Cryptographic Facilities in Unicon

Jafar Al-Gharaibeh

Unicon Technical Report: 27

August 2026

Abstract

Unicon has long used OpenSSL for encrypted network sockets, but hashing, HMAC, signing, and symmetric encryption were not available as language primitives. This report describes the facilities that close that gap: cryptographic operations reached through the same open(), read(), write(), and close() interface already used for files and sockets, with "[ ]" for status and Attrib() for idle-window mutation. Mode letter "e" is reused as a modifier; "op=" selects the operation; keys and certificates are loaded as file-typed handles and passed as attributes. Encrypted TCP is unchanged; encrypted UDP uses DTLS. The implementation is the OpenSSL EVP interface already linked for TLS.

Keywords: Unicon, cryptography, OpenSSL, TLS, DTLS, hash, HMAC, sign, encrypt, runtime, technical report.

Unicon Project
http://unicon.org
University of Idaho
Department of Computer Science
Moscow, ID, 83844, USA

1 1. Introduction

This report describes the cryptographic facilities in the Unicon runtime: how a handle is opened, how payload flows through `write()` / `read()`, how key and certificate material is loaded and reused, and how the same letters and attributes compose with encrypted sockets. It is formatted as a Unicon Technical Report [1]. The language-facing reference lives in *Programming with Unicon* (Chapter 6) and the language reference (`open` modes `e` / `er` / `eh` / `re` / `we`, and `ne` / `nue`, subscript peek, `key()`). Automated tests live under `tests/crypto/`.

The feature is optional in the same sense as TLS. A build with OpenSSL reports `secure sockets layer encryption` in `&features` (the `_SSL` flag). Without the library, the crypto paths are compiled out.

2 2. Motivation

Unicon already links OpenSSL [2] for mode `n` with `e` – encrypted TCP. Programs that needed a SHA-256 digest, an HMAC, a signature, or a file encrypted at rest had to use external tools. The facilities here expose those primitives without a second API: hashing a stream is opening a handle and writing to it; reading an encrypted file is opening the file with a transform on the way through.

No new external dependency is introduced. The implementation uses OpenSSL's EVP interface, which the encrypted-socket support already requires.

3 3. Design principles

`e` is the crypto letter, reused not invented. Unicon already uses `e` for encryption (`Fs_Encrypt`). New modes carry that letter rather than adding one letter per operation. `op=` selects what the handle does.

`e` composes as a modifier. Mode letters already stack, and several already branch on bits accumulated so far: `n` + `e` is an encrypted socket; `r` + `e` / `w` + `e` is a file transform; `e` + `h` is an in-memory hash; `e` + `r` is raw key material rather than a path.

The open target configures the operation; payload is I/O. Target is never the message being hashed or encrypted. What target means depends

on the mode (algorithm name, key path, data file, host:port). Payload always flows through `write()` / `read()` / `reads()`.

Attributes are `name=value` strings, or handles. A string element is parsed as always. A **handle** – the return value of a prior `open(..., "e")` – is a typed attribute that could not survive being flattened into a string. Handles are interpreted by **role** from their own content (private key, public key, cert, symmetric key), not by position or by an attribute name.

Material comes from a file, raw bytes, or a handle. A file path is the target under mode `e`, or `key=` when the target is already spoken for (`re` / `we`, `n` + `e`). Raw bytes use mode `er`. A handle avoids re-parsing.

Initialization is lazy. `open()` does not create the underlying EVP context until the first `write()` (or the first transforming `read()` on a file). A handle that is never used costs nothing. `read()` finalizes and resets, so the next `write()` initializes again.

`Attrib()` **reconfigures only in the idle window** – before the first `write()`, or after a `read()` and before the next `write()`. Changing `op=`, `alg=`, `cipher=`, or `iv=` mid-operation fails. Handles are accepted by `open()`.

4 4. Modes

`open()`'s mode string is scanned character by character in `src/runtime/fsys.r`. Letters are not independent: `u` after `n` is UDP, `r` after `n` is raw, `h` after `e` is hash. Order matters. `e` already set `Fs_Encrypt`; this design adds `Fs_Crypto` for hash / HMAC / sign / verify / encrypt-decrypt handles so they are distinct from TLS sockets that also carry `Fs_Encrypt`.

Mode	Meaning	Target
`e`	Load material, or an operation whose target is material	Key/cert path, or a material handle
`er`	Same, but the target is raw bytes	In-memory string
`eh`	Hash; no `op=` needed	Digest name, e.g. `sha256`
`re`	Read a file, transformed (hash or decrypt)	Data-file path

Mode	Meaning	Target
`we`	Write a file, transformed (encrypt)	Data-file path
`ne`	Encrypted TCP (TLS) – unchanged	`host:port`
`nue` / `naue`	Encrypted UDP (DTLS)	`host:port`

`e` / `er` are the **data-pipe** shape: write payload in, read the result out. `eh` is separate because its target is an algorithm name, not material. `re` / `we` are the **file-transform** shape: the target is the data file, so key material arrives via `key=` or a handle attribute.

`h` alone remains SSH. `r` alone remains read. The new meanings apply only when `e` was seen first.

Mode `-` skips peer verification on TLS (`ne-`) and SSH (`h-`), the same as `verifyPeer=no`. An explicit `verifyPeer=` overrides the mode flag.

5 5. Operations

`op=` names the operation when the mode does not already imply one. Loading material (`e` / `er` with a file or raw target and no `op=`), hashing in memory (`eh`), and TLS / DTLS sockets take no `op=`.

`op=`	What it does	Typical mode	Other attributes
`hash`	Digest of a file	`re`	`alg=`
`hmac`	Keyed digest (RFC 2104)	`e` / `er`	`alg=`
`sign`	Produce a signature	`e` / `er`	`alg=`
`verify`	Check a signature	`e` / `er`	`alg=`, `sig=`
`encrypt`	Symmetric encrypt	`e` / `er`, or `we`	`cipher=`; `key=` or a handle for `we`
`decrypt`	Symmetric decrypt	`e` / `er`, or `re`	`cipher=`; `key=` or a handle for `re`

HMAC is the construction of RFC 2104 [3] over the digest named by `alg=`. Algorithm and cipher names are OpenSSL's own (`sha256`, `aes-256-gcm`) and are passed through rather than mapped to a Unicon-defined set.

Defaults are `alg=sha256` and `cipher=aes-256-gcm`.

6 6. Attributes

Trailing `open()` arguments are `name=value` strings or crypto handles. Empty values and unknown names fail (TLS attribute errors use 1302; crypto state/name mistakes use 1316).

6.1 6.1 TLS / DTLS attributes

These are the names `create_ssl_context()` accepts. They apply to `ne` / `nue` and are unchanged except for the two alignments in section 6.3.

Attribute	Meaning
`cert=`	Certificate file path
`key=`	Private key file path
`keypass=`	Passphrase for an encrypted private key
`ca=` / `caDir=` / `caStore=`	Trust store
`ciphers=` / `ciphers1.3=`	TLS 1.2 list (e.g. `HIGH`, `ECDHE-RSA-AES256-GCM-SHA384`)
`minProto=` / `maxProto=`	Protocol version bounds, e.g. `TLS1.2`, `TLS1.3`
`verifyPeer=`	`yes` / `no`; `no` is the same as mode `-`

Socket attributes (`reuseaddr`, `ttl`, `iface`, `join`, `proto`, ...) are applied separately and are unaffected. Live-session status is section 6.4.

6.2 6.2 Crypto attributes

Attribute	Meaning	Used with
`op=`	Operation: `hash`, `hmac`, `sign`, `verify`,	All data-pipe and file-transform ops

Attribute	Meaning	Used with
`alg=`	Digest name passed to OpenSSL (default `sha256`;	`hash`, `hmac`, `sign`, `verify`
`cipher=`	Symmetric cipher (default `aes-256-gcm`; also	`encrypt`, `decrypt`
`iv=`	Explicit IV; omit for automatic IV	`encrypt`, `decrypt`
`sig=`	Signature being checked	`verify`
`key=`	Key path when the target is already a data file or host	`re` / `we`, and TLS
`type=`	Narrow a multi-item file: `cert`, `key`, `pubkey`	Material load (`e` / `er`)
`keypass=`	Passphrase for an encrypted key file	Material load

`key=` on a crypto handle is typed by content and by the operation: PEM / DER private keys for `sign` and TLS; raw bytes for `encrypt`, `decrypt`, and `hmac`. A role mismatch fails.

Without `iv=`, `encrypt` generates a fresh random IV per operation and prepends it to the ciphertext; `decrypt` reads that prefix. Supplying `iv=` opts out – useful for a wire format that does not prepend an IV, or for test vectors – and hands uniqueness back to the caller. The default cipher is AEAD; the runtime appends the authentication tag after the ciphertext. Status peeks are section 6.4.

6.3 6.3 Shared names with SSH

`key=` is a private-key path on TLS, crypto, and SSH. `keypass=` is the shared name for a key-file passphrase. SSH

`verifyPeer=yes|no` and mode `-` mean the same thing on TLS and SSH. An explicit `verifyPeer=` overrides the mode flag. Trust stores stay distinct: TLS uses `ca=` (X.509); SSH uses `hostkeyfile=` (OpenSSH `known_hosts`).

6.4 6.4 Status peek ([ ] and key())

h["name"] / conn["name"] is a non-destructive get. Attrib() only assigns in the idle window (op=, alg=, sig=, ...). Unknown names raise 1302 on a crypto handle and 1326 on a TLS socket. An unpopulated field fails. A boolean field that *did* answer succeeds: "yes" for true, &null for false , which would make things like if \h["expired"] succeed on a valid certificate. key(h) generates every *answerable* field, including false booleans, so any k from key(h) makes h[k] succeed. Dump with image(h[k]) so &null is visible.

h["*"] returns a table of those same fields captured under one lock. No status field is named *. Peeking a closed handle is error 174. key(h) that has not yet produced a name also raises 174 if the handle is already closed. After the first suspend, close() makes the generator fail instead of raising, so a walk does not turn a mid-generation close into an error.

h["type"] is a role label (key, cert, key,cert), not a list. h["san"] and conn["san"] are lists of SAN strings (DNS:host, IP:1.2.3.4). conn["certchain"] is a list of PEM strings. A single item is still a list of length 1. h["op"] is always populated on an operation handle.

Material (mode e / er, no op=).

Name	Value
type	key, pubkey, cert, symkey, or a
alg	Public-key algorithm: rsaEncryption,
keysize	Size in bits (integer), e.g. 2048, 256
subject / issuer	Certificate DN, OpenSSL oneline
san	List of Subject Alternative Names (DNS:localhost,
notbefore / notafter	Validity timestamps (2024-01-01T00:00:00Z)
expired	"yes" or &null
fingerprint	SHA-256 of the cert or public key (hex)

Certificate names fail on a key-only handle.

Hash (eh, or op=hash).

Name	Value
`op`	`hash`
`alg`	Digest name: `SHA256` (default), `SHA512`, `SHA1`,
`hash`	Current digest / MAC / signature bytes without
`bytecount`	Bytes written so far (integer)
`blocksize` / `digestsize`	Algorithm sizes (integers); SHA-256 is 64 / 32

HMAC, sign, verify.

Name	Value
`op`	`hmac`, `sign`, or `verify`
`hash`	Running HMAC or signature bytes (same copy-and-finalize
`alg`	Digest used for HMAC or for the signature: `SHA256`,
`bytecount`	Bytes written (integer)
`verified`	`"yes"` or `&null` after a verify; unpopulated before

Encrypt / decrypt.

Name	Value
`op`	`encrypt` or `decrypt`
`cipher`	Symmetric cipher: `AES-256-GCM` (default),
`iv`	IV that was set or generated (bytes)
`bytecount`	Bytes processed (integer)

TLS socket (after handshake). These are not `open()` attributes.

Name	Value
`cipher`	Negotiated suite: `TLS_AES_256_GCM_SHA384`,

Name	Value
`version`	`TLSv1.3`, `TLSv1.2`
`alpn`	Selected protocol: `h2`, `http/1.1`; unpopulated if none
`peercert`	Peer certificate as PEM
`certchain`	List of PEMs
`subject` / `issuer`	Peer DN, e.g. `/CN=host.example`
`san`	Peer SANs as a list (`DNS:host.example`)
`notbefore` / `notafter` / `expired`	Peer validity; `expired` is `"yes"` or `&null`
`certverified`	`"yes"` or `&null`. Always populated: with
`verifyresult`	`ok`, `expired`, `self-signed`, `untrusted-CA`,
`hostnamematch`	`"yes"` or `&null`: did CN/SAN match the requested

Handshake failure means `open()` failed; there is no handle to subscript. Distinguish expired / untrusted CA / hostname mismatch / no shared cipher / protocol version via `&errornumber` 1320–1325.

```
cert := open("client.crt", "e") | stop(&errortext)
write("material ", cert["type"], " ", cert["subject"])
every name := !cert["san"] do
   write("  SAN ", name)
every k := key(cert) do
   write("  ", k)

h := open("sha256", "eh") | stop(&errortext)
write(h, "abc")
write("op=", h["op"], " peek hash *", *h["hash"])
close(h)

conn := open("host:443", "ne", "ca=unicon-ca.crt") |
↪  stop(&errortext)
write(conn["cipher"], " ", conn["version"])
```

```
every pem := !conn["certchain"] do
   write(*pem, " byte PEM")
close(conn)
```

7 7. Material handles

Mode `e` with no `op=` loads keys and certificates. Content is auto-detected (PEM by header, DER by probe-by-parse). A file that holds several items – a combined cert+key PEM, or a cert plus chain – loads into one handle. `type=` narrows that to a single role.

```
k  := open("signing.pem", "e")
pk := open("verify.pem", "e")
c  := open("client.pem", "e")
kb := open(key_bytes, "er")
ek := open("enc.key", "e", "keypass=" || pw)
c  := open("bundle.pem", "e", "type=cert")
```

A raw key is still a file value; `h["type"]` reports the role:

```
procedure main()
   local k
   k := open("sixteen-byte-key-material-here!!", "er") |
      stop(&errortext)
   write(type(k))
   write(k["type"])
   close(k)
end
```

Output:

```
file
symkey
```

Material split across files can be merged:

```
idh := open("client.pem", "e")
Attrib(idh, "key=client.key")
```

A handle is a file value (`type(h)` is `"file"`) carrying `Fs_Crypto`, the same way a window or socket is a file that does not support every file operation. The underlying `CryptoFile` is malloc'd so pointers stay stable across garbage collection and are freed by `close()`.

Consumers ask a handle for the roles they need. An encrypted socket wants a cert and a private key: one merged handle satisfies both, or two handles each contribute what they have. `op=sign` wants a private key; `op=hmac` wants symmetric bytes. A missing role, or two handles supplying the same role, fails at `open()`.

8 8. Examples

These assume a build with `secure sockets layer encryption`. Failures set `&errortext`; the usual form is `h := open(...) | stop(&errortext)`. Package `crypto` (`uni/lib/crypto.icn`) supplies `hexencode` / `hexbytes` and base64 helpers for dumping binary digests and keys. The programs under `tests/crypto/` are the executable form of this section.

8.1 8.1 Hash

SHA-256 of the known vector `abc`:

```
import crypto

procedure main()
   local h, digest
   h := open("sha256", "eh") | stop(&errortext)
   write(h, "abc")
   digest := read(h)
   write(hexbytes(digest, 1))
   close(h)
end
```

Output:

```
ba7816bf8f01cfea414140de5dae2223b00361a396177a9cb410ff61f20015ad
```

`write()` may be called many times; `read()` finalizes the digest **and resets** the handle, so the same handle can hash unrelated messages. Chunked `"hel"` + `"lo"` matches a single write of `"hello"`. A read with nothing pending fails:

```
import crypto

procedure main()
   local h, d1, d2
   h := open("sha256", "eh") | stop(&errortext)
   write(h, "hello")
   d1 := hexbytes(read(h), 1)
   write(h, "hel")
   write(h, "lo")
   d2 := hexbytes(read(h), 1)
   write(d1)
```

```
   write(d2)
   if read(h) then write("idle:fail") else write("idle:ok")
   close(h)
end
```

Output:

```
2cf24dba5fb0a30e26e83b2ac5b9e29e1b161e5c1fa7425e73043362938b9824
2cf24dba5fb0a30e26e83b2ac5b9e29e1b161e5c1fa7425e73043362938b9824
idle:ok
```

A whole file uses the file-transform shape. The first `read()` consumes the file and returns the digest:

```
import crypto

procedure main()
   local f, h, digest
   f := open("hash.dat", "w") | stop(&errortext)
   writes(f, "abc")
   close(f)
   h := open("hash.dat", "re", "op=hash", "alg=sha256") |
      stop(&errortext)
   digest := read(h)
   write(hexbytes(digest, 1))
   close(h)
   remove("hash.dat")
end
```

Output:

```
ba7816bf8f01cfea414140de5dae2223b00361a396177a9cb410ff61f20015ad
```

8.2 8.2 HMAC

Raw key bytes use mode `er`. `read()` returns the MAC; the same handle can MAC many messages after each finalize:

```
import crypto

procedure main()
   local hm, mac
   hm := open("secret-key-bytes", "er", "op=hmac", "alg=sha256") |
      stop(&errortext)
   write(hm, "message one")
   mac := read(hm)
   write(hexbytes(mac, 1))
   write(*mac)
   close(hm)
end
```

Output:

```
9fd1a900325bd173af31bf6467de8139d6615202654cf0fe2645cd2997eaf879
32
```

A key loaded from a file is the same, with the material handle as the `open()` target and `op=hmac` as an attribute:

```
k  := open("hmac.key", "e")
hm := open(k, "e", "op=hmac", "alg=sha256")
```

8.3 8.3 Sign and verify

A private key signs; a certificate or public key verifies. `read()` on the signer returns the signature bytes. The listing assumes `signing.pem` and `verify.pem` in the current directory:

```
procedure main()
   local k, sg, pk, vf, signature
   k := open("signing.pem", "e") | stop(&errortext)
   sg := open(k, "e", "op=sign", "alg=sha256") | stop(&errortext)
   write(sg, "hello")
   signature := read(sg) | stop(&errortext)
   close(sg)

   pk := open("verify.pem", "e") | stop(&errortext)
   vf := open(pk, "e", "op=verify", "alg=sha256",
      "sig=" || signature) | stop(&errortext)
   write(vf, "hello")
   if read(vf) then write("ok") else stop("bad signature: ",
      &errortext)
   close(vf)
   close(k)
   close(pk)
end
```

Output:

```
ok
```

A failed verify sets `&errornumber` 1315 so it is distinct from "nothing written." The same handle can check many messages by setting `sig=` in the idle window:

```
procedure main()
   local k, sg, pk, vf, signature, msg
   k := open("signing.pem", "e") | stop(&errortext)
   sg := open(k, "e", "op=sign", "alg=sha256") | stop(&errortext)
   pk := open("verify.pem", "e") | stop(&errortext)
   vf := open(pk, "e", "op=verify", "alg=sha256") |
      stop(&errortext)
   every msg := !["one", "two"] do {
      write(sg, msg)
      signature := read(sg) | stop(&errortext)
      Attrib(vf, "sig=" || signature)
      write(vf, msg)
      if read(vf) then write(msg, ":ok") else write(msg, ":fail")
      }
   every close(sg | vf | k | pk)
end
```

Output:

```
one:ok
two:ok
```

8.4 8.4 Symmetric encrypt and decrypt

The idle-window `Attrib(..., "op=decrypt")` reuses the bound key. Ciphertext is binary; the recovered plaintext is what matters:

```
procedure main()
   local ky, c, ct, pt
   ky := open("sixteen-byte-key-material-here!!", "er") |
      stop(&errortext)
   c := open(ky, "e", "op=encrypt") | stop(&errortext)
   write(c, "hello")
   ct := read(c)
   Attrib(c, "op=decrypt")
   write(c, ct)
   pt := read(c)
   write(pt)
   close(c)
   close(ky)
end
```

Output:

```
hello
```

With no `iv=`, each `read()` generates a fresh IV and prepends it, so one handle can encrypt many messages. Lengths below include the IV and AEAD tag:

```
procedure main()
   local ky, c, msg, ct
   ky := open("sixteen-byte-key-material-here!!", "er") |
      stop(&errortext)
   c := open(ky, "e", "op=encrypt") | stop(&errortext)
   every msg := !["alpha", "beta", "gamma"] do {
      write(c, msg)
      ct := read(c)
```

```
      write(msg, " ", *ct, " bytes")
      }
   close(c)
   close(ky)
end
```

Output:

```
alpha 33 bytes
beta 32 bytes
gamma 33 bytes
```

For a wire format that does not prepend an IV, or for test vectors, set `iv=` in the idle window and take uniqueness yourself:

```
every plaintext := !messages do {
   Attrib(c, "iv=" || newiv())
   write(c, plaintext)
   ciphertext := read(c)
   }
```

8.5 8.5 File encrypt and decrypt

Writes encrypt and are finalized on `close()`. Reads decrypt as a stream. AEAD authentication is checked on the **final** read: earlier `reads()` chunks are provisional. Nothing decrypted should be acted on until the stream completes without error.

```
procedure main()
   local ky, f, chunk, pending
   ky := open("sixteen-byte-key-material-here!!", "er") |
      stop(&errortext)
   f := open("secrets.dat", "we", "op=encrypt", ky) |
      stop(&errortext)
   writes(f, "classified")
   close(f)
   f := open("secrets.dat", "re", "op=decrypt", ky) |
      stop(&errortext)
   pending := ""
   while chunk := reads(f, 4096) do
      pending ||:= chunk
   if
&errornumber then
      stop("tampered: ", &errortext)
   write(pending)
   close(f)
   remove("secrets.dat")
   close(ky)
end
```

Output:

```
classified
```

Callers who cannot defer processing should use the data-pipe shape, where a single `read()` returns plaintext only after the tag verifies.

The same key handle can wrap many files:

```
keyh := open("aes.key", "e")
every name := !filenames do {
```

```
   f := open(name || ".enc", "we", "op=encrypt", keyh)
   write(f, contents(name))
   close(f)
   }
```

8.6 8.6 TLS identity reuse and DTLS

Encrypted TCP remains `"ne"`. Material handles may be passed as attributes so certs and keys are parsed once. The socket asks by role, not by position: one combined PEM, or two handles, both work.

```
id := open("client.pem", "e")
conn := open("host:443", "ne", id, "verifyPeer=no")
conn := open("host:443", "ne-", id)    # same as verifyPeer=no

certh := open("client.pem", "e")
keyh  := open("client.key", "e")
conn  := open("host:443", "ne", certh, keyh)
```

After a handshake, `conn["cipher"]` peeks the negotiated suite and `conn["certchain"]` is a list of peer certificates as PEM strings; see section 6.4 (`tests/crypto/tlspeek.icn`). `certh["type"]` / `keyh["type"]` are material-handle peeks, not session fields.

Encrypted UDP (DTLS) uses `"nue"` / `"naue"`. The runtime selects a DTLS context when the socket is datagram.

9 9. I/O and `Attrib()`

On a data-pipe handle, `write()` accumulates and `read()` finalizes. `read()` is always terminal – one read consumes the whole result and resets the handle – for two reasons that are easy to forget later. First, the default cipher is AEAD (`aes-256-gcm`): nothing authenticated can be released until the tag

is checked at the end, so a mid-stream plaintext read would be a lie. Second, `read()` already means "finalize and reset" for hash / HMAC / sign; a partial encrypt read would look identical to a final one, and the handle would reset under the caller. Block ciphers such as CBC or CTR *could* emit ciphertext at a block boundary, but that would be a second meaning of `read()` next to the AEAD and digest rules, so it is not offered. A second `read()` with nothing written since the last one fails, as when reading a socket with no data. `close()` is idempotent: a second `close()`, or `close()` on a never-written handle, releases the handle and drops its reference to any bound key.

On `we`, writes encrypt to the file and `close()` flushes the AEAD tag. On `re` with `op=decrypt`, `reads()` yields plaintext chunks. On `re` with `op=hash`, the first `read()` returns the digest.

`Attrib()` during the idle window may change `op=`, `alg=`, `cipher=`, `iv=`, and `sig=`. A change while a digest or cipher is in flight fails (1316). Status gets are section 6.4. Handles are not valid `Attrib()` values.

Framing is owned by the runtime: IV prepended, tag appended, both fixed-length per cipher. The same layout is used for `e` / `er` output and for `we` files.

10 10. Error handling

Crypto and TLS connect failures **fail**, they do not `runerr`. This matches the SSL connect path: set `&errortext`, tear down what was allocated, and fail, so callers write `if h := open(...) then ... else ...`. Runtime errors are reserved for type mistakes (a bad mode letter, a non-file where a handle is required).

Error numbers, defined in `src/runtime/data.r`:

Number	Text	Typical cause
1300–1308	SSL / certificate / cipher errors	TLS context and handshake
1311	unknown cryptographic algorithm or cipher	Bad `alg=` or `cipher=`
1312	cryptographic handle lacks required material	Missing key, cert, or wrong role

Number	Text	Typical cause
1313	duplicate cryptographic material role	Two handles both supply the same role
1314	AEAD authentication failed	Tampered or truncated ciphertext
1315	signature verification failed	`op=verify` did not match
1316	invalid cryptographic operation state	Mid-operation `Attrib()`, bad `op=`, empty read
1317	cryptographic material could not be loaded	Unreadable path, parse failure, allocation

11 11. Build integration

Crypto compiles under `HAVE_LIBSSL`, the same guard as TLS. `configure --disable-ssl` turns both off. `--enable-thin` disables SSL as part of a minimal build. Otherwise `CHECK_OPENSSL` probes for the library and headers.

`&features` reports `secure sockets layer encryption`. `tests/crypto/Makefile` skips the suite when that string is absent. TLS echo tests also require `concurrent threads`.

Guards are `#if HAVE_LIBSSL` in `rstructs.h`, `rmacros.h`, `feature.h`, `fsys.r`, `fmisc.r`, `rposix.r`, and `rcrypto.ri`. TLS helpers (`create_ssl_context`, `is_ssl_attr`) and the `CryptoFile` implementation share `src/runtime/rcrypto.ri`.

12 12. Status and remaining work

The language surface described here is implemented: modes `e` / `er` / `eh` / `re` / `we`, TLS handle reuse, DTLS context selection and handshake, `keypass=` / `verifyPeer=` alignment with SSH, and the error numbers in section 10.

Separate configure switch. Crypto cannot be compiled out while leaving TLS sockets on. A `Unicon_Crypto` / `NoCrypto` guard driven by `--disable-crypto` is the obvious next step if a distribution wants encrypted

sockets without the new `open()` modes.

`~` in `key=`. Not expanded, matching SSH. Callers must expand a leading `~/` themselves.

Asymmetric encrypt / decrypt. The `op=` set is hash, HMAC, sign, verify, and symmetric encrypt/decrypt. Public-key encryption is not exposed.

Trust-on-first-use and key generation are out of scope. Certificates and keys are created out of band.

13 Appendix: test coverage

Automated tests live under `tests/crypto/`, one program per concern: `hash`, `hmac`, `sha512`, `sign`, `encrypt`, `iv`, `freshiv`, `tamper`, `filehash`, `filecrypt`, `filekey`, `filekeyattr`, `filetamper`, `opswitch`, `midattrib`, `badalg`, `badkey`, `badrole`, `badverify`, plus TLS / DTLS (`tlsplain`, `tlsverify`, `tlsauth`, `tlsproto`, `tlscipher`, `tlshandle`, `tlspeek`, `dtls`, `dtlsecho`). `tests/crypto/Makefile` skips the suite when OpenSSL is absent, and skips the threaded TLS echo tests when concurrency is absent. Expected output is `tests/crypto/stand/*.std`.

14 References

References

[1] Clinton Jeffery. "How to Write a Unicon Technical Report". doc/utr/utr15.tex. 2013.

[2] The OpenSSL Project. "OpenSSL – Cryptography and SSL/TLS Toolkit". https://www.openssl.org/. 2026.

[3] H. Krawczyk and M. Bellare and R. Canetti. "HMAC: Keyed-Hashing for Message Authentication". RFC 2104. 1997.