block tags under acc_key, providing file-level integrity without per-block decryption. Each contribution is a PRF output under acc_key; XOR accumulation is unforgeable without knowledge of the CEK.

F.2.3. Alternative Designs Considered

An alternative design could include a "recipient_id" in block AAD, but this would require additional per-recipient metadata and complicate multi-recipient scenarios. SAFE's choice favors performance and simplicity for the common case of single-recipient or trust-equivalent multi-recipient files, while accepting that ciphertext blocks alone do not directly identify which LOCK unlocked them.

SAFE provides built-in truncation and extension detection via the is_final flag in block AAD (Section 5.7.9). The final block is marked with is_final=1; all preceding blocks use is_final=0. This design, inspired by the STREAM construction [STREAM], enables:

* Truncation detection: a block with is_final=0 and no successor indicates truncation

* Extension prevention: appending after is_final=1 fails AEAD verification

* Streaming writes: encryptors buffer the last block until the stream closes, then encrypt it with is_final=1

Per-block random nonces (Section 5.7.5) enable selective editing: individual blocks can be re-encrypted without affecting LOCK blocks or other blocks. This design trades a small storage overhead (Nn octets per block) for flexibility in payload modification.

F.3. Fixed HKDF Salt

SafeDerive (Section 5.4.1) uses a fixed string ("SAFE-v1") as the HKDF salt in its Extract step. The same string appears in the Encode-framed Input Keying Material (IKM), making its presence in the salt redundant but harmless. This design is intentional: the fixed salt binds every Extract call to the protocol version at the HMAC key position, while security relies on the entropy of the IKM, not the salt. All security-critical IKMs (CEK, step secrets, hedged random values) have sufficient min-entropy for this to hold. Krawczyk's analysis of HKDF shows that Extract with a fixed salt is a good randomness extractor when the IKM has high min-entropy. For the XOF single-stage variant, there is no separate salt; the version string appears only within the Encode frame.

Appendix G. Test Vectors

This appendix is informative.

This appendix provides a complete known-answer test for a passphrase-based SAFE encoding using default parameters (aes-256-gcm, Block-Size=65536, Hash=sha-256, Data-Encoding=armored). All values are hex unless noted.

```
## Inputs
Passphrase:    "correct horse battery staple" (UTF-8, 28 octets)
Salt:          01010101010101010101010101010101
CEK:           aaaaaaaa...aa (32 octets of 0xAA)
lock_nonce:    020202020202020202020202
payload_salt:  04040404...04 (32 octets of 0x04)
Plaintext:     "Hello, SAFE!" (12 octets)

## encryption_parameters
encryption_parameters = ["aes-256-gcm", "65536", "sha-256"]

## KEK Schedule
step_token:       000470617373000861726 76f6e326964
                  001001010101010101010101010101010101
    Encode("pass", "argon2id", salt)
step_secret:      7d3491ac8af1b54526792869b7257f5dbf7cc3c20929417bb193e396c51d7965
agg_init:         1b257512ce57328cbb04bbf80b4b3aa220d875832c8439c0cdda85e1e4f8428b
    SafeDerive("kek_init", "",
    encryption_parameters, 32)
agg_step:         596a483b938ad11da3369007f1b7f073502101879eb257f0f4b22c0758fdee21
    SafeDerive("kek_step",
    [agg, secret], token, 32)
derived_kek:      bfedcafd41d9da3c1c77f73358b973a4ececfbc212ae558eed0dfba709cdc24e
    SafeDerive("kek", agg,
    encryption_parameters, 32)

## Encrypted-CEK
aad:              "" (empty)
Encrypted-CEK:    0202020202020202020202352cbe85
                  a8e4434e5cd98d6507c80759dfe41fbe
                  13a649df57a9f7f46d1a7f90c60e1531
                  92ecb8c83a649656a6785487

## Payload Schedule
payload_info = [...encryption_parameters, payload_salt]
commitment:       42330a7379357f4f369f0271369546047f702ff37c53a8e17eb2342731683905
    SafeDerive("commit", CEK,
    payload_info, 32)
payload_key:      01a830b8a79a687b784109020b70d58dd53e3b51260d468c8c5ba05181ae09d8
```

```
    SafeDerive("payload_key", CEK,
    payload_info, 32)
acc_key:          9ce7a1a28f00e17c601b49ef3959a797088c8872ec7dd33f7b1258c0362da6ec
    SafeDerive("acc_key", CEK,
    payload_info, 32)

## Block Encryption (block 0, is_final=1)
raw_random:       030303030303030303030303
uint64(0):        0000000000000000
nonce0:           030303030303030303030303 (= raw_random XOR 0)
data_aad:         0009534146452d444154410008000000000000000000000101
    Encode("SAFE-DATA",
    I2OSP(0, 8), I2OSP(1, 1))
ciphertext+tag:   b42ba125cc5d376aef03f1ec3ecbc9c96c4264265f94dea4a1312bbd
encrypted_block:  030303030303030303030303b42ba125
                  cc5d376aef03f1ec3ecbc9c96c426426
                  5f94dea4a1312bbd

## Snapshot Accumulator
tag_0:            3ecbc9c96c4264265f94dea4a1312bbd
contrib_0:        4b4160f1af84bd74fbb1cf7fdccae69b2027c7ffdc67a7a03bc33c1b9489a4e8
    SafeDerive("acc_contrib", acc_key,
    [uint64(0), tag_0], 32)
accumulator:      4b4160f1af84bd74fbb1cf7fdccae69b2027c7ffdc67a7a03bc33c1b9489a4e8
    = contrib_0 (single block)

## Complete SAFE Object
SAFE DATA = salt (32) + commitment (32) + accumulator (32)
            + encrypted_block (40)
= 136 octets, 184 Base64 chars.
```

Readable format: ~~~~ -----BEGIN SAFE CONFIG----- Lock-Encoding: readable -----END SAFE CONFIG----- -----BEGIN SAFE LOCK----- Step: pass(kdf=argon2id, salt=AQEBAQEBAQEBAQEBAQEBAQ==) Encrypted-CEK: AgICAgICAgICAgICNSy+hajkQ05c2Y1lB8gHWd/kH74TpknfV6n39G0af5DGDhUx kuy4yDpkllameFSH -----END SAFE LOCK----- -----BEGIN SAFE DATA----- BAQEBAQEBAQEBAQEBAQEBAQEBAQEBAQEBAQEBARCMwpzeTV/TzafAnE2lUYE f3Av83xTqOF+sjQnMWg5BUtBYPGvhL10+7HPf9zK5psgJ8f/3GenoDvDPBuUiaTo AwMDAwMDAwMDAwMDtCuhJcxdN2rvA/HsPsvJyWxCZCZflN6koTErvQ== -----END SAFE DATA----- ~~~~

Same object in armored format (default Lock-Encoding). The armored LOCK body is Base64(Encode(step, ecek)) per Section 6.2.2:

```
-----BEGIN SAFE LOCK-----
ACIABHBhc3MACGFyZ29uMmlkABABAQEBAQEBAQEBAQEBADwCAgICAgICAgIC
AgI1LL6FqORDTlzZjWUHyAdZ3+QfvhOmSd9Xqff0bRp/kMYOFTGS7LjIOmSWVqZ4
VIc=
-----END SAFE LOCK-----
-----BEGIN SAFE DATA-----
BAQEBAQEBAQEBAQEBAQEBAQEBAQEBAQEBAQEBARCMwpzeTV/TzafAnE2lUYE
f3Av83xTqOF+sjQnMWg5BUtBYPGvhL10+7HPf9zK5psgJ8f/3GenoDvDPBuUiaTo
AwMDAwMDAwMDAwMDtCuhJcxdN2rvA/HsPsvJyWxCZCZflN6koTErvQ==
-----END SAFE DATA-----
```

Appendix H. X25519 Test Vector

This appendix is informative.

This appendix provides a complete known-answer test for a single X25519 recipient using DHKEM(X25519, HKDF-SHA256) per Section 5.2.2. All values are hex unless noted. The ephemeral and recipient keys are from RFC 9180, Appendix A.7.1 (DHKEM(X25519, HKDF-SHA256), HKDF-SHA256, AES-128-GCM), so the shared_secret value can be independently verified against that appendix.

```
## Inputs
Recipient skR:  4612c550263fc8ad58375df3f557aac531d26850
                903e55a9f23f21d8534e8ac8
Recipient pkR:  3948cfe0ad1ddb695d780e59077195da6c56506b
                027329794ab02bca80815c4d
Ephemeral skE:  52c4a758a802cd8b936eceea314432798d5baf2d
                7e9235dc084ab1b9cfa2f736
Ephemeral pkE:  37fda3567bdbd628e88668c3c8d7e97d1d1253b6
                d4ea6d44c150f741f1bf4431 (= enc)
CEK:            aaaaaaaa...aa (32 octets of 0xAA)
lock_nonce:     020202020202020202020202
payload_salt:   04040404...04 (32 octets of 0x04)
Plaintext:      "Hello, SAFE!" (12 octets)

## encryption_parameters
encryption_parameters = ["aes-256-gcm", "65536", "sha-256"]

## DHKEM(X25519, HKDF-SHA256)
dh = DH(skE, pkR):
  b3b5c19eab3f088ac18f23f774ff6414
  ba4fde45404d10085efc3e4dc9c72e35
kem_context = enc || pkR:
  37fda3567bdbd628e88668c3c8d7e97d
  1d1253b6d4ea6d44c150f741f1bf4431
  3948cfe0ad1ddb695d780e59077195da
  6c56506b027329794ab02bca80815c4d
```

```
   prk = LabeledExtract("", "eae_prk", dh):
     d8b4a0e70fe904de0f643c91e091903c
     f3c628994f51a025a7306bd4d4eb03c3
   shared_secret = LabeledExpand(prk,
       "shared_secret", kem_context, 32):
     fe0e18c9f024ce43799ae393c7e8fe8f
     ce9d218875e8227b0187c04e7d2ea1fc

   ## SAFE Step Secret
   key_id = SafeDerive("SAFE-SPKI-v1",
       [SPKI(pkR)], [""], 32):
     98cdd10b776ac15ed78f5520bed9f3e6
     ffdf682fe3ecb68163b4f1dd8b1dfefa
   step_token = Encode("hpke", "x25519",
       enc, key_id):
     000468706b650006783235353139
     002037fda3567bdbd628e88668c3c8d7
     e97d1d1253b6d4ea6d44c150f741f1bf
     4431002098cdd10b776ac15ed78f5520
     bed9f3e6ffdf682fe3ecb68163b4f1dd
     8b1dfefa
   exporter_context = SafeDerive(
       "SAFE-STEP", [token], [""], 32):
     3be7e9568086a415ada19b306534bc64
     6fcf0efc90e5faaaca358305c3ab7360
   exporter_secret = KeySchedule(Base,
       shared_secret, info="SAFE-v1"):
     3b2120e20d71e9e93c01b659c4835c72
     d0e43c660dca0dba8439b64cc78a9b2b
   step_secret = Export(exporter_secret,
       exporter_context, 32):
     42a4a3f299e1a71a97b04a3d9a7e9ae6
     7cd1b8ea3dec017e26fa1e369ee6f85b

   ## KEK Schedule
   agg_init:
     1b257512ce57328cbb04bbf80b4b3aa2
     20d875832c8439c0cdda85e1e4f8428b
   agg_step:
     16dfec67e5ab80a27b5c338de616c453
     c0660ed1c030dc29e6f4d46fadbad3a1
   derived_kek:
     c88d0730216dd222c4d64ffceac8be3e
     25815bdf067cc3a27760928ab0cc82f6

   ## Encrypted-CEK
   Encrypted-CEK:
     02020202020202020202020202
```

```
    8865cde5f682dcd6155b30ffbcd80bd9
    879d6663ac56b340dfc0e082e78f23ea
    a44944abc2e4cb1bd2fba5ebffd08a8f

  ## Payload Schedule
  payload_info = [...encryption_parameters, payload_salt]
  commitment:
    42330a7379357f4f369f0271369546047f702ff3
    7c53a8e17eb2342731683905
  payload_key:
    01a830b8a79a687b784109020b70d58dd53e3b51
    260d468c8c5ba05181ae09d8
  acc_key:
    9ce7a1a28f00e17c601b49ef3959a797088c8872
    ec7dd33f7b1258c0362da6ec

  ## Block Encryption (block 0, is_final=1)
  ciphertext+tag:
    b42ba125cc5d376aef03f1ec3ecbc9c9
    6c4264265f94dea4a1312bbd
  encrypted_block:
    03030303030303030303030303b42ba125
    cc5d376aef03f1ec3ecbc9c96c426426
    5f94dea4a1312bbd

  ## Snapshot Accumulator
  tag_0:
    3ecbc9c96c4264265f94dea4a1312bbd
  contrib_0:
    4b4160f1af84bd74fbb1cf7fdccae69b
    2027c7ffdc67a7a03bc33c1b9489a4e8
      SafeDerive("acc_contrib", acc_key,
      [uint64(0), tag_0], 32)
  accumulator:
    4b4160f1af84bd74fbb1cf7fdccae69b
    2027c7ffdc67a7a03bc33c1b9489a4e8
      = contrib_0 (single block)

  ## Complete SAFE Object
  SAFE DATA = salt (32) + commitment (32) + accumulator (32)
              + encrypted_block (40)
  = 136 octets, 184 Base64 chars.
```

Appendix I. Auth Mode Test Vector

This appendix is informative.

This appendix provides a known-answer test for Auth mode HPKE (X25519) with a single recipient. The recipient keys are the same as Appendix H. All values are hex unless noted.

```
## Inputs
Recipient skR:  4612c550263fc8ad58375df3f557aac531d26850
                903e55a9f23f21d8534e8ac8
Recipient pkR:  3948cfe0ad1ddb695d780e59077195da6c56506b
                027329794ab02bca80815c4d
Sender skS:     6b7298af684f45181f80ac5cb3d9a3713abb62cb
                ecd21db5dba0eb2a8bfb3a05
Sender pkS:     ccc340219b8098b48749f1c36e2c336faefb87f9
                cbe3463e59e3b8ec18c44c49
Ephemeral skE:  52c4a758a802cd8b936eceea314432798d5baf2d
                7e9235dc084ab1b9cfa2f736
Ephemeral pkE:  37fda3567bdbd628e88668c3c8d7e97d1d1253b6
                d4ea6d44c150f741f1bf4431 (= enc)
CEK:            aaaaaaaa...aa (32 octets of 0xAA)
lock_nonce:     020202020202020202020202

## encryption_parameters
encryption_parameters = ["aes-256-gcm", "65536", "sha-256"]

## DHKEM Auth Mode (X25519)
shared_secret:
  8c1cd5a4e40708af36bf6446d65f62ed
  7a7b7f1286d966d9bc96b96808021ba8

## SAFE Step Secret (Auth mode)
id:
  98cdd10b776ac15ed78f5520bed9f3e6
  ffdf682fe3ecb68163b4f1dd8b1dfefa
sid:
  d9b9d59d0f10a55a8365b2f440dbf787
  f280e0f400427beaaf17de1f8662fcdc
step_token = Encode("hpke", "x25519",
    enc, id, "auth", sid):
  000468706b650006783235353139
  002037fda3567bdbd628e88668c3c8d7
  e97d1d1253b6d4ea6d44c150f741f1bf
  4431002098cdd10b776ac15ed78f5520
  bed9f3e6ffdf682fe3ecb68163b4f1dd
  8b1dfefa00046175746800 20d9b9d59d
  0f10a55a8365b2f440dbf787f280e0f4
  00427beaaf17de1f8662fcdc
exporter_context:
  6da02fe479338d9a1798f8c02e09f601
  6dc3e0c49e2dff50c63ba273eabd2382
```

```
   exporter_secret = KeySchedule(Auth,
       shared_secret, info="SAFE-v1"):
     c35c97b436c388b766bd65008d1c7c5f
     0f52aad653b4072cbc8bb70e98c035bf
   step_secret:
     24c50a152128baaa3f77c255b0e5b895
     2cece2e7680b21bd7dc24aa1807e9374

   ## KEK Schedule
   agg_init:
     1b257512ce57328cbb04bbf80b4b3aa2
     20d875832c8439c0cdda85e1e4f8428b
   agg_step:
     2cb2cd8062e86b7dda6580d6bf8ba351
     25ef38644af50a8b1f6ab7eb53bae59c
   derived_kek:
     4a2b4a469f961e824c8c1681723393f0
     3937ede4bb824fc507200f932ef9c739

   ## Encrypted-CEK
   Encrypted-CEK:
     020202020202020202020202
     e29d2e1e0502fdbadaa1232a013a401c
     309e30dd7db132692a8dc03dfc64375d
     effa4696385b308f1093e41cf9f407f4
```

Appendix J. Multi-Block Test Vector

This appendix is informative.

This appendix provides a known-answer test with two blocks, demonstrating block index > 0, is_final transitions, and accumulator XOR of multiple contributions. Uses a passphrase LOCK with default parameters. All values are hex unless noted.

```
## Inputs
Passphrase:   "correct horse battery staple" (28 octets)
Salt:         01010101...01 (32 octets of 0x01)
CEK:          aaaaaaaa...aa (32 octets of 0xAA)
lock_nonce:   020202020202020202020202
payload_salt: 04040404...04 (32 octets of 0x04)
Block 0:      "Block zero data!" (16 octets)
Block 1:      "Final block." (12 octets, final)

## Step Secret
step_secret:
  2bab4c606fb5c84123e48f0bb5eeb2a1
  39a6dd996c6cff2efad2ad78a12143e0
```

```
## KEK Schedule
derived_kek:
  152c31bd033c76fc1d4987681b7ca76a
  8948514d6e5b7c2a5d8a4b681873a56f

## Encrypted-CEK
Encrypted-CEK:
  02020202020202020202020202
  3dbace72c0486e28026e6be8c1f0fc12
  dfc833d4c76b76a40dd68be3f423568c
  b4fb3d2ef43a2374f5a2fc0642e90cd1

## Payload Schedule
payload_info = [...encryption_parameters, payload_salt]
commitment:
  42330a7379357f4f369f027136954604
  7f702ff37c53a8e17eb2342731683905
payload_key:
  01a830b8a79a687b784109020b70d58d
  d53e3b51260d468c8c5ba05181ae09d8
acc_key:
  9ce7a1a28f00e17c601b49ef3959a797
  088c8872ec7dd33f7b1258c0362da6ec

## Block 0 (is_final=0)
nonce_0:          030303030303030303030303
data_aad_0:       0009534146452d44415441
                  000800000000000000000000100
    Encode("SAFE-DATA", I2OSP(0, 8), I2OSP(0, 1))
ciphertext+tag_0:
  be22a22ac8516d5cdc2a94a9863ced1c
  712ded5352105fddab8539c9570eda40
tag_0:            712ded5352105fddab8539c9570eda40

## Block 1 (is_final=1)
nonce_1:          050505050505050505050505
data_aad_1:       0009534146452d44415441
                  000800000000000000001000101
    Encode("SAFE-DATA", I2OSP(1, 8), I2OSP(1, 1))
ciphertext+tag_1:
  128cb7c8a035399b40d0a69d
  866cbbc0f49d8f85ce6b1883a0f0c028
tag_1:            866cbbc0f49d8f85ce6b1883a0f0c028

## Snapshot Accumulator (2 blocks)
contrib_0:
  c1ad6f915fa1babef344e5307b62bc7f
  33d6bf7269cdb84dbe5c76889204220d
```

```
   contrib_1:
     4862e6122d79464e22a033c2b8dd17e5
     c8922f35898f35638802ae4b9e4f0bb9
   accumulator = contrib_0 XOR contrib_1:
     89cf898372d8fcf0d1e4d6f2c3bfab9a
     fb449047e0428d2e365ed8c30c4b29b4
```

Appendix K. SafeDerive Test Vectors

This appendix is informative. These vectors validate the SafeDerive function in isolation, independent of the SAFE protocol. Inputs are deliberately short for readability. See Section 5.4.1 for the definition.

Common inputs for all vectors:

```
label:  "SAFE-TEST" (9 octets, hex: 534146452d54455354)
ikm:    0a0b0c0d0e0f (6 octets)
info:   "" (0 octets)
```

K.1. Hash=sha-256, L=32

```
Extract salt = "SAFE-v1":
  534146452d7631

Extract ikm = Encode("SAFE-v1",
    label, ...ikm):
  0007534146452d763100095341464 52d5445535400060a0b0c0d0e0f

prk = Extract(salt, ikm):
  983d59830192955caf33fff4056ed415e2cd1cef7fe3072e075cf90903c97146

Expand info = Encode("SAFE-v1",
    label, ...info,
    I2OSP(32, 2)):
  0007534146452d763100095341464 52d54455354000000020020

output = Expand(prk, info, 32):
  d7413c70bb7bde999f5e543c0796d63a0af6839ebbe5203cc526776b978ba147
```

K.2. Hash=sha-256, L=16

```
output = SafeDerive(label, ikm, info, 16):
  e190628e91995808047c49a7269b9d3b
```

K.3. Hash=turboshake256, L=32

```
Derive(Encode("SAFE-v1",
              label, ...ikm,
              I2OSP(32, 2), ...info), 32):
  input:  0007534146452d763100095341464 52d5445535400060a0b0c0d0e0f00020020
          0000
  output: 0394158419b3a24cb4f29376ada39b833f100a1825bfac7bc64f4f2ca9ff8a50
```

K.4. Hash=turboshake256, L=16

```
   output = SafeDerive(label, ikm, info, 16):
     f1c0c2cd38c6b0cbd0fa6a986b12c82d
```

Appendix L. Defining New Step Types

This appendix is informative.

New step types can be defined and registered to extend SAFE with additional authentication mechanisms. This section illustrates the process using three hypothetical steps: two Privacy Pass Key Derivation Function (PPKDF) steps for token-gated key derivation (Appendix L.1), and a WebAuthn PRF step for hardware token authentication.

L.1. Example: Privacy Pass Steps

Privacy Pass [RFC9578] type 0x0001 tokens use a Verifiable Oblivious Pseudorandom Function (VOPRF) [RFC9497]. The client constructs a TokenInput containing a nonce, blinds it, and sends the blinded element to the issuer. The issuer evaluates its PRF on the blinded element and returns the result. The client unblinds to obtain the authenticator, which is a deterministic function of the TokenInput and the issuer’s secret key.

In normal issuance the nonce is random, making each token unique. PPKDF sets the nonce to a deterministic value derived from the SAFE context, so the authenticator is reproducible and serves as step key material. The issuer’s behavior is unchanged and cannot distinguish a PPKDF request from normal token issuance.

Two step types use this mechanism:

* ppkdf: token-gated key derivation with application-supplied context.

* ppkdf-pass: token-gated key derivation with password-derived context for online throttling.

L.1.1. Shared Parameters

issuer (REQUIRED): Server name (host or host:port) of the Privacy Pass token issuer. The issuer holds the VOPRF key pair; its public key pkI must be known to the client.

salt (REQUIRED): Base64-encoded salt; must decode to exactly 32 octets. Generated at encryption time using SafeRandom.

The binding step_token uses the Encode form (Section 5.6):

* ppkdf: Encode("ppkdf", issuer, salt)

* ppkdf-pass: Encode("ppkdf-pass", issuer, kdf, salt)

where issuer and kdf are UTF-8 strings and salt is the raw decoded 32 octets.

L.1.2. ppkdf Step

The ppkdf step provides token-gated key derivation with application-supplied context.

Token form:

ppkdf(issuer=tokens.example.com,salt=<Base64>)

Grammar:

```
ppkdf-step    = "ppkdf(" ppkdf-params ")"
ppkdf-params  = "issuer=" pp-host "," "salt=" salt
pp-host       = host [ ":" port ]
host          = 1*( ALPHA / DIGIT / "-" / "." )
port          = 1*DIGIT
salt          = 1*BASE64CHAR    ; 44 chars = 32 octets
```

Derivation:

Decode salt to salt_bytes (32 octets). Compute a deterministic nonce for the TokenInput:

```
nonce = SafeDerive("SAFE-PPKDF",
    "", [...encryption_parameters,
         binding_step_token, salt_bytes], 32)
```

where binding_step_token is the Encode-based binding form defined in Section 5.6. Construct a type 0x0001 TokenInput ([RFC9578], Section 5.1) with nonce as above. Set challenge_digest to

SafeDerive("SAFE-PPKDF", TokenChallenge, encryption_parameters, 32), using a TokenChallenge with issuer_name = issuer and empty redemption_context and origin_info.

Blind the TokenInput and send in a standard type 0x0001 TokenRequest. The issuer evaluates the VOPRF and returns a TokenResponse. Verify the VOPRF proof and unblind to obtain the authenticator (32 octets). Set step_secret = authenticator.

L.1.3. ppkdf-pass Step

The ppkdf-pass step adds password-derived input. Each password guess requires a VOPRF evaluation from the issuer.

Token form:

ppkdf-pass(issuer=tokens.example.com,kdf=argon2id,salt=<Base64>)

Grammar:

```
ppkdf-pass-step  = "ppkdf-pass(" ppkdf-pass-params ")"
ppkdf-pass-params = "issuer=" pp-host ","
                    "kdf=" kdf-name "," "salt=" salt
kdf-name          = "argon2id" / "pbkdf2"
```

The kdf parameter selects the password KDF using the same algorithms and default parameters as the pass() step (Section 5.6.2).

Derivation:

Decode salt to salt_bytes (32 octets). Derive pw32 from the user's password using the KDF indicated by the kdf parameter, with salt_bytes as salt input and the default parameters defined in Section 5.6.2. Compute the deterministic nonce:

```
nonce = SafeDerive("SAFE-PPKDF-PASS",
    pw32, [...encryption_parameters,
           binding_step_token], 32)
```

Execute the PPKDF protocol as in the ppkdf step using this nonce. Set step_secret = authenticator.

L.1.4. IANA Registry Entries

Registration of ppkdf and ppkdf-pass is deferred to a separate document; this appendix is informative. A registration for these steps would include:

Step Name	Parameters	Inputs	Secret	Ref
ppkdf	issuer=X, salt=X	PPKDF token	32 octets	(this doc)
ppkdf-pass	issuer=X, kdf=X, salt=X	PPKDF token, password	32 octets	(this doc)

Table 20

L.1.5. Security Considerations for Privacy Pass Steps

For ppkdf:

The issuer sees only the blinded_element. It cannot learn context, step_secret, or anything about the SAFE file. The VOPRF proof lets the client detect the wrong issuer key. Compromise of skI enables offline computation of step_secret for any context, breaking the online-gating property.

For ppkdf-pass:

Each password guess requires a VOPRF evaluation; issuer rate limits control guessing frequency. The issuer never sees the password-derived context. Compromise of skI still requires inverting the memory-hard KDF to recover the password.

Example LOCK block:

```
-----BEGIN SAFE CONFIG-----
Lock-Encoding: readable
-----END SAFE CONFIG-----
-----BEGIN SAFE LOCK-----
Step: ppkdf(issuer=tokens.example.com,salt=<Base64>)
Encrypted-CEK:
  Base64-encoded encrypted CEK
-----END SAFE LOCK-----
```

L.2. Example: WebAuthn PRF Step

A WebAuthn-based step would allow hardware token authentication using the PRF extension defined in Web Authentication Level 3 [W3C.webauthn-3]. Unlike WebAuthn assertions (signatures), the PRF extension provides deterministic output suitable for SAFE's step model.

L.2.1. Step Definition

Step name: webauthn-prf

The webauthn-prf step token has three forms:

```
webauthn-prf(rpid=example.com,salt=xyz...)           ; Identified RP
webauthn-prf(salt=xyz...)                            ; Anonymous RP
webauthn-prf(rpid=example.com,salt=xyz...,label=YubiKey) ; With label
```

The parameters are:

rpid (OPTIONAL): The WebAuthn relying party identifier. When present, the Decryptor uses this rpId for the WebAuthn ceremony. When omitted, selects anonymous RP mode.

salt (REQUIRED): The Base64-encoded PRF salt; must decode to exactly 32 octets. Generated at encryption time using SafeRandom.

label (OPTIONAL): A human-readable display name for this credential (e.g., "YubiKey", "Phone"). Not included in the binding step_token; see Section 5.6.

Credential selection is delegated to the authenticator via WebAuthn's allowCredentials mechanism. The Decryptor passes all candidate credential IDs for the rpId; the authenticator selects the matching credential internally.

Grammar:

```
webauthn-prf-step = "webauthn-prf(" webauthn-params ")"
webauthn-params   = [ "rpid=" rpid "," ] "salt=" salt
                    [ "," "label=" label-value ]
rpid              = 1*( ALPHA / DIGIT / "-" / "." )
salt              = 1*BASE64CHAR    ; 44 chars = 32 octets
label-value       = 1*( ALPHA / DIGIT / "-" )
```

Anonymous RP mode: When rpid is omitted from the token, the Decryptor tries each rpId for which it holds credentials. Each rpId requires a separate WebAuthn ceremony (and potentially a user prompt). Privacy benefit: hides the relying party from passive observers. Cost: one ceremony per candidate rpId.

Derivation: The authenticator evaluates the PRF extension with the selected credential and the decoded salt:

```
      prf_salt = decode(salt)      ; 32 octets

      prf_output = WebAuthn_PRF(credential, prf_salt)

      step_secret = SafeDerive(
          "webauthn-prf", prf_output,
          encryption_parameters, 32)
```

Inputs: credential (local, selected by authenticator for the rpId), prf_salt (from token).

Encode form: Encode("webauthn-prf", rpid, salt). rpid is UTF-8; salt is the raw decoded 32 octets. rpid is always present in the binding form even when omitted on-wire; when omitted, the Encryptor or Decryptor uses the rpId from the WebAuthn ceremony. Label is not included in binding.

Validation: salt must decode to exactly 32 octets. rpid, when present, must match the hostname grammar 1*(ALPHA / DIGIT / "-" / ".").

Example LOCK:

```
-----BEGIN SAFE CONFIG-----
Lock-Encoding: readable
-----END SAFE CONFIG-----
-----BEGIN SAFE LOCK-----
Step: webauthn-prf(rpid=example.com,salt=xyz...)
Encrypted-CEK:
  Base64-encoded encrypted CEK
-----END SAFE LOCK-----
```

L.2.2. IANA Registry Entry

Step Name	Parameters	Inputs	Secret	Ref
webauthn-prf	rpid=X, salt=X	Credential, rpId	32 octets	(this doc)

Table 21

L.2.3. Security Considerations for WebAuthn PRF Step

The WebAuthn PRF step provides hardware-bound key material (the authenticator holds the secret), user presence verification (touch required), phishing resistance (rpid binding), and offline decryption capability once the PRF output is computed.

The PRF extension requires WebAuthn Level 3 support in the browser. Non-discoverable credentials wrap key material in the credential ID; losing the credential ID means losing access to the encrypted file. Unlike the Privacy Pass steps, no server-side rate limiting is possible.

Privacy: rpid in cleartext reveals the relying party to passive observers. Anonymous RP mode (rpid omitted) hides this but the rpId may still be guessable from context.

Trial bounds: anonymous RP mode requires one WebAuthn ceremony per candidate rpId. Decryptors should impose a local bound on the number of rpIds to try. Prefer identified mode when privacy is not required.

Author’s Address

Nick Sullivan
Cryptography Consulting LLC
Email: nicholas.sullivan+ietf@gmail.com