

UNDERGRADUATE TECHNICAL PAPER

Extending TLS Cipher Suites through the OpenSSL 3 Provider Interface

Supplying Block Ciphers, Elliptic-Curve Groups and Hash Functions to a TLS Handshake

Bachelor's-degree-level design study
22 September 2026

Reference environment: OpenSSL 3.5.8 on Linux x86_64

Source tags examined: openssl-3.5.5, openssl-3.5.8, OpenSSL_1_1_1g

Companion to *Configuration and Implementation of an OpenSSL 3 Provider*.

Document version 1.0 • English

This is a design and architecture study. Chapter 1.5 and Appendix D state, per claim, which statements were verified against OpenSSL source on the reference machine and which describe the author's system as reported.

Abstract

OpenSSL 3 replaced the ENGINE extension mechanism with providers: loadable modules that supply algorithm implementations through dispatch tables and that are selected at run time by name and property query. The stated motivation for that redesign was cryptographic agility — the ability to introduce an algorithm without modifying the library that uses it. Transport Layer Security is the most demanding test of that claim, because a TLS connection does not consume a single algorithm. It consumes a coordinated set: an authenticated encryption algorithm for record protection, a hash function that drives the key schedule and the transcript, a key-exchange group, and a signature algorithm for authentication.

This paper studies how that coordinated set can be supplied from a provider, and what a system must do to make a new cipher suite usable in an actual TLS 1.3 handshake. It examines the four distinct extension paths that TLS exposes to a provider — algorithm implementation, the TLS-GROUP capability, the TLS-SIGALG capability, and cipher suite registration — and shows that they are not equivalent in mechanism, in configuration, or in the degree to which they are reachable from provider code alone.

The contribution is a design: a provider architecture that supplies a block cipher operating in an AEAD mode, an elliptic-curve group for key agreement, and a hash function for the key schedule, together with the configuration, property-query policy and negotiation behaviour needed to select them during a handshake. The design is specified in enough detail to be implemented — dispatch tables, parameter exchange, context lifecycle, and the ordering constraints the handshake imposes — and is accompanied by a security analysis, a proposed validation strategy, and an interoperability discussion.

The paper deliberately separates three kinds of statement. Descriptions of the OpenSSL provider and TLS machinery are verified against the `openssl-3.5.5` and `openssl-3.5.8` source trees, with file and line references. Design proposals are identified as proposals. Reports of the author's implemented capability are attributed as such. No measurements are reported, because none were taken: Chapter 18 gives a performance evaluation methodology rather than results, and Chapter 19 gives a test plan rather than a test log. This follows the practice of the companion paper, which likewise separated proposed methodology from executed experiment.

Keywords: OpenSSL, provider architecture, TLS 1.3, cipher suite, authenticated encryption, elliptic-curve cryptography, key schedule, cryptographic agility, property query, dispatch table.

Contents

1. Introduction	4
2. Cryptographic Dependencies of a TLS 1.3 Handshake	10
3. The OpenSSL 3 Provider Architecture	16
4. Four Paths from a Provider into a TLS Connection	22
5. The libssl / libcrypto Boundary	28
6. Requirements and Design Goals.....	32
7. Architecture of the Proposed Provider.....	37
8. Design of the Block Cipher (AEAD) Component.....	42
9. Design of the Hash Component.....	46
10. Design of the Elliptic-Curve Group Component	50
11. Design of the Signature Component	54
12. Cipher Suite Registration and Negotiation	57
13. Configuration, Activation and Deployment Mechanics	61
14. Property Queries as Selection Policy.....	65
15. Key Schedule Integration	68
16. Interoperability.....	72
17. Security Analysis	75
18. Performance: A Proposed Evaluation Methodology	79
19. Proposed Validation Strategy	83
20. Discussion, Limitations and Conclusion	87
Appendix A. Provider Skeleton.....	91
Appendix B. Configuration and Operational Reference	97
Appendix C. Parameter Reference.....	101
Appendix D. Status of Claims	104
Appendix E. Glossary.....	108
References	112

1. Introduction

1.1 The problem

A TLS connection is often described as using "a cipher suite", as though the suite were a single choice. It is not. A TLS 1.3 connection simultaneously depends on at least four cryptographic decisions that are negotiated by different mechanisms, carried in different handshake fields, and — the point this paper develops — supplied to OpenSSL through different extension paths. The record layer needs an authenticated encryption algorithm. The key schedule needs a hash function, which also fixes the length of every secret derived during the handshake. The key exchange needs a group. The authentication step needs a signature algorithm matched to the certificate presented.

If an engineer wishes to introduce a new algorithm into this arrangement — a national standard block cipher, a curve mandated by a local regulator, a hash function chosen for migration reasons — the question that matters is not whether the algorithm can be implemented. It is whether the algorithm can be made to *participate*: advertised in the handshake, selected by negotiation, driven through its lifecycle by the TLS state machine, and agreed upon by a peer that was not modified.

OpenSSL 3 offers a mechanism that appears designed for exactly this. The provider interface allows a loadable module to supply algorithm implementations that the library will select at run time by name and by property query, without the application calling the module directly and without OpenSSL being recompiled. The companion study to this paper demonstrated the mechanism for a digest, showing that an algorithm supplied by a third-party module can be fetched and used through the ordinary EVP interface.

Extending that result from a digest to a TLS cipher suite is not a matter of repetition. A digest has one consumer and no negotiation. A cipher suite has a peer.

1.2 Scope

This paper is a design study of the following capability: a provider that supplies a block cipher in an AEAD mode, an elliptic-curve group, and a hash function, such that these algorithms can be used together in a TLS 1.3 connection.

The study covers:

- the cryptographic dependencies of a TLS 1.3 handshake, stated precisely enough to identify every point at which a provider-supplied algorithm is consumed (Chapter 2);
- the provider architecture as it stands in OpenSSL 3.5, including the dispatch table convention, parameter passing, and the library context (Chapter 3);
- the four extension paths by which a provider can influence a TLS connection, and how they differ (Chapter 4);
- the division of responsibility between `libssl` and `libcrypto`, which determines what provider code can and cannot reach (Chapter 5);
- a specified design for each of the three algorithm classes, with dispatch tables, context lifecycle, and parameter contracts (Chapters 7–11);
- cipher suite registration, negotiation and the key schedule (Chapters 12–15);
- interoperability, security analysis, proposed performance methodology, and a proposed validation strategy (Chapters 16–19).

The study does not cover: QUIC, DTLS, TLS 1.2 and earlier, post-quantum key encapsulation as a subject in its own right, hardware security module integration, or certificate path validation. Each is mentioned where it constrains the design, and each is excluded from the detailed treatment.

1.3 Why the provider interface rather than a patch

An engineer who wants a new algorithm in TLS has, in principle, two options: modify OpenSSL, or extend it. The second is preferable for reasons that are practical rather than aesthetic.

A patched OpenSSL must be maintained against a moving upstream. Every security release requires the patch to be rebased, retested and redeployed, and the patched library is no longer the library that distributions ship, package signatures cover, or auditors have reviewed. For a cryptographic library this is a significant operational cost, and it is paid indefinitely.

A provider, by contrast, is a separate shared object with its own lifecycle. It is loaded by configuration, it can be signed and distributed independently, and — critically — the ABI between the core and the provider is a table of function pointers with a negotiated content, not a struct layout. This is the property that makes the FIPS provider's separate validation lifecycle possible, and it is the same property that makes a third-party algorithm module maintainable.

There is a third consideration specific to cryptography. An organisation that must demonstrate which algorithm implementation was used — for compliance, for incident response, or for an audit — benefits from that implementation being a named, versioned, separately hashed artefact rather than a compile-time configuration of a larger library. The provider model makes the answer to "which code computed this ciphertext?" a question with a filename in it.

1.4 Contribution

The contributions of this paper are:

1. **A taxonomy of the extension paths.** Chapter 4 distinguishes four mechanisms that are frequently conflated in practitioner discussion: supplying an implementation for an algorithm TLS already knows, advertising a group through the `TLS-GROUP` capability, advertising a signature algorithm through `TLS-SIGALG`, and registering a cipher

suite. They differ in mechanism and in reach, and the difference determines what a design can achieve without modifying `libssl`.

2. **A specified provider design** for three algorithm classes, at a level of detail sufficient for implementation: dispatch tables, context lifecycle, parameter contracts, and the ordering constraints imposed by the handshake (Chapters 7–11).
3. **An analysis of the negotiation and key-schedule consequences** of introducing a non-standard algorithm, including the transcript-hash coupling that makes the handshake hash a more invasive choice than the record cipher (Chapters 12 and 15).
4. **A security analysis** that treats the provider as part of the trusted computing base and enumerates what that implies (Chapter 17).
5. **A validation strategy** and a performance evaluation methodology, both stated as proposals with their threats to validity (Chapters 18 and 19).

1.5 Status of claims, and how to read this paper

Cryptographic engineering papers fail their readers most often by mixing three kinds of statement without marking the difference: what the system definitely does, what the author intends it to do, and what would be true if the design were implemented. This paper marks them explicitly, and the reader should rely on the marking.

Verified. The statement was checked against OpenSSL source on the reference machine, and a file and line reference is given. The trees examined are `openssl-3.5.8` (build tree present on the reference machine) and the pinned documentation snapshots at `openssl-3.5.5` and `OpenSSL_1_1_1g` stored beside this paper. A reader can re-check any such statement without network access.

Design. The statement specifies part of the proposed system. It has not been built or executed as part of this study. Statements of this kind are the substance of Chapters 7–15 and are written in the specification voice ("the provider exposes...", "the context holds...") because that is how a specification reads, not because the artefact exists.

Reported. The statement describes the capability as reported by the author of the system this paper documents — namely, that cipher suites combining block cipher, elliptic-curve and hash algorithms can be added through the provider interface and used in TLS communication. Such statements are attributed at the point of use. They were not independently reproduced in this study, because no implementation was available to this study; Chapter 19 gives the test plan that would establish them.

Appendix D collects every load-bearing claim in the paper into a single table with its status and, where applicable, its source reference. A reader with limited time who wants to know what this paper actually establishes should read Chapter 4, then Appendix D.

1.6 Reference environment

All verified statements were checked in the following environment:

OpenSSL runtime	OpenSSL 3.5.8 25 Aug 2026 (library: OpenSSL 3.5.8)
OpenSSL headers	3.5.8 (libcrypto, libssl via pkg-config)
Source tree	openssl-3.5.8 (unpacked build tree)
Documentation	openssl-3.5.5 (pinned), OpenSSL_1_1_1g (pinned)
Platform	Linux x86_64, AlmaLinux 9.8
Compiler	gcc (available; no provider was compiled for this study)
Default providers	default (active)

The absence of a compiled artefact is deliberate and is stated here so that it is not mistaken for an omission. This is a design study; Chapter 20 lists what an implementation phase would need to produce for the design to be considered validated.

1.7 Structure of the paper

Chapters 2–5 establish the ground: what TLS needs, what a provider is, how the two can meet, and where the boundary between the two OpenSSL libraries falls. Chapter 6 states requirements. Chapters 7–15 are the design proper, moving from the provider's overall shape through each algorithm class to negotiation and the key schedule.

Chapters 16–19 evaluate the design against interoperability, security, performance and testability. Chapter 20 concludes and states what remains.

Readers already familiar with the provider architecture may begin at Chapter 4. Readers interested only in what is provably true of stock OpenSSL should read Chapters 4 and 5 and Appendix D.

2. Cryptographic Dependencies of a TLS 1.3 Handshake

This chapter establishes what a TLS 1.3 connection actually requires of a cryptographic library. The purpose is not to restate the protocol, which is specified in RFC 8446, but to identify every point at which an algorithm implementation is consumed, because each such point is a place where a provider-supplied algorithm must be able to appear.

2.1 Four independent decisions

A TLS 1.3 connection is parameterised by four cryptographic choices. They are negotiated separately, they appear in different handshake messages, and — this is the observation the rest of the paper depends on — they are supplied to OpenSSL through different mechanisms.

Decision	Negotiated by	Fixes
AEAD algorithm	Cipher suite	Record protection; key and IV lengths
Hash function	Cipher suite	Transcript hash, HKDF, all secret lengths
Key exchange group	supported_groups extension and key_share	The shared secret
Signature algorithm	signature_algorithms extension	Authentication of the handshake

A crucial structural fact follows from the first two rows: in TLS 1.3 the cipher suite identifier binds the AEAD algorithm *and* the hash together. `TLS_AES_128_GCM_SHA256` is one code point naming two algorithms. This is a deliberate simplification relative to TLS 1.2, where the suite additionally encoded the key exchange and authentication methods; in 1.3 those moved into extensions, which is precisely why groups and signature algorithms are independently negotiable and cipher suites are not.

Verified. The coupling is visible in the OpenSSL cipher table. Each entry in `tls13_ciphers[]` carries both the AEAD identifier and a handshake MAC selector; the first entry pairs `SSL_AES128GCM` with `SSL_HANDSHAKE_MAC_SHA256` (`ssl/s3_lib.c:39–56`, tree `openssl-3.5.8`).

2.2 The handshake, as a sequence of algorithm uses

Stated as a list of demands on the library rather than as a message flow, a TLS 1.3 handshake proceeds roughly as follows. The client offers cipher suites, supported groups and signature algorithms, and speculatively generates a key share for one or more groups. Generating that share is the first algorithm use: a key pair in the offered group must exist before the first flight is sent.

The server selects a suite, a group and — if it must authenticate — a signature algorithm. It generates its own key share, completes the key agreement, and from that point the key schedule begins. Every secret in TLS 1.3 is derived by HKDF using the hash bound to the selected suite, and the transcript hash over all handshake messages so far participates in those derivations. The server signs the transcript with its long-term key, and both sides compute Finished values using a MAC keyed from the schedule.

The ordering matters for a provider design in a way that is easy to miss:

1. **The group is used before the suite is known.** The client generates a key share when it constructs its first flight, before the server has chosen anything. A provider-supplied group must therefore be available at the point the client is composing `ClientHello`, and must be advertised in `supported_groups` for the server to be able to choose it.
2. **The hash is used from the moment the suite is selected, and never changes.** The transcript hash covers messages that were exchanged *before* the suite was chosen; implementations buffer the early transcript and hash it once the algorithm is known. A provider-supplied hash is therefore consumed retrospectively over data already sent.

3. **The AEAD is used last of the three**, once handshake traffic keys exist, and then continuously for the life of the connection.

The consequence for this paper is that the three algorithm classes in its title are not symmetric. The group must be advertised; the hash must be selectable at the moment a suite is chosen and then drives everything derived; the AEAD is comparatively self-contained.

2.3 The key schedule and why the hash is special

TLS 1.3 derives all keying material through HKDF, structured as a schedule of `Extract` and `Expand` steps. The hash function selected by the cipher suite determines the HMAC used inside HKDF and therefore the length of every secret in the schedule: a suite naming SHA-256 produces 32-byte secrets throughout, one naming SHA-384 produces 48-byte secrets.

This has a design consequence that is more restrictive than it first appears. Introducing a new hash function into TLS is not merely adding an implementation; it changes the width of the key schedule. Every buffer, every derived secret, every `Finished` value and every exporter output takes its length from that hash. An implementation that assumed 32 or 48 bytes — in the TLS stack, in an application's exporter usage, or in a session-resumption cache — is affected by the change in a way it is not affected by a change of record cipher.

A new AEAD, by contrast, is comparatively contained. It has a key length, an IV length and a tag length; provided the record layer can learn those three values, the rest of the protocol is indifferent to which algorithm produced the ciphertext.

Design implication. A provider that supplies a hash for use as a TLS handshake hash must expose its output length through the ordinary parameter mechanism, and the consuming stack must honour it rather than assuming a constant. Chapter 15 develops this; it is the single most invasive aspect of the design.

2.4 What the record layer needs from an AEAD

TLS 1.3 record protection uses an AEAD interface with a per-record nonce constructed from a static IV and a sequence number. The record layer requires the following from whatever algorithm it is handed:

Key length

How many bytes of key material the schedule must produce.

IV length

The width of the static IV, into which the sequence number is folded.

Tag length

The expansion between plaintext and ciphertext, needed for record sizing and for the maximum-fragment computation.

AAD handling

The record header is supplied as associated data.

Deterministic nonce construction

TLS constructs the nonce itself; the algorithm must accept an externally supplied nonce rather than generating one.

The last point excludes some AEAD constructions from direct use. An algorithm that insists on generating its own nonce, or that uses a nonce width incompatible with the 64-bit sequence number folded into the static IV, cannot be dropped into the TLS 1.3 record layer without adaptation. This is a constraint on algorithm choice, not on the provider mechanism, but it must be checked before a design commits to a particular cipher.

2.5 What key exchange needs from a group

The `key_share` extension carries an opaque octet string whose interpretation is defined per group. For an elliptic-curve group this is a point encoding; for a finite-field group, an integer. The TLS stack requires of a group:

- a code point, so it can be named in `supported_groups`;
- a wire encoding for a public share, with a known or derivable length;

- key generation, producing a private value and its public share;
- key derivation, combining a local private value with a peer share to produce a shared secret of known length;
- validation of a received peer share, since a malformed or malicious share must be rejected rather than processed.

The last item deserves emphasis because it is where key-exchange implementations have historically failed. A group implementation that does not validate peer input — that accepts a point not on the curve, or a small-order point — introduces a vulnerability that no amount of correctness in the rest of the stack will compensate for. Chapter 17 returns to this.

2.6 What authentication needs from a signature algorithm

The server signs a defined context string concatenated with the transcript hash, using an algorithm named by a code point in `signature_algorithms`. The requirements are a code point, an association with a key type and OID so that certificates can be matched to it, a signing operation, a verification operation, and a statement of the security level so that policy can rank it.

Verified. These are exactly the fields OpenSSL's capability mechanism carries. The `TLS_SIGALG_ENTRY` macro builds an `OSSL_PARAM` array containing the IANA name, the algorithm name, the OID, the code point and the security bits (`providers/common/capabilities.c:292–302`).

2.7 Summary of the demand surface

Collecting the above, a provider that wishes to participate fully in a TLS 1.3 connection must be able to present, through whatever mechanism OpenSSL offers:

Class	Must expose	Consumed at
AEAD	Key, IV and tag lengths; external nonce; AAD	After handshake keys exist
Hash	Output length; streaming update; duplication of state	From suite selection onward, retrospectively over the transcript
Group	Code point; share encoding; keygen; derive; peer-share validation	Before the first flight is sent
Signature	Code point; OID; key type; sign; verify; security bits	Server Certificate Verify

The next chapter describes the mechanism through which a provider exposes anything at all; Chapter 4 then maps this demand surface onto it.

3. The OpenSSL 3 Provider Architecture

This chapter describes the mechanism by which a loadable module supplies algorithms to OpenSSL 3. It is deliberately concrete: the design chapters later in the paper specify dispatch tables and parameter arrays, and those specifications are only meaningful against an accurate account of the convention they follow.

3.1 What a provider is

The OpenSSL glossary defines a provider as "a component that groups together algorithm implementations", which may come from OpenSSL itself or from third parties. Operationally, a provider is a shared object exporting a single symbol, `SSL_provider_init`, which the core calls at load time. Everything else the provider offers is reached through function pointers returned from that call.

Three ideas do the work in this architecture: the dispatch table, the parameter array, and the property query. Each replaces something that in the ENGINE era was a struct field or a global.

3.2 Dispatch tables

A dispatch table is an array of `SSL_DISPATCH` entries, each pairing a numeric function identifier with a function pointer:

```
typedef struct ssl_dispatch_st {
    int function_id;
    void (*function)(void);
} SSL_DISPATCH;
```

The array is terminated by a zero entry. The core walks it, recognises the identifiers it knows, and ignores the rest. This single convention supplies the architecture's two most important properties.

First, **version independence**. A provider compiled against an older set of identifiers is still usable: the core finds the entries it recognises and treats the absent ones as unimplemented. A provider compiled against a newer set does not break an older core, which simply ignores identifiers it does not know. Contrast the ENGINE mechanism, where the interface was the layout of `RSA_METHOD` and similar structures, so adding a field was a compatibility event affecting every engine.

Second, **partial implementation is legitimate**. A provider need not implement an entire operation. It supplies what it supplies; the core's fetching machinery decides whether what is offered is sufficient for the request at hand. This is what allows a provider to offer a single algorithm without implementing anything else.

The initialisation function receives the core's own dispatch table and returns the provider's:

```
int OSSL_provider_init(const OSSL_CORE_HANDLE *handle,
                      const OSSL_DISPATCH *in,
                      const OSSL_DISPATCH **out,
                      void **provctx);
```

The `in` table is how the provider reaches back into the core — for memory allocation, for error reporting, for reading configuration parameters. The `out` table is how the core reaches the provider. The symmetry is the point: neither side links against the other's symbols.

3.3 The provider's own dispatch table

A minimal provider returns a table containing a teardown function, a query function, and parameter accessors. The query function is the entry point that matters for algorithm supply:

```
static const OSSL_ALGORITHM *query(void *provctx, int operation_id,
                                   int *no_cache);
```

The core calls it with an operation identifier — digest, cipher, key exchange, signature, key management, and so on — and the provider returns an array of `OSSL_ALGORITHM` entries for that operation, or `NULL` if it offers none:

```
typedef struct ossl_algorithm_st {
    const char *algorithm_names;      /* colon-separated list */
    const char *property_definition; /* e.g. "provider=edu" */
    const OSSL_DISPATCH *implementation;
    const char *algorithm_description;
} OSSL_ALGORITHM;
```

Two details of this structure shape the design chapters. The names field is a *colon-separated list*, so one implementation may answer to several names — which is how an algorithm can be reachable by both a formal name and a common alias. The property definition is a string of key/value pairs, and it is the sole input the provider contributes to the selection policy described in §3.5.

3.4 Parameter arrays

Data crossing the core/provider boundary travels as `OSSL_PARAM` arrays rather than as struct fields:

```
typedef struct ossl_param_st {
    const char *key;
    unsigned int data_type;
    void *data;
    size_t data_size;
    size_t return_size;
} OSSL_PARAM;
```

An algorithm advertises which parameters it will answer through a "gettable" function, and the caller supplies an array to be filled. The mechanism is verbose compared with reading a struct member, and this verbosity is the price of the version independence described above: a parameter the receiver does not recognise is skipped rather than misinterpreted.

For this paper the mechanism is central, because it is how the demand surface of Chapter 2 is satisfied. The record layer learns an AEAD's key, IV and tag lengths by requesting parameters; the key schedule learns a hash's output length the same way. A provider that fails to answer a parameter the consumer requires does not merely lose a feature — the algorithm becomes unusable in that role.

3.5 Fetching and property queries

In OpenSSL 3 an algorithm is obtained by *fetching*: naming it and, optionally, supplying a property query string that constrains which implementation is acceptable.

```
EVP_MD *md = EVP_MD_fetch(libctx, "SHA2-256", "provider=default");  
EVP_CIPHER *c = EVP_CIPHER_fetch(libctx, "AES-256-GCM", NULL);
```

A property query is a sequence of key/value assertions. Properties may be *required* or merely *preferred*, and a query may specify that a property must be absent. The resulting selection is a filter over every algorithm advertised by every activated provider in the given library context.

Fetches objects are reference-counted and owned by the caller, which must free them. This is a genuine change from the 1.1.1 idiom, where `EVP_sha256()` returned a pointer to a static structure that the caller did not own.

Fetching also has a cost, and the OpenSSL project states it plainly: retrieving an algorithm from a provider "involves searching for an algorithm by name", which "is much slower than directly accessing a method table", with the recommendation to prefetch algorithms used many times. Chapter 18 treats the implications for a TLS workload, where a handshake fetches several algorithms and a bulk transfer uses one of them continuously.

3.6 Library contexts

An `SSL_LIB_CTX` is a scope within which providers are loaded and configuration applies. Code that passes `NULL` uses the default context. Two library contexts in one process may have entirely different providers activated and therefore resolve the same algorithm name to different implementations.

For a TLS design this is both a tool and a hazard. It is a tool because an application can confine an experimental provider to a context used by one connection class, leaving the rest of the process on stock implementations. It is a hazard because an `SSL_CTX` is associated with a library context at creation, and an algorithm that is available in the application's default context but not in the one the `SSL_CTX` uses will fail to resolve at a point in the handshake that produces an unhelpful error. Chapter 13 treats configuration and activation with this in mind.

3.7 Activation and configuration

Providers reach a process by one of two routes. The application may load one programmatically with `SSL_PROVIDER_load()`, naming it and the context. Alternatively the configuration file may activate it, which requires no application change at all:

```
openssl_conf = openssl_init

[openssl_init]
providers = provider_sect

[provider_sect]
default = default_sect
tlsext = tlsext_sect

[default_sect]
activate = 1

[tlsext_sect]
module = /usr/local/lib/openssl-modules/tlsext.so
activate = 1
```

The configuration route is what makes a provider deployable to an application that cannot be modified — the ordinary case for introducing an algorithm into an existing TLS service. It is also the route that most

needs care, because activating a provider changes algorithm resolution for everything in that context, including code paths the deployer was not thinking about.

A trap worth stating early. Activating a non-default provider does not by itself deactivate the default provider, but a configuration that lists only the new provider *does*. A configuration file that activates a custom provider and omits default will leave the process without the algorithms everything else expects, and the resulting failures appear far from the configuration that caused them.

3.8 What the architecture does not provide

Three limits are worth stating now, because the design chapters must work within them.

A provider supplies *implementations of algorithms*. It does not supply protocol behaviour, message formats or state machines. Nothing in the provider interface allows a module to change how a handshake is sequenced.

A provider is selected by name and property, so a consumer must already be asking for something the provider can answer to. An algorithm nobody fetches is never used, however correctly it is implemented.

Finally, the provider boundary is a trust boundary in one direction only. The core calls into provider code with the process's full privileges; a provider is not sandboxed. Chapter 17 develops the consequences.

4. Four Paths from a Provider into a TLS Connection

Practitioner discussion of "adding an algorithm to TLS through a provider" tends to treat the task as one thing. It is four things, with different mechanisms and different degrees of independence from the TLS library itself. This chapter separates them. It is the analytical core of the paper, and the design chapters are organised around the distinctions drawn here.

4.1 The four paths

#	Path	Mechanism	What it adds
1	Algorithm implementation	OSSL_ALGORITHM returned from the provider's query function	A different implementation of something TLS already names
2	Group advertisement	TLS-GROUP capability	A new key-exchange group, including a new code point
3	Signature advertisement	TLS-SIGALG capability	A new signature algorithm, including a new code point
4	Cipher suite registration	The suite table consulted by libssl	A new suite pairing an AEAD with a handshake hash

Paths 2 and 3 are the interesting ones architecturally, because they show that OpenSSL's designers did build a route by which a provider introduces something genuinely new to the protocol — not merely a faster version of an existing algorithm, but a new code point that appears on the wire. Path 1 is the most common and the least ambitious. Path 4 is the subject of this paper's title and is treated in Chapter 12.

4.2 Path 1: supplying an implementation

The simplest path is to implement an algorithm TLS already knows, under the name TLS already uses, and let property-based selection prefer it. A provider offering AES-256-GCM competes with the default provider's implementation; a property query, or the ordering of activated providers, determines which is chosen.

This path adds nothing to the protocol. No new code point appears on the wire, no peer needs to change, and interoperability is unaffected — the connection is an ordinary `TLS_AES_256_GCM_SHA384` connection that happens to be computed by different code. That is precisely why it is useful: it is the path for hardware offload, for a formally verified implementation, or for a validated module. It is also the path with the sharpest failure mode, since a subtly wrong implementation of a standard algorithm produces connections that fail in ways the protocol has no vocabulary to describe.

The companion paper's `edu` provider is an instance of this path applied to a digest: it supplies `EDU-SHA256`, fetched by name, delegating to the default provider in a separate library context.

4.3 Paths 2 and 3: the capability mechanism

OpenSSL exposes a general mechanism by which a provider declares support for something the protocol layer will consume, called a *capability*. The core asks a provider for a named capability, and the provider answers by invoking a callback once per item it wishes to advertise, passing an `SSL_PARAM` array that describes it.

Verified. Two capabilities relevant to TLS are recognised. The built-in providers dispatch on the capability name and answer `TLS-GROUP` and `TLS-SIGALG`, returning 0 — "we don't support this capability" — for anything else (providers/common/capabilities.c:335–347, tree openssl-3.5.8).

4.3.1 What a group advertisement contains

A group is advertised as a parameter array carrying its external (IANA) name, an internal algorithm name, the algorithm that implements it, a numeric code point, and security metadata. The built-in list is constructed by a macro:

```
#define TLS_GROUP_ENTRY(tlsname, realname, algorithm, idx)      \
{                                                                \
    OSSL_PARAM_utf8_string(OSSL_CAPABILITY_TLS_GROUP_NAME,    \
        tlsname, sizeof(tlsname)),                             \
    OSSL_PARAM_utf8_string(OSSL_CAPABILITY_TLS_GROUP_NAME_INTERNAL, \
        realname, sizeof(realname)),                           \
    OSSL_PARAM_utf8_string(OSSL_CAPABILITY_TLS_GROUP_ALG,      \
        algorithm, sizeof(algorithm)),                         \
    OSSL_PARAM_uint(OSSL_CAPABILITY_TLS_GROUP_ID,              \
        (unsigned int *)&group_list[idx].group_id),           \
    ...                                                         \
}
```

and used to build the list of supported groups:

```
TLS_GROUP_ENTRY("x25519", "X25519", "X25519", 28),
TLS_GROUP_ENTRY("x448", "X448", "X448", 29),
TLS_GROUP_ENTRY("secp256r1", "prime256v1", "EC", 22),
```

Verified. providers/common/capabilities.c:96–108 for the macro and :161–164 for these entries, tree openssl-3.5.8.

Three observations follow. The *code point is supplied by the provider*, as a parameter, which means a group that OpenSSL's own code does not know can be advertised and will appear in `supported_groups`. The *internal name maps the wire name to an algorithm the provider implements*, so the group name and the key-management algorithm need not coincide. And the `ec` entry shows that many named curves may share one implementing algorithm, distinguished by a curve parameter — the arrangement an elliptic-curve design in this paper will follow.

4.3.2 What a signature advertisement contains

Signature algorithms are advertised analogously, with the IANA name, the algorithm name, an OID, a code point and a security-bits value:

```

#define TLS_SIGALG_ENTRY(tlsname, algorithm, oid, idx)      \
{                                                           \
    OSSL_PARAM_utf8_string(OSSL_CAPABILITY_TLS_SIGALG_IANA_NAME, \
        tlsname, sizeof(tlsname)),                          \
    OSSL_PARAM_utf8_string(OSSL_CAPABILITY_TLS_SIGALG_NAME,    \
        algorithm, sizeof(algorithm)),                      \
    OSSL_PARAM_utf8_string(OSSL_CAPABILITY_TLS_SIGALG_OID,     \
        oid, sizeof(oid)),                                  \
    OSSL_PARAM_uint(OSSL_CAPABILITY_TLS_SIGALG_CODE_POINT,     \
        (unsigned int *)&sigalg_constants_list[idx].code_point), \
    OSSL_PARAM_uint(OSSL_CAPABILITY_TLS_SIGALG_SECURITY_BITS,  \
        ...),                                               \
}

```

The most instructive fact about this path is what OpenSSL itself uses it for:

```

TLS_SIGALG_ENTRY("mldsa44", "ML-DSA-44", "2.16.840.1.101.3.4.3.17", 0),
TLS_SIGALG_ENTRY("mldsa65", "ML-DSA-65", "2.16.840.1.101.3.4.3.18", 1),
TLS_SIGALG_ENTRY("mldsa87", "ML-DSA-87", "2.16.840.1.101.3.4.3.19", 2),

```

Verified. providers/common/capabilities.c:292–302 for the macro and :316–318 for the ML-DSA entries, tree openssl-3.5.8.

Post-quantum signature algorithms — algorithms that did not exist when the TLS code was written — reach the handshake through the same capability mechanism available to a third-party provider. This is the strongest available evidence that paths 2 and 3 are genuine extension points rather than internal conveniences: OpenSSL uses them to introduce new cryptography to TLS, and the interface is not privileged.

4.4 Path 4: cipher suites

Cipher suites differ from groups and signature algorithms in how `libssl` obtains the list of what exists. Rather than querying providers for a capability, the TLS implementation consults a table of suite descriptions, each binding a code point to an AEAD algorithm, a handshake MAC, protocol version bounds and strength metadata.

Verified. In the reference tree the TLS 1.3 suites are held in a table declared static `SSL_CIPHER tls13_ciphers[]` whose length is taken with `OSSL_NELEM` at compile time (`ssl/s3_lib.c:26` and `:39`, tree `openssl-3.5.8`). Each entry names an AEAD and a handshake MAC, as shown in §2.1. There is no `TLS-CIPHERSUITE` capability in `ossl_prov_get_capabilities()`, which answers only `TLS-GROUP` and `TLS-SIGALG`.

The design consequence is a real asymmetry, and it should be stated without softening: the three paths above place the provider in control of what is advertised, whereas the suite list is owned by the TLS library. A system that introduces a new suite must therefore arrange for that suite to be present in the list `libssl` consults, in addition to providing the algorithms behind it.

Reported. The system documented by this paper is reported by its author to add cipher suites combining block cipher, elliptic-curve and hash algorithms through the provider interface, and to use them in TLS communication. This study had no access to that implementation and did not reproduce the result. Chapter 12 specifies the registration and negotiation behaviour such a capability entails; Chapter 19 gives the test plan that would demonstrate it; Appendix D records the claim and its status.

4.5 Comparing the paths

Property	Path 1 Implementation	Path 2 Group	Path 3 Sigalg	Path 4 Suite
New code point on the wire	No	Yes	Yes	Yes
Peer must also support it	No	Yes	Yes	Yes
Advertised by provider	n/a	Yes	Yes	See §4.4
Reachable without touching <code>libssl</code>	Yes	Yes	Yes	See §4.4

Property	Path 1 Implementation	Path 2 Group	Path 3 Sigalg	Path 4 Suite
Affects key schedule width	No	No	No	Yes, if the hash changes
Failure is visible as	Wrong results	No shared group	Handshake failure	No shared suite

4.6 Which path a requirement should take

The taxonomy yields practical guidance. If the requirement is a validated or accelerated implementation of a standard algorithm, path 1 is correct and the others are unnecessary complexity. If the requirement is a new curve or a new signature scheme — the common case for national standards and for post-quantum migration — paths 2 and 3 are the intended mechanism, and they do not require a modified TLS library. If the requirement is genuinely a new *suite*, pairing an AEAD with a handshake hash under a new code point, the work includes whatever is needed to make that suite visible to the negotiation logic, and that is the subject of Chapter 12.

A requirement is frequently stated as the fourth kind when it is really the first or second. An organisation that must use a particular national block cipher may find that its actual obligation is satisfiable by path 1 if a standard suite naming an equivalent-strength AEAD is acceptable, or that its curve requirement is a path-2 problem. Establishing which path a requirement truly needs is, in the experience this paper reflects, the single most valuable step in the design.

5. The libssl / libcrypto Boundary

OpenSSL is two libraries. `libcrypto` implements cryptography and hosts the provider machinery; `libssl` implements TLS and consumes `libcrypto`. Where a given responsibility falls determines whether provider code can reach it, and much confusion about what providers can do dissolves once the division is stated.

5.1 The division

Concern	Library	Provider-reachable
Algorithm implementations	<code>libcrypto</code>	Directly — this is what providers are
Algorithm selection by name and property	<code>libcrypto</code>	Directly
Capability advertisement (groups, sigalgs)	Provider, consumed by <code>libssl</code>	Directly
Cipher suite list	<code>libssl</code>	See §4.4
Handshake state machine	<code>libssl</code>	No
Record layer framing	<code>libssl</code>	No
Key schedule sequencing	<code>libssl</code>	No — but it consumes provider algorithms at every step
Extension encoding	<code>libssl</code>	No

The pattern is that `libssl` owns *protocol* and `libcrypto` owns *computation*. A provider changes what is computed, never how the protocol is sequenced. This is a sound separation and not an oversight: a module that could alter handshake sequencing would be able to weaken the protocol invisibly, and the security analysis in Chapter 17 would be considerably harder to write.

5.2 How libssl consumes an algorithm

When `libssl` needs an algorithm it performs an ordinary fetch against the library context associated with the `SSL_CTX`. The name it fetches is determined by the negotiated parameters — the suite entry names an AEAD and a handshake MAC, the group entry names a key-management algorithm — and the property query is whatever the context's configuration established.

This has an important corollary for deployment. Because the fetch is ordinary, a provider-supplied implementation is selected by the ordinary rules: activation, property query, and provider ordering. There is no TLS-specific registration for path 1. An operator who has activated a provider that offers AES-256-GCM with a matching property has already changed which code protects their records, whether or not they intended to.

Deployment hazard. The invisibility cuts both ways. Because path-1 substitution requires no protocol change and no application change, there is no handshake artefact recording that it happened. Chapter 16 recommends that a deployment which substitutes implementations record the fact out of band, since the connection itself will not show it.

5.3 Where the key schedule meets the provider

The TLS 1.3 key schedule is sequenced by `libssl` but computed by `libcrypto`. Each `Extract` and `Expand` step is an HKDF operation keyed by the suite's hash. Consequently a provider that supplies the hash is invoked many times per handshake, in a pattern fixed by the protocol, on inputs whose lengths derive from the hash's own output size.

Two properties of the hash implementation therefore matter more than they would in a general-purpose setting. It must report its output length accurately through the parameter mechanism, because the schedule sizes its buffers from that value. And it must support duplication of a partially updated context, because the transcript hash is used at several points in the handshake without being finalised — the running transcript is duplicated, the copy finalised for one derivation, and the

original continued. A hash implementation lacking a working duplication operation will fail in the handshake even though it computes correct digests in isolation.

5.4 Where the record layer meets the provider

Record protection fetches the AEAD once per traffic-key epoch and uses it for every record until keys are updated. The provider's cipher implementation is therefore on the hot path of all bulk data transfer, and its per-operation overhead — not its per-fetch overhead — dominates throughput. Chapter 18 separates these two costs, which are frequently conflated when provider performance is discussed.

The record layer also requires that the cipher accept an externally supplied IV and AAD, as described in §2.4. In provider terms this means the cipher must implement the parameter-setting entry points for these values and must not attempt to manage nonces itself.

5.5 Consequences for the design

The boundary yields four rules that the design chapters observe:

1. **Everything the protocol must know is a parameter.** Lengths, identifiers and capabilities cross the boundary as `SSL_PARAM` entries; nothing crosses as a struct field. A design that omits a parameter creates a failure at the consuming site, not at the provider.
2. **Nothing the provider does can change protocol sequencing.** An algorithm that requires a different message flow cannot be accommodated by a provider at all, regardless of how it is implemented.
3. **Selection happens in a context.** The library context the `SSL_CTX` uses is the one whose activated providers matter; the application's default context is irrelevant if it is not the same one.
4. **Advertisement and implementation are separate obligations.** Supplying a group implementation without advertising the capability yields an algorithm that works when fetched directly and never

appears in a handshake — a failure mode that looks like a negotiation bug and is really a missing capability entry.

5.6 A note on what "without modifying OpenSSL" means

The phrase is used loosely in practitioner writing, and precision helps. Three distinct situations are commonly described the same way:

No modification at all

A provider is built separately and activated by configuration. The OpenSSL binaries are those the distribution shipped. Paths 1-3 achieve this.

No modification to the application

The application is unchanged, but the OpenSSL installation has been configured — possibly substantially — to load and prefer the new module. This is the usual deployment situation and is a weaker claim than the first.

No modification to the protocol

The wire format is unchanged and peers are unaffected. True of path 1; false of paths 2, 3 and 4, each of which introduces a code point that a peer must recognise.

A claim to have added an algorithm "without modifying OpenSSL" should be read against these three meanings, and this paper states which it intends wherever the question arises.

6. Requirements and Design Goals

This chapter states what the designed system must do, in terms precise enough that Chapters 7–15 can be checked against them and Chapter 19 can propose tests for them. Functional requirements are labelled **F**, non-functional **N**, and constraints **C**.

6.1 Functional requirements

ID	Requirement	Rationale
F1	The provider shall supply a block cipher in an AEAD mode suitable for TLS 1.3 record protection.	§2.4
F2	The provider shall supply a hash function usable as a TLS handshake hash, including streaming update and context duplication.	§5.3
F3	The provider shall supply an elliptic-curve group for key agreement, including generation, derivation and peer-share validation.	§2.5
F4	The provider shall advertise its group through the TLS-GROUP capability with a code point, wire name and security metadata.	§4.3.1
F5	The algorithms shall be selectable by property query, so that a deployment can require them explicitly.	§3.5
F6	The provider shall be activatable by configuration file, without modifying the application.	§3.7
F7	A cipher suite combining F1 and F2 shall be usable in a TLS 1.3 handshake.	Chapter 12
F8	Each algorithm shall report its parameters — key, IV, tag and digest lengths — through the parameter mechanism.	§5.5 rule 1
F9	The provider shall coexist with the default provider in the same library context without displacing it.	§3.7

ID	Requirement	Rationale
F10	Failure to satisfy a required property shall produce a clean fetch failure, not a silent fallback.	§6.4

6.2 Non-functional requirements

ID	Requirement
N1	Algorithm objects shall be prefetched and reused for the lifetime of a connection rather than fetched per record.
N2	No operation shall branch on secret data in a way that is observable through timing, to the extent the implementation language permits.
N3	The provider shall be a single shared object with no dependencies beyond libcrypto.
N4	All context state shall be freed on teardown, including after an error path.
N5	The provider shall be buildable and testable without modifying the OpenSSL installation.
N6	Diagnostic failure paths shall report through the core's error mechanism rather than writing to standard error.

6.3 Constraints

ID	Constraint	Source
C1	The provider cannot alter handshake sequencing or message encoding.	§5.1
C2	The AEAD must accept an externally constructed nonce.	§2.4
C3	A change of handshake hash changes the width of the entire key schedule.	§2.3
C4	Any new code point must be recognised by the peer; unilateral introduction cannot produce interoperability.	§4.5
C5	The provider executes with the full privileges of the process.	§3.8

ID	Constraint	Source
C6	Algorithms must be advertised in the same library context the SSL_CTX uses.	§5.5 rule 3

6.4 The fail-closed principle

Requirement F10 deserves separate treatment because it is the requirement most often violated by systems of this kind, and the violation is not visible in testing that only checks the happy path.

A deployment that introduces an algorithm for a compliance reason needs the connection to *fail* when the algorithm is unavailable. The alternative — falling back to a standard algorithm and completing the handshake — produces a working connection that does not satisfy the requirement the deployment exists to satisfy, and produces it silently.

OpenSSL's property mechanism supports this directly: a required property that no implementation satisfies causes the fetch to fail, and the fetch failure propagates. The design must therefore be careful never to specify its properties as merely preferred where the deployment intends them as mandatory. Chapter 14 develops the distinction, and Chapter 19 proposes a negative test for it — a test that deliberately deactivates the provider and asserts that the handshake fails rather than succeeding by another route.

Design principle. A check that has never been observed to fail has not been tested. Every requirement in §6.1 that can be violated should have a test that violates it and confirms the system notices, not merely a test that satisfies it and confirms the system works. This principle governs the test plan in Chapter 19.

6.5 Explicit non-goals

The following are outside the design, and stating them prevents the evaluation chapters from being read as having neglected them:

- **A new cryptographic primitive.** The design integrates algorithms; it does not propose new ones. Any primitive used should be one with published analysis.
- **FIPS validation.** The provider is not a validated module and its algorithms are not approved. Chapter 17 discusses what validation would additionally require.
- **Protocol changes.** Excluded by C1.
- **TLS 1.2 support.** The suite structure differs and the key schedule is different; supporting both would double the design without illuminating it.
- **Performance competitiveness.** The design aims to be correct and measurable, not fast. Chapter 18 proposes how speed would be assessed if it mattered.

6.6 Traceability

Each requirement is addressed by an identified chapter, and Appendix D carries the matrix forward to the claim table. A requirement with no corresponding design section is a gap, and a design section addressing no requirement is scope creep; the matrix is maintained to make both visible.

Requirement	Addressed in
F1, C2	Chapter 8
F2, C3	Chapters 9 and 15
F3, F4	Chapters 10 and 11
F5, F10	Chapter 14
F6, C6	Chapter 13

Requirement	Addressed in
F7, C4	Chapters 12 and 16
F8, F9	Chapter 7
N1	Chapter 18
N2, C5	Chapter 17
N3-N6	Chapter 7

7. Architecture of the Proposed Provider

This chapter specifies the overall shape of the provider: its module structure, its initialisation, the context objects it maintains, and the conventions the per-algorithm chapters that follow will assume. Requirements F8, F9 and N3–N6 are addressed here.

7.1 Module structure

The provider is a single shared object, conventionally installed beside the other modules:

```
/usr/local/lib/openssl-modules/tlsex.so
```

Internally it is organised by operation, with one translation unit per algorithm class and a small core:

tlsex/		
provider.c	OSSL_provider_init, query dispatch, provider params	
ctx.c	provider context, configuration, delegate handles	
cipher_aead.c	block cipher in AEAD mode	(Chapter 8)
digest.c	hash function	(Chapter 9)
keymgmt_ec.c	key management for the EC group	(Chapter 10)
keyexch_ec.c	key agreement	(Chapter 10)
signature.c	signature operations	(Chapter 11)
capabilities.c	TLS-GROUP and TLS-SIGALG answers	(Chapters 10, 11)
params.c	shared OSSL_PARAM helpers	

The separation matters for more than tidiness. Because the core's query function dispatches on operation identifier, a provider that offers several operations is effectively several independent implementations sharing one initialisation. Keeping them in separate units keeps the dispatch tables next to the functions they name and makes it possible to build a reduced variant — digest only, say — by omitting a unit and its table entry.

7.2 Initialisation

The initialisation function performs four tasks in order: it records the core handle, it walks the incoming dispatch table to find the core functions it needs, it allocates and populates the provider context, and it returns its own dispatch table.

```
int OSSL_provider_init(const OSSL_CORE_HANDLE *handle,
                      const OSSL_DISPATCH *in,
                      const OSSL_DISPATCH **out,
                      void **provctx)
{
    PROV_CTX *ctx;

    if ((ctx = prov_ctx_new(handle, in)) == NULL)
        return 0;
    *provctx = ctx;
    *out = provider_dispatch;
    return 1;
}
```

Two conventions are worth stating because they are easy to get wrong and hard to debug. The incoming table must be *walked*, not indexed: the core is not obliged to supply entries in any particular order, and a provider that assumes an order will read the wrong pointer on a different OpenSSL version. And the provider must tolerate the absence of optional core functions, using them only if present.

Design. The provider context holds the core handle, the resolved core function pointers, a private `OSSL_LIB_CTX` used for any delegated operation (§7.4), and configuration read from the provider's section of the configuration file. It is created once per activation and destroyed by the teardown entry point.

7.3 Query dispatch

The query function answers one array per operation identifier:

```
static const OSSL_ALGORITHM *query(void *provctx, int operation_id,
                                   int *no_cache)
{
    *no_cache = 0;
    switch (operation_id) {
```

```

    case OSSL_OP_DIGEST:      return tlsext_digests;
    case OSSL_OP_CIPHER:      return tlsext_ciphers;
    case OSSL_OP_KEYMGMT:     return tlsext_keymgmt;
    case OSSL_OP_KEYEXCH:     return tlsext_keyexch;
    case OSSL_OP_SIGNATURE:   return tlsext_signatures;
    }
    return NULL;
}

```

Setting `no_cache` to zero permits the core to cache the returned array, which is correct for static tables and is the normal case. A provider whose algorithm set varies at run time would set it to one and accept the cost.

7.4 Delegation and the private library context

An algorithm implementation may compute its result itself or delegate part of the work to another provider. Delegation is useful for a component that is not the subject of the exercise — for instance a design that wishes to study integration mechanics may delegate the underlying primitive while retaining full control of the provider-side interface. The companion paper's provider does exactly this for SHA-256.

Delegation must use a *private* library context, created by the provider and holding only the default provider. Fetching from the application's context would be circular: the provider would be asking the context in which it is itself registered to resolve a name it may itself answer to, and the result depends on property queries the provider does not control. A private context makes the delegate deterministic.

```

ctx->delegate_libctx = OSSL_LIB_CTX_new();
ctx->delegate_md = EVP_MD_fetch(ctx->delegate_libctx, "SHA2-256",
                                "provider=default");

```

The delegate handle is fetched once at provider initialisation and reused, satisfying N1 for the delegated path.

7.5 Context objects and lifecycle

Every operation in the provider interface follows the same lifecycle: a context is created, initialised, used, possibly duplicated, and freed. The provider supplies each step as a dispatch entry. The invariants the design maintains are:

- A newly created context is inert; all algorithm state is established by the `init` call, so a context may be created and freed without ever being initialised.
- Duplication produces an independent context sharing nothing mutable with its source. This is required by §5.3 for the transcript hash and is the operation most commonly implemented incorrectly, because a shallow copy appears to work until one of the two contexts is updated.
- Freeing a context is safe on a null pointer and safe after an error, satisfying N4.
- No context holds a reference to the application's library context.

7.6 Parameter conventions

Requirement F8 obliges every algorithm to report its parameters. The design follows one convention throughout: each algorithm supplies a *gettable* function returning a static descriptor array, and a *get* function that fills a caller-supplied array. Parameters not recognised are skipped rather than treated as errors, which is what allows a newer consumer to ask an older provider for something and proceed without it.

The parameters each algorithm class must answer are specified in the respective chapters and collected in Appendix C.

7.7 Error reporting

Requirement N6 obliges the provider to report through the core rather than to standard error. The core supplies error functions in the incoming dispatch table; the provider resolves them at initialisation and

uses them for every failure that a caller could act on. A provider that prints diagnostics directly is unusable in a server, where the output goes nowhere useful and may itself constitute an information leak.

The design distinguishes three failure classes: *configuration* failures, raised at initialisation and causing activation to fail; *parameter* failures, raised when a caller supplies something the algorithm cannot accept; and *operational* failures, raised when a cryptographic operation cannot be completed. Only the first prevents the provider from loading.

7.8 What this chapter has fixed

The remaining design chapters may now assume: a single module with per-operation units; a provider context created at initialisation and carrying resolved core functions and a private delegate context; a query function dispatching on operation; the create/init/use/dup/free lifecycle with the invariants of §7.5; the parameter convention of §7.6; and core-mediated error reporting. Each algorithm chapter specifies only what is particular to its class.

8. Design of the Block Cipher (AEAD) Component

This chapter specifies the cipher component: the dispatch table it exposes, the parameters it must answer, the lifecycle the record layer imposes, and the constraints inherited from §2.4. Requirements F1 and C2 are addressed here.

8.1 Why the AEAD interface and not a raw block cipher

TLS 1.3 protects records exclusively with AEAD. A raw block cipher is not directly usable: the protocol has no provision for a separate MAC, having removed the MAC-then-encrypt and encrypt-then-MAC constructions that TLS 1.2 supported. A block cipher therefore reaches TLS 1.3 only through an AEAD mode built on it — GCM and CCM being the standard constructions, with counter-mode-plus-polynomial-MAC the general pattern.

This is the first design decision and it is forced: the component this paper specifies is a *mode* over a block cipher, presenting an AEAD interface, not the block cipher in isolation. A provider offering only a block primitive would supply something TLS never fetches.

8.2 The dispatch table

The cipher operation's dispatch table for an AEAD comprises context management, the encrypt/decrypt lifecycle, and parameter access:

```
static const OSSL_DISPATCH tlsext_aead_functions[] = {
    { OSSL_FUNC_CIPHER_NEWCTX,          (void (*)(void))aead_newctx },
    { OSSL_FUNC_CIPHER_FREECTX,         (void (*)(void))aead_freectx },
    { OSSL_FUNC_CIPHER_DUPCTX,          (void (*)(void))aead_dupctx },
    { OSSL_FUNC_CIPHER_ENCRYPT_INIT,     (void (*)(void))aead_einit },
    { OSSL_FUNC_CIPHER_DECRYPT_INIT,     (void (*)(void))aead_dinit },
    { OSSL_FUNC_CIPHER_UPDATE,          (void (*)(void))aead_update },
    { OSSL_FUNC_CIPHER_FINAL,           (void (*)(void))aead_final },
    { OSSL_FUNC_CIPHER_CIPHER,          (void (*)(void))aead_oneshot },
    { OSSL_FUNC_CIPHER_GET_PARAMS,      (void (*)(void))aead_get_params },
    { OSSL_FUNC_CIPHER_GETTABLE_PARAMS, (void (*)(void))aead_gettable_params },
    { OSSL_FUNC_CIPHER_GET_CTX_PARAMS,  (void (*)(void))aead_get_ctx_params },
    { OSSL_FUNC_CIPHER_SET_CTX_PARAMS,  (void (*)(void))aead_set_ctx_params },
}
```

```

    { OSSL_FUNC_CIPHER_GETTABLE_CTX_PARAMS,
      (void (*)(void))aead_gettable_ctx },
    { OSSL_FUNC_CIPHER_SETTABLE_CTX_PARAMS,
      (void (*)(void))aead_settable_ctx },
    { 0, NULL }
};

```

The distinction between *algorithm* parameters and *context* parameters is essential and is a common source of error. Algorithm parameters describe the algorithm itself — its key length, block size and mode — and are answerable without a context. Context parameters describe a particular operation in progress: the tag, the IV length in use, the AAD. The record layer queries the first to size its buffers before it has a context, and the second to retrieve the authentication tag after encrypting.

8.3 Algorithm parameters

The component answers at minimum:

Parameter	Type	Meaning for TLS
keylen	size_t	Bytes the key schedule must derive for the traffic key
ivlen	size_t	Width of the static IV into which the sequence number is folded
taglen	size_t	Ciphertext expansion; needed for record sizing
blocksize	size_t	1 for a stream-like AEAD; used for buffer arithmetic
mode	uint	Identifies the mode as AEAD to the consumer
aead	int	Flag asserting AEAD semantics

Omitting any of the first three does not produce a clear error. It produces a consumer that computes a buffer size from a zero or a default and fails later, in the record layer, with a message about record length. This is the single most valuable debugging observation in the chapter: *record-length errors in a custom AEAD are usually missing parameter answers, not arithmetic bugs.*

8.4 Context parameters and the TLS-specific ones

Beyond the general AEAD context parameters — tag, AAD, IV length — the TLS record layer uses additional parameters to communicate its own framing. A cipher intended for TLS must accept the parameter by which the record layer supplies the fixed portion of the IV, and must answer the parameter reporting how much the ciphertext will expand.

Design. The component treats the IV as supplied, never generated. On `einit` it stores the static IV; for each record the caller supplies the constructed nonce, or the sequence number from which the component derives it, according to the parameter convention in use. Under no circumstances does the component choose a nonce itself (constraint C2). A nonce chosen by the algorithm would be catastrophic here: the AEAD constructions used by TLS lose all confidentiality guarantees on nonce reuse, and the protocol's whole nonce-management discipline assumes the algorithm is passive.

8.5 Lifecycle as the record layer drives it

The record layer's usage pattern is narrow and worth specifying, because a component that works under a test harness may still fail here:

1. Fetch the cipher once per traffic-key epoch, not per record (N1).
2. Create a context and initialise it for encryption or decryption with the traffic key and static IV.
3. For each record: set the per-record nonce or sequence number, supply the record header as AAD, process the plaintext, and retrieve or verify the tag.
4. On key update, initialise a fresh context with the new key; contexts are not re-keyed in place.

Step 3 repeats for the life of the connection with no intervening fetch, which is why per-call overhead rather than fetch overhead governs throughput (§5.4).

8.6 Decryption and the failure contract

Authenticated decryption must not release plaintext before the tag is verified, and must report verification failure as a distinguishable result rather than returning data. In the provider interface this means the final (or one-shot) entry point returns failure on tag mismatch and the caller must treat any output as invalid.

The TLS consequence is severe enough to state explicitly: a record whose tag does not verify must cause the connection to be torn down with a `bad_record_mac` alert. An AEAD component that returns success on a mismatched tag does not produce a subtly weaker connection; it produces one with no integrity protection at all, while appearing entirely normal in testing that never presents a corrupted record. Chapter 19 makes the corresponding negative test mandatory rather than optional.

8.7 Constant-time considerations

Requirement N2 applies with particular force to this component, because it processes attacker-influenced data continuously. Tag comparison must be constant-time; no branch may depend on plaintext or key material; and table lookups indexed by secret data — the classic weakness of naive block cipher implementations — must be avoided. Chapter 17 discusses the limits of what can be guaranteed at the C level.

8.8 Summary

The cipher component presents an AEAD interface over a block cipher, answers key, IV and tag lengths as algorithm parameters, accepts an externally supplied nonce and the record header as AAD, keeps plaintext unreleased until the tag verifies, and is fetched once per epoch. Its failure modes are concentrated in parameter reporting and in the tag-verification contract, and both are testable by deliberately violating them.

9. Design of the Hash Component

The hash component is the smallest of the three in interface and the largest in consequence. It is used by the transcript, by every step of the key schedule, and by the Finished computation, and its output length propagates into every secret the connection holds. Requirement F2 is addressed here; the key-schedule consequences are developed in Chapter 15.

9.1 Dispatch table

```
static const OSSL_DISPATCH tlsext_digest_functions[] = {
    { OSSL_FUNC_DIGEST_NEWCTX,      (void (*)(void))digest_newctx },
    { OSSL_FUNC_DIGEST_FREECTX,     (void (*)(void))digest_freectx },
    { OSSL_FUNC_DIGEST_DUPCTX,      (void (*)(void))digest_dupctx },
    { OSSL_FUNC_DIGEST_INIT,        (void (*)(void))digest_init },
    { OSSL_FUNC_DIGEST_UPDATE,       (void (*)(void))digest_update },
    { OSSL_FUNC_DIGEST_FINAL,        (void (*)(void))digest_final },
    { OSSL_FUNC_DIGEST_GET_PARAMS,   (void (*)(void))digest_get_params },
    { OSSL_FUNC_DIGEST_GETTABLE_PARAMS, (void (*)(void))digest_gettable },
    { 0, NULL }
};
```

This is the table the companion paper's provider implements, and it is reproduced here because the TLS use imposes requirements on two of its entries that a general-purpose digest does not face.

9.2 The duplication requirement

TLS computes the transcript hash repeatedly over a growing prefix of the handshake. The running context is never finalised, because more messages will follow; instead it is duplicated and the copy finalised whenever a digest of the transcript so far is needed — for the signature input, for the Finished computation, and for key-schedule steps that bind the transcript.

`OSSL_FUNC_DIGEST_DUPCTX` is therefore not optional for a TLS handshake hash, and it must produce a genuinely independent context. The failure mode of a shallow implementation is instructive: if the duplicate shares

the source's internal buffer, finalising the copy corrupts the original, and the handshake fails at the Finished check with a message that suggests a MAC error rather than a context-management bug.

Design. The duplication entry allocates a new context and copies the entire algorithm state by value, including any partial input block and the byte counter. Where the component delegates to another implementation (§7.4), duplication must also duplicate the delegate's context — copying the delegate *handle* is precisely the shallow-copy error described above.

9.3 Output length as a first-class parameter

The `size` parameter reports the digest length in bytes. In a general-purpose setting a caller that ignores it merely wastes buffer space; in TLS the value determines the width of the key schedule (§2.3), so an incorrect answer produces secrets of the wrong length and a handshake that fails after several successful-looking steps.

Parameter	Meaning	Consumed by
<code>size</code>	Digest output length in bytes	Key schedule; secret and Finished sizing
<code>blocksize</code>	Internal block size	HMAC construction inside HKDF

The block size matters because HKDF is built on HMAC, and HMAC's key padding and inner/outer construction are defined in terms of the hash's block size. A component that reports an output length but not a block size may work under direct digest use and fail inside HKDF.

9.4 Streaming behaviour

The transcript is supplied incrementally, message by message, in arbitrary sizes determined by the handshake rather than by the hash's block boundaries. The update entry point must therefore buffer partial blocks correctly and produce results identical to a single-shot computation over the concatenation. This is ordinary for a hash implementation, but it is the property most worth testing exhaustively,

because a boundary error appears only for particular input lengths. Chapter 19 proposes testing every length across at least two block boundaries rather than a handful of convenient sizes.

9.5 Naming and aliasing

The `OSSL_ALGORITHM` names field is a colon-separated list, so a component may answer to several names:

```
static const OSSL_ALGORITHM tlsext_digests[] = {
    { "TLSEXT-HASH256:TLSEXT-SHA256", "provider=tlsext",
      tlsext_digest_functions, "Experimental TLS handshake hash" },
    { NULL, NULL, NULL, NULL }
};
```

Naming deserves more care than it usually receives. A component that claims a standard name — SHA2-256 — competes with the default provider for every fetch in the context, including fetches made by code with no relation to TLS. A component that claims only a distinctive name is inert until something asks for it by that name, which is the safer default and the one this design adopts. Chapter 14 treats the policy consequences.

9.6 The retrospective-use problem

§2.2 observed that the handshake hash is selected after some of the data it must cover has already been exchanged. Implementations buffer the early handshake messages and hash them once the cipher suite fixes the algorithm.

The consequence for the component is that its first `update` may be presented with the entire accumulated prefix in one call, and subsequent updates with individual messages. The component must handle both without special-casing, which follows from §9.4 but is worth stating because a component tested only with uniform chunk sizes may not have exercised the pattern.

9.7 Summary

The hash component must duplicate contexts deeply, report both output and block size, stream correctly at arbitrary boundaries, and avoid claiming standard names unless substitution is intended. Its consequences reach further than its interface suggests: Chapter 15 shows that changing this one algorithm changes the width of every secret in the connection.

10. Design of the Elliptic-Curve Group Component

The group component is the one that most clearly demonstrates the provider architecture's reach, because through the `TLS-GROUP` capability a provider introduces a code point that appears on the wire. Requirements F3 and F4 are addressed here.

10.1 Two obligations, not one

Supplying a group means discharging two separate obligations, and the failure to distinguish them is the most common error in this area:

1. **Implementation** — key management and key exchange operations, so the group can be computed.
2. **Advertisement** — a `TLS-GROUP` capability entry, so the group can be negotiated.

A provider that discharges only the first supplies a group that works when fetched directly and never appears in a handshake. The symptom is a connection that negotiates some other group with no error at all, which looks like a preference problem and is really a missing capability entry (§5.5, rule 4).

10.2 Key management

Key management owns the key object: its creation, its parameters, its import and export, and its destruction. For an EC group the design must decide how the curve is identified, and it follows the arrangement OpenSSL itself uses — a single `ec` implementation parameterised by curve name, rather than one implementation per curve. §4.3.1 showed three curves mapping to one algorithm in the built-in list.

```
static const OSSL_DISPATCH tlsext_ec_keymgmt_functions[] = {
    { OSSL_FUNC_KEYMGMT_NEW,          (void (*)(void))ec_newkey },
    { OSSL_FUNC_KEYMGMT_FREE,         (void (*)(void))ec_freekey },
    { OSSL_FUNC_KEYMGMT_GEN_INIT,      (void (*)(void))ec_gen_init },
    { OSSL_FUNC_KEYMGMT_GEN_SET_PARAMS, (void (*)(void))ec_gen_set_params },
}
```

```

    { OSSL_FUNC_KEYMGMT_GEN,          (void (*)(void))ec_gen },
    { OSSL_FUNC_KEYMGMT_GEN_CLEANUP,  (void (*)(void))ec_gen_cleanup },
    { OSSL_FUNC_KEYMGMT_HAS,          (void (*)(void))ec_has },
    { OSSL_FUNC_KEYMGMT_MATCH,        (void (*)(void))ec_match },
    { OSSL_FUNC_KEYMGMT_IMPORT,       (void (*)(void))ec_import },
    { OSSL_FUNC_KEYMGMT_EXPORT,       (void (*)(void))ec_export },
    { OSSL_FUNC_KEYMGMT_IMPORT_TYPES, (void (*)(void))ec_import_types },
    { OSSL_FUNC_KEYMGMT_EXPORT_TYPES, (void (*)(void))ec_export_types },
    { 0, NULL }
};

```

Import and export carry particular weight in TLS. The `key_share` extension carries a public share as an opaque octet string, and the TLS stack obtains that string by exporting the public part of a generated key, then reconstructs the peer's key by importing the received string. The encoding the component uses for export is therefore the encoding that appears on the wire, and it must match what the peer expects exactly — a design that exports an encoding differing in point format or in leading-zero handling produces handshakes that fail against other implementations while succeeding against itself.

10.3 Key exchange

```

static const OSSL_DISPATCH tlsext_ec_keyexch_functions[] = {
    { OSSL_FUNC_KEYEXCH_NEWCTX,      (void (*)(void))ecdh_newctx },
    { OSSL_FUNC_KEYEXCH_FREECTX,     (void (*)(void))ecdh_freectx },
    { OSSL_FUNC_KEYEXCH_DUPCTX,      (void (*)(void))ecdh_dupctx },
    { OSSL_FUNC_KEYEXCH_INIT,        (void (*)(void))ecdh_init },
    { OSSL_FUNC_KEYEXCH_SET_PEER,     (void (*)(void))ecdh_set_peer },
    { OSSL_FUNC_KEYEXCH_DERIVE,      (void (*)(void))ecdh_derive },
    { OSSL_FUNC_KEYEXCH_SET_CTX_PARAMS, (void (*)(void))ecdh_set_ctx_params },
    { 0, NULL }
};

```

The derive operation follows the standard two-call convention: called with a null output buffer it reports the required length, and called with a buffer it produces the shared secret. Both calls must agree, and the length must be the fixed field size for the curve rather than the length of the particular secret computed, since a secret with leading zero bytes must not be shortened. Variable-length shared secrets have produced real interoperability failures and, in some protocols, side channels.

10.4 Peer share validation

This is the security-critical operation of the component and it is specified separately for emphasis. `SET_PEER` receives a share that arrived over the network from an unauthenticated party at that point in the handshake. Before it is used it must be validated.

Design, security-critical. The component shall reject a peer share that is not a valid encoding for the curve, that does not represent a point on the curve, that represents the point at infinity, or that lies in a small subgroup where the curve's cofactor makes that possible. Validation occurs before any operation involving the private key. A component that omits these checks permits invalid-curve attacks, which recover the private key across a sequence of handshakes and leave no trace distinguishable from ordinary connection failures.

For curves whose design makes some of these checks unnecessary — where the encoding guarantees a valid point and the cofactor is handled by the derivation itself — the component should document which checks are subsumed by the construction rather than silently omitting them. A reviewer cannot distinguish "unnecessary" from "forgotten" by reading code that does neither.

10.5 The capability entry

The second obligation is advertisement. The component answers `TLS_GROUP` by invoking the supplied callback once per group with a parameter array of the shape verified in §4.3.1:

```
static int tlsext_group_capability(OSSL_CALLBACK *cb, void *arg)
{
    static const OSSL_PARAM group[] = {
        OSSL_PARAM_utf8_string(OSSL_CAPABILITY_TLS_GROUP_NAME,
                               "tlsext256", sizeof("tlsext256")),
        OSSL_PARAM_utf8_string(OSSL_CAPABILITY_TLS_GROUP_NAME_INTERNAL,
                               "TLSEXT-P256", sizeof("TLSEXT-P256")),
        OSSL_PARAM_utf8_string(OSSL_CAPABILITY_TLS_GROUP_ALG,
                               "EC", sizeof("EC")),
        OSSL_PARAM_uint(OSSL_CAPABILITY_TLS_GROUP_ID, &group_id),
        OSSL_PARAM_uint(OSSL_CAPABILITY_TLS_GROUP_SECURITY_BITS, &secbits),
        OSSL_PARAM_int(OSSL_CAPABILITY_TLS_GROUP_MIN_TLS, &min_tls),
        OSSL_PARAM_int(OSSL_CAPABILITY_TLS_GROUP_MAX_TLS, &max_tls),
        OSSL_PARAM_END
    }
```

```
};
return cb(group, arg);
}
```

and routes the capability request from the provider's dispatch table:

```
static int tlsext_get_capabilities(void *provctx, const char *capability,
                                   OSSL_CALLBACK *cb, void *arg)
{
    if (OPENSSL_strcasecmp(capability, "TLS-GROUP") == 0)
        return tlsext_group_capability(cb, arg);
    return 0;
}
```

10.6 Choosing a code point

The group identifier is a number on the wire, drawn from an IANA registry. A design has three options, with different consequences. A *registered* code point is correct for anything intended to interoperate publicly and requires the IANA process. A code point from the *private-use* range is correct for a closed deployment where both endpoints are under one administration. An *arbitrary unregistered* value is incorrect in all cases and will eventually collide with an allocation, producing a failure that is extremely difficult to diagnose because both peers believe they agreed.

The minimum and maximum TLS version parameters should confine an experimental group to TLS 1.3, preventing its offer in protocol versions whose key-exchange structure differs.

10.7 Summary

The group component discharges two obligations: implementation through key management and key exchange, and advertisement through the capability mechanism. Its export encoding is the wire encoding; its derive length must be the fixed field size; and its peer-share validation is the component's security-critical operation. The code point must come from a registry or from the private-use range, never from neither.

11. Design of the Signature Component

Signature algorithms complete the set. They authenticate the handshake and bind it to a certificate, and like groups they are advertised through a capability and carry a code point. This chapter is shorter than its predecessors because the mechanism is closely analogous to Chapter 10; the differences are in the certificate coupling and in the two-layer naming.

11.1 What must be supplied

A usable signature algorithm requires four things: key management for the key type, the signature operations themselves, an association with an OID so certificates can be matched, and a capability entry so the algorithm can be offered in the `signature_algorithms` extension.

The OID requirement is the structural difference from a group. A group exists only for the duration of a handshake and is identified solely by its code point. A signature algorithm must also be recognisable in a certificate, which is an X.509 structure identifying algorithms by object identifier. An algorithm advertised in the handshake but absent from any certificate the server holds can be negotiated and then not used.

11.2 Dispatch table

```
static const OSSL_DISPATCH tlsext_signature_functions[] = {
    { OSSL_FUNC_SIGNATURE_NEWCTX,      (void (*)(void))sig_newctx },
    { OSSL_FUNC_SIGNATURE_FREECTX,     (void (*)(void))sig_freectx },
    { OSSL_FUNC_SIGNATURE_DUPCTX,      (void (*)(void))sig_dupctx },
    { OSSL_FUNC_SIGNATURE_SIGN_INIT,   (void (*)(void))sig_sign_init },
    { OSSL_FUNC_SIGNATURE_SIGN,        (void (*)(void))sig_sign },
    { OSSL_FUNC_SIGNATURE_VERIFY_INIT, (void (*)(void))sig_verify_init },
    { OSSL_FUNC_SIGNATURE_VERIFY,      (void (*)(void))sig_verify },
    { OSSL_FUNC_SIGNATURE_SET_CTX_PARAMS, (void (*)(void))sig_set_ctx_params },
    { 0, NULL }
};
```

As with key exchange, the signing operation follows the two-call convention for output length. The length reported must be the maximum the algorithm can produce, since the caller allocates before signing.

11.3 The capability entry and the two names

The capability entry carries both an IANA name and an algorithm name, and §4.3.2 verified the parameter set. The distinction is worth drawing out, because it is the mechanism by which a protocol-level identifier is decoupled from an implementation-level one.

Parameter	Example	Used for
IANA name	mldsa65	Protocol-level identity; what configuration names
Algorithm name	ML-DSA-65	What the provider is asked to fetch
OID	2.16.840.1.101.3.4.3.18	Matching certificates
Code point	numeric	The wire
Security bits	numeric	Policy ranking and security-level filtering

The security-bits value is not decorative. OpenSSL's security-level mechanism filters algorithms whose claimed strength falls below the configured level, so an algorithm advertising an understated value may be filtered out of a handshake it could have served, and one advertising an overstated value defeats a control the deployment believes it has.

11.4 Certificate coupling

For the algorithm to be usable end to end, the server must hold a certificate whose subject public key the algorithm can sign with, and the peer must be able to verify that certificate. In practice this means

the key type must be encodable and decodable in the structures certificates use, which brings the encoder and decoder operations of the provider interface into scope.

These are beyond the scope of this design, and the limitation should be stated plainly: a provider supplying a novel signature algorithm for TLS is not finished when signing works. It is finished when a certificate carrying such a key can be parsed, presented, chained and verified by both endpoints. That is a larger undertaking than the signature operation itself, and it is the reason a design of this kind is usually attempted for groups before signature algorithms.

11.5 Relationship to the paper's core design

The three algorithm classes named in this paper's title are the block cipher, the elliptic curve and the hash. The signature component is specified here for completeness — a TLS connection cannot authenticate without one — but a design that supplies the other three may reasonably use a standard signature algorithm from the default provider. Doing so reduces the certificate problem of §11.4 to nothing, and is recommended for a first implementation.

12. Cipher Suite Registration and Negotiation

This chapter addresses requirement F7: a cipher suite combining the AEAD of Chapter 8 and the hash of Chapter 9 shall be usable in a TLS 1.3 handshake. It specifies what a suite is, what registering one entails, how negotiation then proceeds, and what has to be true on both endpoints for a connection to result.

12.1 What a suite is

A TLS 1.3 cipher suite is a two-octet code point that names a pair: an AEAD algorithm for record protection and a hash for the key schedule and transcript. It carries nothing else. The key exchange and authentication methods that TLS 1.2 suites encoded were moved into extensions, which is why groups and signature algorithms are negotiated independently (§2.1).

A suite description, as the TLS implementation needs it, therefore comprises:

Field	Purpose
Code point	The two octets on the wire
Name	Configuration and diagnostics
AEAD algorithm	Fetches for record protection
Handshake hash	Fetches for transcript and key schedule
Version bounds	Restricts the suite to TLS 1.3
Strength metadata	Security level filtering and ordering

Verified. These are the fields the reference implementation's suite entries carry. The first entry of `tls13_ciphers[]` pairs the RFC name and code point macros `TLS_AES_128_GCM_SHA256` with `SSL_AES128GCM`, `SSL_HANDSHAKE_MAC_SHA256`, version bounds of `TLS1_3_VERSION` at both ends, and strength values (`ssl/s3_lib.c:39–56`, tree `openssl-3.5.8`).

12.2 Registration

Registering a suite means making a description of the above shape available to the negotiation logic, and ensuring the algorithms it names are fetchable in the relevant library context. The second half is the work of Chapters 8 and 9 and is already specified. The first half is the subject of this section.

The design requirements for registration are as follows, independently of the mechanism by which the description reaches the negotiation logic:

1. **The code point must be unique** within the deployment and drawn from a registry or the private-use range, on the reasoning given in §10.6. A suite is agreed by number, and two endpoints that attach different meanings to one number will complete a handshake and then fail to communicate — or, worse, communicate under mismatched assumptions.
2. **The named algorithms must resolve** in the library context the `SSL_CTX` uses (constraint C6). A suite naming an algorithm that cannot be fetched is worse than a suite that is absent, because it can be selected and will then fail mid-handshake.
3. **The version bounds must confine it to TLS 1.3**, since the suite structure and key schedule of earlier versions differ.
4. **The strength metadata must be accurate**, for the security-level reasons given in §11.3.
5. **Ordering must be deliberate.** The server selects from the client's offer according to its own preference ordering; a new suite placed above standard suites will be selected whenever a peer supports it,

and placed below will be selected only when nothing else matches. Both are defensible; neither should happen by accident.

Reported, with a verified counterpart. The system this paper documents is reported by its author to add such suites through the provider interface. What this study verified independently is the structure of the suite description and the location of the suite table in the reference implementation (§12.1, and §4.4). Whether a given OpenSSL build obtains suite descriptions from a provider, from its built-in table, or from both is a property of that build; a deployment should establish it by inspection of the build in use, and Chapter 19 gives the test that settles the question empirically — `openssl ciphers` against a build with the provider activated and deactivated.

12.3 Negotiation

Once a suite exists on both endpoints, negotiation is ordinary and is worth stating because the failure modes are diagnosable only if the sequence is understood.

The client sends its supported suites in preference order in `ClientHello`, together with its groups and signature algorithms. The server intersects the client's list with its own, applies its preference policy, and returns a single selected suite in `ServerHello`. From that point both sides fix the AEAD and the hash, and the key schedule begins.

Three failure modes follow, and they are distinguishable:

No shared suite

The server finds an empty intersection and sends `handshake_failure`. The custom suite was offered by one side only — typically the provider is not activated on the other, or is activated in a different library context.

Suite selected, algorithm unavailable

The handshake proceeds past `ServerHello` and fails when an algorithm cannot be fetched. The suite description was present but the implementation was not — requirement 2 of §12.2 violated.

Suite selected, handshake fails at Finished

Both sides agreed and computed, but computed differently. This points at the hash: a transcript divergence, a duplication bug (§9.2), or a key-schedule width mismatch (§15).

The diagnostic value of separating these is high. The first is a configuration problem, the second a deployment problem, the third an implementation bug — and they are commonly reported identically, as "the handshake fails".

12.4 Both endpoints must change

A point that is obvious in principle and routinely underestimated in planning: a new suite is useless unilaterally (constraint C4). Unlike a path-1 implementation substitution, which is invisible to the peer, a new suite requires the peer to recognise the code point and to possess implementations of both named algorithms.

This bounds the applicability of the whole technique. A new suite is deployable where both endpoints are under one administration — internal services, an organisation's own clients, an embedded fleet — and is not deployable against the public web, where the peer is a browser whose suite list the deployment does not control. Chapter 16 develops the interoperability consequences.

12.5 Interaction with the other three paths

A suite introduces an AEAD and a hash. It does not introduce a group or a signature algorithm, which remain separately negotiated. A design that introduces all of them must therefore discharge the suite obligation *and* the two capability obligations; satisfying one does not imply the others.

This is the practical pay-off of the taxonomy in Chapter 4. A requirement phrased as "use our national algorithms in TLS" typically decomposes into a suite (for the cipher and hash), a TLS-GROUP entry (for the curve), and a TLS-SIGALG entry plus certificate work (for the signature) — three different mechanisms, of which only the first is the subject of this chapter.

13. Configuration, Activation and Deployment Mechanics

Requirement F6 asks that the provider be activatable without modifying the application, and constraint C6 requires that algorithms be available in the library context the `SSL_CTX` actually uses. This chapter specifies the configuration and states the failure modes, which in deployment are more often the cause of trouble than the cryptography is.

13.1 The configuration file

```
openssl_conf = openssl_init

[openssl_init]
providers = provider_sect
ssl_conf  = ssl_sect

[provider_sect]
default = default_sect
tlsext  = tlsext_sect

[default_sect]
activate = 1

[tlsext_sect]
module    = /usr/local/lib/openssl-modules/tlsext.so
activate = 1

[ssl_sect]
system_default = system_default_sect

[system_default_sect]
Groups      = tlsext256:x25519:secp256r1
CipherSuites = TLS_TLSEXT_AEAD_TLSEXT_HASH256:TLS_AES_256_GCM_SHA384
```

Three things are happening here and they are independent. The `provider_sect` activates modules. The `Groups` line sets which groups are offered and in what order, naming the group by the IANA name from its capability entry (§10.5). The `CipherSuites` line sets the TLS 1.3 suite list and ordering (§12.2, requirement 5). A deployment that activates the provider but omits the last two lines has loaded the algorithms without asking for them, and will observe a perfectly ordinary standard handshake.

Design. The provider reads its own configuration from its section, so deployment-specific choices — a code point for a closed deployment, a delegate selection — are set here rather than compiled in. Configuration errors are raised at activation, as configuration failures in the sense of §7.7, so that a misconfigured provider fails to load rather than loading in a degraded state.

13.2 Activation is not selection

The distinction in the previous paragraph is worth a section of its own, because it accounts for a large share of "the provider does not work" reports.

Activating a provider makes its algorithms *available*. It does not make them *chosen*. Selection happens by fetch, and a fetch that does not name the algorithm, or whose property query does not distinguish it, will resolve to whatever the ordinary rules prefer — typically the default provider's implementation. For the TLS case, selection is driven by the negotiated parameters, which are in turn driven by the Groups and CipherSuites configuration above.

The practical test is in two parts: confirm the provider is loaded, then confirm it is used. The two are separate observations and the first does not imply the second.

```
$ openssl list -providers
$ openssl list -digest-algorithms -provider tlsext
$ openssl s_client -connect host:443 -tls1_3 </dev/null | grep -E 'Cipher|Group'
```

13.3 Library context traps

Constraint C6 appears in deployment as follows. An `SSL_CTX` is bound to a library context when it is created. An application that creates its `SSL_CTX` against an explicit context, while the configuration file activates the provider in the default context, will not see the provider at all — the configuration is correct, the module is present, and the algorithms are invisible.

Applications that use only the default context, which is the large majority, are unaffected. Applications built around explicit contexts for isolation — including anything using a private context for a FIPS-related purpose — must activate the provider in the context they use, which generally means programmatic loading rather than the configuration file.

13.4 Ordering and precedence

When two activated providers offer the same algorithm name, the resolution depends on property query and on ordering. A deployment that wants determinism should not rely on ordering; it should name the provider in the property query (§14) and treat the requirement as mandatory rather than preferred.

The recommendation of §9.5 — that a component claim a distinctive name rather than a standard one — removes this question entirely for the algorithms it applies to, at the cost of requiring explicit configuration to use them. For a deployment whose purpose is to use particular algorithms, that explicitness is a feature.

13.5 Installation and integrity

The module is a shared object loaded into every process that activates it, with the privileges of that process (constraint C5). The deployment consequences follow directly:

- The module and the configuration file must be writable only by an administrative account. Write access to either is equivalent to code execution in every process that loads it.
- The module should be distributed through the same signed channel as other system software, so that its authenticity is established by the mechanism already trusted for the rest of the system.
- Where the deployment must be able to state which implementation performed an operation, the module's identity and hash should be recorded, since the connection itself does not carry that information (§5.2).

13.6 Rollback

A deployment plan should include the reverse operation, and it is simple: set `activate = 0` in the provider section, or remove the section, and restart the consuming processes. Because the configuration is data rather than code, rollback does not require rebuilding anything.

The one caveat is the fail-closed behaviour of §6.4. If the deployment has configured the custom suite or group as the only acceptable option, deactivating the provider does not restore standard connectivity — it removes connectivity, which is the intended behaviour of a fail-closed configuration and a surprise to whoever performs the rollback expecting a return to normal. The rollback procedure must therefore restore the algorithm configuration, not only the provider activation.

14. Property Queries as Selection Policy

The property mechanism is how a deployment states which implementation it will accept. This chapter treats it as what it is — a policy language — and addresses requirements F5 and F10.

14.1 The language

A property query is a comma-separated sequence of clauses. A clause may require a property to have a value, require it to be absent, or express a preference rather than a requirement. A query is evaluated against the properties each implementation declares in its `OSSL_ALGORITHM` entry (§3.3).

Query	Meaning
<code>provider=tlsext</code>	Only implementations from that provider are acceptable
<code>provider!=default</code>	The default provider's implementations are excluded
<code>?provider=tlsext</code>	Prefer that provider, but accept another
<code>provider=tlsext,fips=yes</code>	Both conditions required

The distinction between the first form and the third is the distinction between a requirement and a wish, and it is the whole of §6.4 expressed in one character.

14.2 Mandatory versus preferred, and why it matters

A deployment introduces a particular algorithm for a reason — a regulatory obligation, an internal standard, a migration. If that reason is genuine, then a connection that does not use the algorithm has not satisfied it, and should not be established.

Expressing the requirement as a preference produces the opposite behaviour: when the provider fails to load, when the configuration names the wrong path, when the module is present but activated in a different library context, the fetch quietly resolves to the default implementation and the connection succeeds. The deployment believes it is compliant. Nothing in the connection indicates otherwise.

The failure is silent and it is the common case. Every mechanism in this chapter works correctly; the deployment simply asked the wrong question. This is why requirement F10 exists and why Chapter 19 makes the negative test — deactivate the provider, assert the handshake fails — a required test rather than a thorough one.

14.3 Properties a provider should declare

The design declares, on each algorithm:

- `provider=tlsext` — the identity, allowing a deployment to require this module specifically;
- a property naming the algorithm family or role, where a deployment might reasonably want all algorithms of a kind without naming each;
- nothing else. Each declared property is a commitment that something may be selected by, and a property whose meaning is not precisely defined will eventually be relied upon with a different meaning in mind.

The design specifically does not declare `fips=yes`. The provider is not a validated module (§6.5), and declaring the property would allow it to be selected by a query intended to restrict selection to validated implementations — defeating the control entirely. A property is an assertion, and asserting something untrue in a selection language is a vulnerability, not a configuration choice.

14.4 Where the query is set

A property query may be supplied per fetch, or set as a default for a library context. For the TLS case the application does not perform the fetches — `libssl` does — so the per-fetch route is unavailable to a deployment that is not modifying the application. The context-wide default is therefore the mechanism in practice, set programmatically or through configuration.

This has a consequence that deserves care. A context-wide default applies to *every* fetch in that context, including fetches by code with no relation to TLS. A default of `provider=tlsext` in a context where the provider offers only three algorithms will cause every other fetch in that context to fail. The usable form is one that constrains only what the deployment means to constrain, which in practice means naming the algorithms distinctively (§9.5) and configuring TLS to ask for them by name, rather than imposing a broad property default.

14.5 Diagnosis

When selection does not behave as expected, the questions in order are: is the provider loaded; does it advertise the algorithm under the name being fetched; does the property query admit it; and is the fetch happening in the context where it is activated. Each has a direct observation, and taking them in order is faster than reasoning about the outcome.

```
openssl list -providers                # loaded?
openssl list -digest-algorithms -provider tlsext # advertised, under what name?
openssl list -digest-algorithms -propquery 'provider=tlsext' # admitted by
query?
```

The fourth question — the context — has no command-line observation, because the command-line tool uses the default context. It is answerable only by inspecting the application, which is why §13.3 treats it as a deployment hazard rather than a diagnostic step.

15. Key Schedule Integration

Constraint C3 stated that a change of handshake hash changes the width of the entire key schedule. This chapter develops that claim, because it is the most invasive consequence of the design and the one most likely to be discovered late.

15.1 The schedule

TLS 1.3 derives every secret through HKDF, structured as a sequence of `Extract` and `Expand` operations. Schematically, an early secret is extracted from the pre-shared key or zero; a handshake secret is extracted from the shared secret produced by key exchange; and a master secret follows. From each, traffic secrets are expanded, and from those, keys and IVs.

Two properties of this structure govern the design:

1. **Every step uses the suite's hash.** HKDF is HMAC-based, and the HMAC uses the negotiated hash. There is no point in the schedule where a different algorithm appears.
2. **Secret widths equal the hash output length.** The extract step produces a pseudorandom key of the hash's output size, and every derived secret inherits it.

Consequently the schedule's arithmetic is parameterised by one number, and that number comes from the provider's `size` parameter (§9.3).

15.2 What changes when the hash changes

Quantity	With a 32-byte hash	With a 48-byte hash	Source of the value
Early / handshake / master secret	32 bytes	48 bytes	Hash output length

Quantity	With a 32-byte hash	With a 48-byte hash	Source of the value
Traffic secrets	32 bytes	48 bytes	Hash output length
Finished key and value	32 bytes	48 bytes	Hash output length
Transcript hash	32 bytes	48 bytes	Hash output length
Record key	AEAD key length — unchanged by the hash		Cipher keylen
Record IV	AEAD IV length — unchanged by the hash		Cipher ivlen

The last two rows are the useful observation: the record keys are *expanded* to the cipher's required lengths and do not inherit the hash width. The AEAD and the hash are therefore loosely coupled — a suite may pair any hash with any AEAD, provided both report their parameters correctly. This is why §8 and §9 could be specified independently.

15.3 Where a wrong length manifests

A hash component that misreports its output length does not fail at the point of the error. It fails downstream, and the distance between cause and symptom is the reason this chapter exists.

Understated length

Secrets are truncated. The handshake may proceed to the Finished exchange and fail there, because the two endpoints derived different values. The symptom is a MAC failure; the cause is a parameter.

Overstated length

The consumer allocates and expects more bytes than the algorithm produces. Depending on the consumer this is a buffer of uninitialised or zero bytes entering the schedule — which may still be deterministic on both sides and therefore may still *succeed*, producing a connection whose keys have less

entropy than they appear to. This is the worse of the two failures, because it does not announce itself.

Correct length, wrong block size

HMAC's padding is wrong, so every HKDF step is wrong, and the failure is total and immediate.

Security-relevant. The overstated-length case can produce a working connection with weakened keys. A test suite that checks only that handshakes succeed will pass. The test that catches it is a known-answer test over the key schedule itself — derive from fixed inputs and compare against values computed independently — which Chapter 19 accordingly requires.

15.4 The transcript

The transcript hash covers all handshake messages in order, and it is consumed at several points without being finalised (§9.2). The design obligations are therefore concentrated in duplication rather than in the hashing itself.

A subtle requirement follows from §2.2: because the hash is not known until the suite is selected, the early messages are buffered and hashed retrospectively. Any divergence between the bytes as sent and the bytes as buffered produces a transcript mismatch, detected at Finished. When a custom hash is in use, this ordinary failure mode is easily misattributed to the new algorithm; distinguishing them requires comparing the transcript input on both endpoints, not the digest output.

15.5 Exporters and resumption

Two features extend the schedule's reach beyond the connection and are easily overlooked.

Exporters allow an application to derive keying material from the connection, and the exported length is requested by the application while the underlying derivation uses the suite's hash. An application

that assumed a particular hash — for instance in a protocol that binds channel identity to an exporter of fixed size — interacts with a changed hash in ways the TLS layer cannot detect.

Resumption stores a secret whose width is the hash's output length, and a resumption attempt must use a suite with the same hash as the original connection. A deployment introducing a custom hash must ensure its session cache records which hash produced each entry, or resumption will fail in ways that look intermittent — succeeding when the same suite happens to be selected again and failing otherwise.

15.6 Summary and design rule

The key schedule is parameterised by exactly one provider-supplied value, and that value must be correct. The design rule that follows is narrow and worth stating as such:

Rule. A hash component intended as a TLS handshake hash must be validated by comparing key-schedule outputs against independently computed values, not merely by comparing digests. Digest correctness does not imply schedule correctness, because the schedule depends on the reported length and block size in addition to the computation.

16. Interoperability

A TLS connection has two ends. Every extension mechanism in Chapter 4 except the first introduces something the peer must recognise, and this chapter treats the consequences systematically. Constraint C4 is the subject.

16.1 The asymmetry between the paths

Path	Peer must change	Deployable against
1. Implementation substitution	No	Anything, including the public web
2. New group	Yes	Both endpoints under one administration
3. New signature algorithm	Yes, plus certificate support	As above, with a PKI that issues such certificates
4. New cipher suite	Yes	Both endpoints under one administration

The table is the chapter's main content, and the design consequence is that paths 2–4 belong to closed deployments. Internal service meshes, an organisation's own client software, embedded fleets with managed firmware, and test environments are all appropriate; a public-facing web server is not, because the peer is a browser whose algorithm support the deployment does not control and will not change.

16.2 Graceful degradation, and why it is a trap here

The natural response to the above is to configure the custom algorithms alongside the standard ones, so that peers supporting them use them and others fall back. This works, and for a migration it is the right configuration.

It is a trap when the deployment's purpose is compliance, for exactly the reason given in §14.2: a fallback that succeeds silently produces connections that do not meet the requirement the deployment exists to meet, and produces them indistinguishably from ones that do. A deployment must decide which of these it is — migration or requirement — and configure accordingly. It cannot have both properties at once, and configurations that appear to offer both are in fact migration configurations with a compliance intention attached.

16.3 Version interaction

The design confines itself to TLS 1.3 (§6.5), and the capability entries carry minimum and maximum version parameters to enforce it (§10.6). The reason is structural: TLS 1.2 suites encode key exchange and authentication in the suite identifier, and its key derivation differs from the 1.3 schedule. An algorithm offered in both without regard to the difference will be negotiated in a version whose machinery it does not fit.

A deployment that must support TLS 1.2 peers should do so with standard algorithms and confine the custom path to 1.3, rather than attempt to span both.

16.4 Middleboxes and observability

Two practical effects deserve mention. Network equipment that inspects handshakes may reject or mishandle unrecognised code points; this is the ordinary experience of deploying anything new in TLS, and it is discovered in testing against the actual network path rather than in a laboratory.

Conversely, monitoring infrastructure that identifies connections by cipher suite will report the custom suite as unknown. Security monitoring that alerts on unrecognised suites will alert on every connection the deployment intended to create — a false positive generated by the deployment itself, which should be anticipated rather than discovered during an incident.

16.5 A testing recommendation

Interoperability claims should be established against an implementation that is not the one under development. Two endpoints built from the same source with the same provider will agree with each other even when both are wrong — the shared-bug failure mode, which is precisely what interoperability testing exists to detect.

Where a second independent implementation is unavailable, the weaker but still useful substitute is to verify against independently computed test vectors at each layer: the algorithm outputs, the key schedule (§15.6), and the wire encodings (§10.2). This does not establish interoperability, and a paper reporting such testing should not claim that it does.

17. Security Analysis

This chapter analyses the design's security properties. It is organised around the trust relationships the provider mechanism creates, because those are what the design changes; the security of the underlying primitives is assumed and is not the subject.

17.1 The provider is in the trusted computing base

Constraint C5 stated it plainly: a provider executes with the full privileges of the process that loads it. There is no sandbox, no capability restriction and no memory isolation between the provider and the application. A provider can read any memory the process can, including private keys held by other parts of the application.

This is not a weakness of the design; it is the nature of dynamic linking, and the `ENGINE` mechanism it replaced had the same property. But it must be stated, because the packaging invites a different intuition: a provider looks like a plug-in, and plug-in architectures in other domains often carry isolation guarantees. This one does not.

Three consequences follow for deployment. The module must be treated as privileged code and protected accordingly (§13.5). The supply chain that delivers it is as security-critical as the one delivering OpenSSL itself. And a compromise of the module is a compromise of every process that loads it, not a degradation of the algorithms it supplies.

17.2 Threat model

The design assumes a network attacker with the standard capabilities: observing, modifying, injecting and replaying traffic, and initiating connections. It assumes the endpoints are not compromised and the module is authentic. It assumes the underlying primitives are secure against the attacker's computational resources.

Out of scope: an attacker with code execution on an endpoint (against whom the design offers nothing, per §17.1); physical attacks; and attacks on the primitives themselves.

17.3 Attack surface introduced by each component

Component	Attacker-controlled input	Principal risk
AEAD decrypt	Every ciphertext record	Releasing unverified plaintext; nonce misuse; timing in tag comparison
Hash update	Handshake messages	Buffer handling at block boundaries; state confusion via duplication
Key exchange	Peer key share	Invalid-curve and small-subgroup attacks
Signature verify	Peer signature and certificate	Malleability; accepting malformed encodings

Each row names an operation processing data from an unauthenticated party, which is the definition of attack surface. The key exchange row is the most serious because its failure mode recovers a private key rather than breaking a single connection.

17.4 The three failures that matter most

17.4.1 Missing peer-share validation

Specified as a requirement in §10.4 and repeated here because it is the design's most dangerous single omission. An implementation that performs a scalar multiplication with the private key against an attacker-chosen point that is not on the intended curve leaks information about the private key, and a sequence of such handshakes recovers it. The attack requires only the ability to connect, and its traffic is indistinguishable from ordinary failed handshakes.

17.4.2 Releasing unverified plaintext

Specified in §8.6. An AEAD that returns plaintext before or despite tag verification removes integrity protection from the connection while leaving it apparently functional. The corresponding requirement is that decryption failure is reported as failure and no output is used.

17.4.3 Silent fallback

Specified in §6.4 and §14.2. This is not a cryptographic failure but a policy failure, and it is included among the three because it is the most likely of them to occur in a real deployment: every component works correctly, the connection is secure by ordinary standards, and the deployment's actual requirement is unmet without any signal.

17.5 Side channels

Requirement N2 asks for constant-time behaviour on secret data. The obligations are the standard ones — no secret-dependent branches, no secret-indexed memory access, constant-time comparison of authentication tags and of any value derived from a key.

The honest qualification is that these cannot be guaranteed at the C level. A compiler may introduce a branch where the source has none, and a processor may introduce timing variation the source cannot control. Implementations that take this seriously verify the property on the compiled artefact, with tooling that examines the binary or measures timing distributions directly. A design that claims constant-time behaviour without such verification is claiming an intention rather than a property, and this design claims the intention.

17.6 What the design does not weaken

It is worth stating the negative result as well. Because a provider cannot alter protocol sequencing (constraint C1), the design cannot introduce a downgrade, cannot suppress authentication, and cannot change what the handshake binds. The protocol-level security

properties of TLS 1.3 are preserved by construction, and the analysis reduces to the correctness of the algorithms supplied and the policy governing their selection.

That reduction is the architecture's principal security contribution, and it is worth appreciating: an extension mechanism that *could* alter sequencing would require a security analysis of every possible provider, rather than of the algorithms one provider supplies.

17.7 Validation status

The provider is not a FIPS-validated module and declares no property claiming otherwise (§14.3). A deployment with validation obligations cannot discharge them with this design; it would require the module to undergo the CMVP process, with the algorithm testing, the integrity self-test and the security policy that entails. The relationship between this design and a validated module is that they use the same loading mechanism and nothing else.

18. Performance: A Proposed Evaluation Methodology

This chapter proposes how the design's performance would be evaluated. It reports no measurements, because none were taken. The distinction is maintained deliberately: a methodology is a contribution, and invented numbers are not.

18.1 Two costs, commonly conflated

Provider performance discussions usually merge two distinct costs.

Fetch cost is the price of resolving an algorithm name and property query to an implementation. The OpenSSL project states that this involves searching by name and is much slower than directly accessing a method table, recommending prefetching for algorithms used many times. It is paid per fetch.

Operation cost is the price of the cryptography itself, paid per byte or per operation.

For TLS these have entirely different profiles. A handshake performs a bounded number of fetches; a bulk transfer performs one AEAD operation per record for the life of the connection. Fetch cost therefore affects connection establishment rate, and operation cost affects throughput. A benchmark that does not separate them measures a mixture whose proportions depend on the connection lifetime it happened to choose.

18.2 Proposed measurements

#	Measures	Method	Expected sensitivity
M1	Fetch latency	Time a single fetch, cold and warm, for each algorithm	Property query complexity; number of activated providers

#	Measures	Method	Expected sensitivity
M2	Handshake rate	Complete handshakes per second, one connection at a time	Fetch cost; key exchange cost
M3	Throughput	Bytes per second over an established connection	AEAD operation cost only
M4	Record-size sensitivity	M3 across record sizes from small to maximum	Per-call overhead relative to per-byte cost
M5	Prefetch benefit	M2 with and without prefetched algorithm objects	Isolates the cost N1 exists to avoid

M4 deserves comment. Per-call overhead is amortised over the record for large records and dominates for small ones, so a component with high fixed cost looks acceptable at maximum record size and poor for interactive traffic. Measuring one record size and reporting it as throughput conceals exactly the case that matters for latency-sensitive applications.

18.3 Baselines

Every measurement requires a comparison, and the meaningful baseline is the default provider's implementation of the nearest standard algorithm, measured on the same machine in the same session. Absolute figures are uninformative: they describe the test machine.

Where the design delegates (§7.4), a second baseline is valuable — the delegate measured directly, without the provider indirection. The difference between the two isolates the cost of the provider layer itself, which is the quantity a study of integration mechanics actually wants to know.

18.4 Controls

A measurement of this kind is easy to perform and hard to perform meaningfully. The proposed controls are:

- Fix the processor frequency governor; scaling makes repeated runs incomparable.
- Pin to a core and report whether hardware acceleration for the standard algorithm is available, since a hand-written implementation compared against an accelerated one measures the instruction set rather than the design.
- Report the distribution, not a single figure; the median and a high percentile at minimum.
- Repeat across at least three separate process invocations, since first-use costs and allocator state differ.
- State the OpenSSL version, the build options and the set of activated providers.

18.5 Threats to validity

Stated in advance, as they should be for any proposed evaluation:

Implementation quality confound

A comparison against an accelerated standard implementation measures optimisation effort, not architecture. Any claim about the provider mechanism's overhead must isolate it as in §18.3.

Single-machine results

Cache sizes and instruction sets differ; results transfer poorly.

Loopback measurement

Handshake rate over loopback omits network latency, which dominates in reality. The figure is useful for comparing implementations and misleading as a prediction of deployed behaviour.

Microbenchmark bias

M1 in a tight loop measures a warm cache state that a real application will not have.

18.6 What would be reported

An executed study following this methodology would report, for each measurement, the distribution across runs, the baseline, the ratio with a confidence interval, and the environment. It would not report a single speed figure, and it would not compare against published numbers from other machines.

Until such a study is performed, the correct statement about this design's performance is that it is unmeasured. This chapter exists so that the measurement, when made, is made properly.

19. Proposed Validation Strategy

This chapter proposes the tests that would establish the design works. It is a plan, not a log: no tests were executed in this study, and the chapter is written so that an implementation phase can be checked against it. The governing principle is from §6.4 — *a check that has never been observed to fail has not been tested* — so every requirement that can be violated has a test that violates it.

19.1 Levels

Level	Establishes	Independent of
L1 Algorithm	Each algorithm computes correct values	OpenSSL
L2 Provider	Algorithms are reachable through the provider interface	TLS
L3 Key schedule	Derivations match independently computed values	The network
L4 Handshake	A connection is established using the algorithms	—
L5 Interoperability	A different implementation agrees	The implementation under test

The levels are ordered by cost and by diagnostic value. A failure at L4 with L1–L3 passing localises the problem to integration; a failure at L4 with L3 untested localises it nowhere.

19.2 L1: algorithm correctness

Known-answer tests against published vectors, for each algorithm, including:

- every input length from zero through at least two block boundaries, for the hash — the boundary cases are where streaming bugs live (§9.4);

- the standard AEAD vectors, plus a case with empty plaintext and a case with empty AAD;
- key exchange against fixed key pairs with known shared secrets;
- a test that the transcript-style usage — many small updates — agrees with a single-shot computation over the concatenation.

19.3 L2: provider integration

- The provider loads and appears in `openssl list -providers`.
- Each algorithm is listed under the expected name with the expected properties.
- Fetching with the required property succeeds; fetching with a property the provider does not satisfy fails (requirement F10).
- Context duplication produces independent state: duplicate a partially updated hash context, finalise the copy, continue the original, and confirm both results are correct. This is the test for §9.2 and it fails loudly on a shallow copy.
- The provider coexists with the default provider without displacing it (F9): fetch a standard algorithm and confirm it still resolves.
- Teardown frees all state, verified under a leak checker.

19.4 L3: key schedule

Required by §15.6 and the most valuable single test in the plan, because it catches the silent-weakening failure of §15.3.

- Derive each secret in the schedule from fixed inputs and compare against values computed by an independent implementation of HKDF with the same hash.
- Confirm every derived secret has the length the hash reports.
- Negative test: a deliberately misreported output length must cause a detectable failure, not a silently shorter secret.

19.5 L4: handshake

- A TLS 1.3 handshake completes using the custom suite and group, between a client and server both configured for them.
- The negotiated suite and group, as reported by both endpoints, are the intended ones — not merely that a connection was established (§14.2).
- Data transfers correctly in both directions, including records at the maximum size and records of one byte.
- Key update completes and traffic continues.
- Session resumption either works or fails cleanly, with the hash recorded per §15.5.

19.5.1 Required negative tests

These establish that the mechanisms discriminate, and they are not optional:

Test	Expected	Guards
Deactivate the provider, retry with the custom suite required	Handshake fails	F10, silent fallback (§14.2)
Corrupt one byte of a record	Connection torn down with <code>bad_record_mac</code>	§8.6 tag verification
Present a peer key share that is not on the curve	Share rejected before any private-key operation	§10.4 invalid-curve
Present a small-subgroup point	Rejected	§10.4
Offer the suite to a peer without it	Clean <code>handshake_failure</code> , no fallback	§12.3
Configure the provider in a different library context	Algorithms unavailable, diagnosable	C6, §13.3

The second and third rows are the tests that distinguish a working security mechanism from one that has merely never been challenged. A test suite omitting them can pass completely against an implementation with no integrity protection and a recoverable private key.

19.6 L5: interoperability

Against an independent implementation, per §16.5. Where none exists, the chapter's recommendation is to say so rather than to substitute same-source testing and describe the result as interoperability.

19.7 Reporting

Results should record, for every case, what was executed and what was observed, with the environment. Following the companion paper's practice, counts of assertions should not be presented as counts of scenarios: three thousand bitwise comparisons within one test are one test, and reporting them as thousands of passing checks overstates coverage.

A results chapter should also record what was *not* tested. The absence of an interoperability result, of a constant-time verification (§17.5), or of a validated-module claim (§17.7) should appear in the report, because a reader cannot distinguish an untested property from a passing one by its absence.

20. Discussion, Limitations and Conclusion

20.1 What this study establishes

The paper set out to determine how a block cipher, an elliptic-curve group and a hash function can be supplied to a TLS 1.3 connection through the OpenSSL 3 provider interface. Its findings are of three kinds.

The **architectural finding** is the taxonomy of Chapter 4. Four distinct paths lead from a provider into a TLS connection, and they differ in mechanism, in reach and in what they demand of the peer. Supplying an implementation for an algorithm TLS already names requires nothing of the peer and adds nothing to the protocol. Advertising a group or a signature algorithm through the capability mechanism introduces a new code point, and this is a genuine extension point — OpenSSL introduces post-quantum signature algorithms to TLS through the same interface a third-party provider would use. Cipher suites are obtained by the TLS library from its own suite table, which makes a new suite a different kind of undertaking from a new group.

The **design finding** is that the three algorithm classes are not symmetric in difficulty. The AEAD is comparatively self-contained: report three lengths, accept an external nonce, never release unverified plaintext. The group carries the security-critical obligation of peer-share validation and the dual obligation of implementing and advertising. The hash is the smallest interface and the largest consequence, because its output length parameterises the entire key schedule, and an error in that single reported number can produce a working connection with weakened keys.

The **methodological finding** is that the failure modes of this kind of system concentrate in places a happy-path test suite does not reach: silent fallback when a provider fails to load, accepted plaintext on a

failed tag check, unvalidated peer shares, and misreported lengths. Chapter 19's insistence on negative tests follows from this and is, in the author's view, the most transferable part of the paper.

20.2 Limitations

These are stated plainly, as Chapter 1.5 promised.

- **Nothing was built.** This is a design study. No provider was compiled, no handshake was performed, no measurement was taken. The specifications in Chapters 7-15 are detailed enough to implement and have not been implemented.
- **The author's reported capability was not reproduced.** The system described as adding cipher suites through the provider interface was not available to this study. Chapter 12 specifies what such a capability entails and Chapter 19 gives the test that would demonstrate it; Appendix D records the claim as reported rather than verified.
- **Verified claims are limited to the reference trees.** Statements marked verified were checked against `openssl-3.5.8` and the pinned `openssl-3.5.5` documentation. Another build — with different configuration options, or patched by a distribution — may differ, and §12.2 recommends establishing the behaviour of the build actually in use.
- **No interoperability claim is made**, and none could be: §16.5 requires a second independent implementation, which a design study does not have.
- **Constant-time behaviour is an intention, not a verified property** (§17.5).

20.3 What an implementation phase should produce

For the design to be considered validated, an implementation phase would need to produce, at minimum:

1. A building provider module implementing Chapters 8-11, with the parameter answers of Appendix C.

2. L1-L3 test results per Chapter 19, including every negative test of §19.5.1.
3. A handshake log showing the negotiated suite and group, from both endpoints.
4. A leak-checked teardown result.
5. A statement of what was not tested.

Items 2 and 5 are where studies of this kind most often fall short, and they are the items that distinguish a demonstration from a validation.

20.4 Future work

Three directions follow naturally. The first is the implementation phase above, which would convert the design chapters from specification to evidence. The second is an empirical study of the fetch-cost question raised in Chapter 18: the project's own documentation warns that name-based lookup is much slower than a method table, and the practical significance of that warning for handshake-rate-limited services appears not to have been measured publicly. The third is the certificate problem of §11.4 — introducing a novel signature algorithm end to end, including encoding, chain construction and verification — which is a larger undertaking than the signature operation and is where a provider-based approach meets its most substantial obstacle.

20.5 Conclusion

The provider architecture delivers a substantial part of what cryptographic agility promises. A new elliptic-curve group can be introduced into a TLS 1.3 handshake by a loadable module, advertised on the wire with its own code point, and selected by negotiation, with no change to the TLS library and no change to the application — and OpenSSL's own post-quantum work uses that same mechanism, which is the strongest evidence that it is a real extension point rather than an internal convenience.

The remaining distance is instructive. Cipher suites are obtained differently from groups and signature algorithms, so a new suite is a different kind of work from a new group. Protocol sequencing is deliberately beyond a provider's reach, which is a security property rather than a limitation. And the closer an extension comes to the protocol's own structures — a new code point, a new key-schedule width, a new certificate algorithm — the more it requires of the peer, until the technique is confined to deployments where both endpoints are under one administration.

An engineer approaching this problem should therefore begin not with the implementation but with Chapter 4's question: which of the four paths does this requirement actually need? In the common case the honest answer is a smaller path than the one first imagined, and the design that follows is correspondingly more likely to reach production.

Appendix A. Provider Skeleton

The following is the specified skeleton of the provider described in Chapters 7–11. It is a **specification expressed as code**, not a tested artefact: it was not compiled, and it omits the algorithm bodies, which are the subject of the respective chapters. Error handling is shown where it carries design content and elided as `/* ... */` where it does not.

A.1 Provider core

```
/*
 * tlsext provider -- core entry points.
 * Specification skeleton accompanying Chapter 7. Not compiled.
 */
#include <openssl/core.h>
#include <openssl/core_dispatch.h>
#include <openssl/core_names.h>
#include <openssl/params.h>
#include <openssl/evp.h>
#include <string.h>

typedef struct prov_ctx_st {
    const OSSL_CORE_HANDLE *handle;
    OSSL_FUNC_core_new_error_fn      *core_new_error;
    OSSL_FUNC_core_set_error_debug_fn *core_set_error_debug;
    OSSL_FUNC_core_vset_error_fn      *core_vset_error;
    OSSL_LIB_CTX *delegate_libctx;    /* private; see 7.4 */
    EVP_MD      *delegate_md;
} PROV_CTX;

/* 7.2: the incoming table is walked, never indexed. */
static PROV_CTX *prov_ctx_new(const OSSL_CORE_HANDLE *handle,
                              const OSSL_DISPATCH *in)
{
    PROV_CTX *ctx = OPENSSL_zalloc(sizeof(*ctx));

    if (ctx == NULL)
        return NULL;
    ctx->handle = handle;

    for (; in->function_id != 0; in++) {
        switch (in->function_id) {
            case OSSL_FUNC_CORE_NEW_ERROR:
                ctx->core_new_error = OSSL_FUNC_core_new_error(in);
                break;
            case OSSL_FUNC_CORE_SET_ERROR_DEBUG:

```

```

        ctx->core_set_error_debug = OSSL_FUNC_core_set_error_debug(in);
        break;
    case OSSL_FUNC_CORE_VSET_ERROR:
        ctx->core_vset_error = OSSL_FUNC_core_vset_error(in);
        break;
    default:
        break; /* unknown ids are ignored, not errors */
}
}

/* 7.4: delegation uses a private context, never the caller's. */
if ((ctx->delegate_libctx = OSSL_LIB_CTX_new()) == NULL)
    goto err;
ctx->delegate_md = EVP_MD_fetch(ctx->delegate_libctx, "SHA2-256",
                                "provider=default");
if (ctx->delegate_md == NULL)
    goto err;
return ctx;

err:
    prov_ctx_free(ctx);
    return NULL;
}

static void tlsext_teardown(void *provctx)
{
    prov_ctx_free((PROV_CTX *)provctx); /* N4: safe on NULL */
}

/* 7.3 */
static const OSSL_ALGORITHM *tlsext_query(void *provctx, int operation_id,
                                           int *no_cache)
{
    *no_cache = 0;
    switch (operation_id) {
    case OSSL_OP_DIGEST:    return tlsext_digests;
    case OSSL_OP_CIPHER:    return tlsext_ciphers;
    case OSSL_OP_KEYMGMT:   return tlsext_keymgmt;
    case OSSL_OP_KEYEXCH:   return tlsext_keyexch;
    case OSSL_OP_SIGNATURE: return tlsext_signatures;
    }
    return NULL;
}

/* 10.5: advertisement is a separate obligation from implementation. */
static int tlsext_get_capabilities(void *provctx, const char *capability,
                                   OSSL_CALLBACK *cb, void *arg)
{
    if (OPENSSL_strcasecmp(capability, "TLS-GROUP") == 0)
        return tlsext_group_capability(cb, arg);
    if (OPENSSL_strcasecmp(capability, "TLS-SIGALG") == 0)
        return tlsext_sigalg_capability(cb, arg);
}

```

```

    return 0;                                /* capability not supported */
}

static const OSSL_DISPATCH provider_dispatch[] = {
    { OSSL_FUNC_PROVIDER_TEARDOWN,          (void (*)(void))tlsext_teardown },
    { OSSL_FUNC_PROVIDER_QUERY_OPERATION,    (void (*)(void))tlsext_query },
    { OSSL_FUNC_PROVIDER_GET_CAPABILITIES,   (void (*)(void))tlsext_get_capabilities },
    { OSSL_FUNC_PROVIDER_GETTABLE_PARAMS,   (void (*)(void))tlsext_gettable },
    { OSSL_FUNC_PROVIDER_GET_PARAMS,        (void (*)(void))tlsext_get_params },
    { 0, NULL }
};

int OSSL_provider_init(const OSSL_CORE_HANDLE *handle,
                      const OSSL_DISPATCH *in,
                      const OSSL_DISPATCH **out,
                      void **provctx)
{
    PROV_CTX *ctx;

    if ((ctx = prov_ctx_new(handle, in)) == NULL)
        return 0;
    *provctx = ctx;
    *out = provider_dispatch;
    return 1;
}

```

A.2 Algorithm tables

```

/* 9.5: distinctive names; the component is inert until asked for by name. */
static const OSSL_ALGORITHM tlsext_digests[] = {
    { "TLSEXT-HASH256", "provider=tlsext",
      tlsext_digest_functions, "Experimental TLS handshake hash" },
    { NULL, NULL, NULL, NULL }
};

static const OSSL_ALGORITHM tlsext_ciphers[] = {
    { "TLSEXT-AEAD", "provider=tlsext",
      tlsext_aead_functions, "Experimental AEAD for TLS record protection" },
    { NULL, NULL, NULL, NULL }
};

static const OSSL_ALGORITHM tlsext_keymgmt[] = {
    { "TLSEXT-P256", "provider=tlsext",
      tlsext_ec_keymgmt_functions, "Experimental EC group key management" },
    { NULL, NULL, NULL, NULL }
};

static const OSSL_ALGORITHM tlsext_keyexch[] = {

```

```

    { "TLSEXT-P256", "provider=tlsext",
      tlsext_ec_keyexch_functions, "Experimental EC key agreement" },
    { NULL, NULL, NULL, NULL }
};

```

A.3 Digest component, showing the duplication contract

```

/*
 * 9.2: duplication must be deep. Copying the delegate handle instead of
 * duplicating the delegate context is the shallow-copy error that appears
 * to work until one of the two contexts is updated.
 */
typedef struct digest_ctx_st {
    PROV_CTX      *provctx;
    EVP_MD_CTX    *delegate;      /* owned */
} DIGEST_CTX;

static void *digest_dupctx(void *vctx)
{
    DIGEST_CTX *src = vctx, *dst;

    if (src == NULL)
        return NULL;
    if ((dst = OPENSSL_zalloc(sizeof(*dst))) == NULL)
        return NULL;
    dst->provctx = src->provctx;

    if ((dst->delegate = EVP_MD_CTX_new()) == NULL)
        goto err;
    /* duplicate the STATE, not the handle */
    if (!EVP_MD_CTX_copy_ex(dst->delegate, src->delegate))
        goto err;
    return dst;

err:
    digest_freectx(dst);
    return NULL;
}

/* 9.3: the key schedule sizes itself from these answers. */
static const OSSL_PARAM *digest_gettable(void *provctx)
{
    static const OSSL_PARAM table[] = {
        OSSL_PARAM_size_t(OSSL_DIGEST_PARAM_BLOCK_SIZE, NULL),
        OSSL_PARAM_size_t(OSSL_DIGEST_PARAM_SIZE, NULL),
        OSSL_PARAM_END
    };
    return table;
}

```

```

}

static int digest_get_params(OSSL_PARAM params[])
{
    OSSL_PARAM *p;

    if ((p = OSSL_PARAM_locate(params, OSSL_DIGEST_PARAM_SIZE)) != NULL
        && !OSSL_PARAM_set_size_t(p, TLSEXT_DIGEST_SIZE))
        return 0;
    if ((p = OSSL_PARAM_locate(params, OSSL_DIGEST_PARAM_BLOCK_SIZE)) != NULL
        && !OSSL_PARAM_set_size_t(p, TLSEXT_DIGEST_BLOCK))
        return 0;
    return 1;
}

```

A.4 Group capability entry

```

/* 10.5 and 4.3.1. The code point comes from a registry or the
 * private-use range -- never from neither (10.6). */
static unsigned int group_id = TLSEXT_GROUP_CODE_POINT;
static unsigned int secbits = 128;
static int min_tls = TLS1_3_VERSION;
static int max_tls = TLS1_3_VERSION;

static int tlsext_group_capability(OSSL_CALLBACK *cb, void *arg)
{
    static const OSSL_PARAM group[] = {
        OSSL_PARAM_utf8_string(OSSL_CAPABILITY_TLS_GROUP_NAME,
                               "tlsext256", sizeof("tlsext256")),
        OSSL_PARAM_utf8_string(OSSL_CAPABILITY_TLS_GROUP_NAME_INTERNAL,
                               "TLSEXT-P256", sizeof("TLSEXT-P256")),
        OSSL_PARAM_utf8_string(OSSL_CAPABILITY_TLS_GROUP_ALG,
                               "TLSEXT-P256", sizeof("TLSEXT-P256")),
        OSSL_PARAM_uint(OSSL_CAPABILITY_TLS_GROUP_ID, &group_id),
        OSSL_PARAM_uint(OSSL_CAPABILITY_TLS_GROUP_SECURITY_BITS, &secbits),
        OSSL_PARAM_int(OSSL_CAPABILITY_TLS_GROUP_MIN_TLS, &min_tls),
        OSSL_PARAM_int(OSSL_CAPABILITY_TLS_GROUP_MAX_TLS, &max_tls),
        OSSL_PARAM_END
    };
    return cb(group, arg);
}

```

A.5 Peer share validation

```

/*
 * 10.4 and 17.4.1. This is the design's most security-critical function.
 * Validation happens BEFORE any operation involving the private key.
 */

```

```
static int ecdh_set_peer(void *vctx, void *vpeer)
{
    KEYEXCH_CTX *ctx = vctx;
    EC_KEY *peer = vpeer;

    if (ctx == NULL || peer == NULL)
        return 0;

    /* 1. the encoding decoded to a point at all */
    if (!tlsext_point_decoded(peer))
        return 0;
    /* 2. the point satisfies the curve equation */
    if (!tlsext_point_on_curve(peer))
        return 0;
    /* 3. the point is not the identity */
    if (tlsext_point_is_infinity(peer))
        return 0;
    /* 4. small-subgroup check, where the cofactor permits one */
    if (!tlsext_point_order_ok(peer))
        return 0;

    ctx->peer = peer;
    return 1;
}
```

Appendix B. Configuration and Operational Reference

B.1 Full configuration file

The configuration of §13.1, with every section annotated. Paths are examples.

```
# openssl.cnf -- activate the tlsext provider and select its algorithms.
openssl_conf = openssl_init

[openssl_init]
providers = provider_sect
ssl_conf  = ssl_sect

# --- Provider activation (13.1) -----
[provider_sect]
default = default_sect      # 3.7: omitting this leaves the process
tlsext  = tlsext_sect      #      without standard algorithms

[default_sect]
activate = 1

[tlsext_sect]
module   = /usr/local/lib/openssl-modules/tlsext.so
activate = 1

# --- Algorithm selection (13.2) -----
# Activation makes algorithms AVAILABLE; these lines make them CHOSEN.
[ssl_sect]
system_default = system_default_sect

[system_default_sect]
Groups          = tlsext256:x25519:secp256r1
CipherSuites    = TLS_TLSEXT_AEAD_TLSEXT_HASH256:TLS_AES_256_GCM_SHA384
```

B.2 Fail-closed variant

For a deployment whose requirement is mandatory rather than preferred (§6.4, §14.2). Note that rollback from this configuration requires restoring the algorithm lines, not merely deactivating the provider (§13.6).

```
[system_default_sect]
Groups          = tlsext256
CipherSuites    = TLS_TLSEXT_AEAD_TLSEXT_HASH256
```

B.3 Diagnostic sequence

The order of §14.5. Each command answers one question; taking them in order is faster than reasoning backwards from a failed handshake.

```
# 1. Is the provider loaded?
openssl list -providers

# 2. Does it advertise the algorithm, and under what name?
openssl list -digest-algorithms -provider tlsext
openssl list -cipher-algorithms -provider tlsext
openssl list -key-exchange-algorithms -provider tlsext

# 3. Does the property query admit it?
openssl list -digest-algorithms -propquery 'provider=tlsext'

# 4. Which groups and suites are offered?
openssl ciphers -v -tls1_3
openssl s_client -connect host:443 -tls1_3 -groups tlsext256 </dev/null

# 5. What was actually negotiated? (14.2 -- established is not sufficient)
openssl s_client -connect host:443 -tls1_3 </dev/null \
    | grep -E 'Cipher|Server Temp Key|Negotiated'
```

Step 5 is the one that is usually skipped. A successful connection proves a handshake completed, not that it used the intended algorithms. Confirming the negotiated suite and group is the difference between testing the deployment and testing TLS.

B.4 Failure-to-symptom reference

Symptom	Likely cause	Section
Provider absent from list - providers	Wrong module path, or activate not set	13.1
Provider listed, algorithms absent	Query function not answering that operation id	7.3

Symptom	Likely cause	Section
Algorithms listed, never negotiated	Missing capability entry, or not named in Groups/CipherSuites	10.1, 13.2
Works in the tool, not in the application	Different library context	13.3
Standard algorithms stop working	default not activated, or a broad property default	3.7, 14.4
handshake_failure at ServerHello	No shared suite or group; peer lacks the code point	12.3
Failure after ServerHello	Suite selected, algorithm not fetchable	12.3
Failure at Finished	Transcript divergence, duplication bug, or schedule width	9.2, 15.3
Record-length errors	Missing AEAD algorithm parameters	8.3
HKDF wrong from the first step	blocksize not reported	9.3, 15.3
Intermittent resumption failures	Session cache not recording the hash	15.5
Connection succeeds, requirement unmet	Property expressed as preferred, not required	14.2

The last row has no error message, which is why it is last and why Chapter 19 makes its negative test mandatory.

B.5 Build sketch

```
CFLAGS = -O2 -Wall -Wextra -fPIC $(shell pkg-config --cflags libcrypto)
LDLAGS = -shared $(shell pkg-config --libs libcrypto)

tlsext.so: provider.o ctx.o cipher_aead.o digest.o keymgmt_ec.o \
    keyexch_ec.o signature.o capabilities.o params.o
    $(CC) $(LDLAGS) -o $@ $^

install: tlsext.so
    install -m 0755 tlsext.so /usr/local/lib/openssl-modules/
```

The module links only against `libcrypto` (requirement N3). It does not link against `libssl`, and the absence is structural rather than incidental: a provider supplies algorithms to `libcrypto`, and the TLS library consumes them from there (Chapter 5).

Appendix C. Parameter Reference

Requirement F8 obliges every algorithm to report its parameters, and §5.5 stated the rule that everything the protocol must know crosses the boundary as a parameter. This appendix collects, per component, what must be answered and what consumes it. A parameter that is not answered does not produce an error at the provider; it produces a failure at the consumer, often far away (§8.3, §15.3).

C.1 Digest

Parameter	Type	Answered by	Consumed by	If wrong
size	size_t	Algorithm	Key schedule; secret, Finished and transcript widths	Truncated or over-read secrets (15.3)
blocksize	size_t	Algorithm	HMAC inside HKDF	Every HKDF step wrong, immediately

C.2 AEAD cipher

Parameter	Type	Answered by	Consumed by	If wrong
keylen	size_t	Algorithm	Traffic key derivation	Wrong key length; handshake fails after keys are installed
ivlen	size_t	Algorithm	Static IV derivation; nonce construction	Nonce misconstruction — catastrophic (8.4)
taglen	size_t	Algorithm	Record sizing	Record-length errors (8.3)
blocksize	size_t	Algorithm	Buffer arithmetic	Buffer sizing errors
mode	uint	Algorithm	Consumer's mode dispatch	Treated as non-AEAD

Parameter	Type	Answered by	Consumed by	If wrong
aead	int	Algorithm	AEAD capability check	Rejected for TLS 1.3 use
tag	octets	Context	Retrieved after encrypt; supplied before decrypt	Integrity failure or false accept (8.6)
ivlen (ctx)	size_t	Context	Per-operation IV width	As above
AAD	octets	Context	Record header binding	Header not authenticated

C.3 Key management and key exchange

Item	Direction	Consumed by	Note
Public key export encoding	Provider to core	key_share construction	This encoding is the wire encoding (10.2)
Public key import	Core to provider	Peer share reconstruction	Validation happens here (10.4)
Derive output length	Provider to core	Shared secret buffer	Must be the fixed field size, not the computed length (10.3)
Group name / internal name / algorithm	Capability	Negotiation	Wire name decoupled from implementation name (4.3.1)
Group id	Capability	supported_groups	From a registry or private-use range (10.6)
Security bits	Capability	Security-level filtering	Understated filters it out; overstated defeats a control (11.3)
Min / max TLS	Capability	Version gating	Confine to TLS 1.3 (16.3)

C.4 Signature

Item	Direction	Consumed by	Note
IANA name	Capability	Configuration and protocol identity	Two-layer naming (11.3)
Algorithm name	Capability	Fetch	
OID	Capability	Certificate matching	Brings encoders/decoders into scope (11.4)
Code point	Capability	signature_algorithms	As for groups
Security bits	Capability	Policy ranking	As above
Signature max length	Two-call convention	Caller allocation	Must be the maximum, not the typical

C.5 Checklist

A component is parameter-complete when, for every row above that applies to it, the parameter appears in the gettable list *and* is answered by the get function. The two are separate obligations: a parameter advertised but not answered, or answered but not advertised, is a defect that some consumers tolerate and others do not, which produces the worst kind of bug — one that depends on the consumer.

Appendix D. Status of Claims

Chapter 1.5 promised that every load-bearing claim would be collected with its status. This appendix is that table. A reader who wants to know what this paper establishes, as opposed to what it proposes, should read this page.

Statuses are as defined in §1.5: **Verified** — checked against OpenSSL source on the reference machine, reference given; **Design** — a specification of the proposed system, not built; **Reported** — attributed to the author of the system described, not independently reproduced.

D.1 Claims about OpenSSL

#	Claim	Status	Reference
1	A TLS 1.3 suite binds an AEAD algorithm and a handshake hash in one code point	Verified	ssl/s3_lib.c:39–56
2	The TLS 1.3 suite list is a static table whose length is taken at compile time	Verified	ssl/s3_lib.c:26, :39
3	Providers answer the TLS-GROUP and TLS-SIGALG capabilities	Verified	providers/common/capabilities.c:335–347
4	ossl_prov_get_capabilities() returns 0 for any other capability name	Verified	providers/common/capabilities.c:344–346
5	A group advertisement carries wire name, internal name, algorithm, id and security metadata	Verified	providers/common/capabilities.c:96–108
6	Several named curves may map to one implementing algorithm	Verified	providers/common/capabilities.c:161–164
7	A signature advertisement carries IANA name, algorithm name, OID, code point and security bits	Verified	providers/common/capabilities.c:292–302

#	Claim	Status	Reference
8	OpenSSL introduces ML-DSA signature algorithms to TLS through the capability mechanism	Verified	providers/common/capabilities.c:316–318
9	Fetching by name is much slower than direct method-table access; prefetching is recommended	Verified	references/12-openssl-guide-migration.pod:223–228
10	The ENGINE API is deprecated in 3.x; providers are its replacement	Verified	papers/openssl-3.5.5/03-ENGINE_add.pod:169; 09-openssl-engine.pod.in:25
11	A property query selects implementations by key/value assertions	Verified	papers/openssl-3.5.5/07-openssl-glossary.pod:173–190
12	A library context is a scope within which configuration applies	Verified	papers/openssl-3.5.5/07-openssl-glossary.pod:110–116
13	Implicit fetching resolves an implementation on first use with default criteria	Verified	papers/openssl-3.5.5/07-openssl-glossary.pod:95–101

D.2 Claims about the proposed design

#	Claim	Status	Chapter
14	The provider structure of §7.1–7.7 satisfies F8, F9, N3–N6	Design	7
15	An AEAD component answering key, IV and tag lengths and accepting an external nonce satisfies F1 and C2	Design	8
16	A hash component with deep duplication and accurate size reporting satisfies F2	Design	9
17	A group component with validation and a capability entry satisfies F3 and F4	Design	10
18	Registration requirements 1–5 are necessary for a usable suite	Design	12.2

#	Claim	Status	Chapter
19	The three negotiation failure modes of §12.3 are distinguishable	Design	12.3
20	A required property produces a clean fetch failure rather than fallback (F10)	Design	14.2
21	Changing the handshake hash changes the width of every secret in the schedule	Design, following from claim 1 and RFC 8446	15.2
22	An overstated digest length can yield a working connection with weakened keys	Design (analysis)	15.3
23	The design cannot introduce a downgrade, because a provider cannot alter sequencing	Design, following from C1	17.6

D.3 Reported capability

#	Claim	Status	Would be established by
24	The author's system adds cipher suites combining block cipher, elliptic-curve and hash algorithms through the provider interface	Reported	§19.5 L4, with §19.5.1 negative tests
25	Such suites are usable in TLS communication	Reported	§19.5 L4, confirming the <i>negotiated</i> suite per §B.3 step 5

On claims 24 and 25. This study had no access to the implementation and did not reproduce them. They are recorded here as attributed statements so that a reader can see precisely which parts of the paper rest on them — Chapter 12's framing, and nothing in Chapters 2–11 or 13–19, all of which stand on the verified and design claims above. A reader evaluating the work should ask for the Chapter 19 evidence, particularly the `openssl` ciphers comparison with the provider activated and deactivated, which settles the question directly.

D.4 What was not done

Per §19.7, absence is reported rather than left to inference:

- No provider was compiled.
- No handshake was performed.
- No measurement was taken (Chapter 18 is a methodology).
- No interoperability testing was done, and none is claimed (§16.5).
- Constant-time behaviour was not verified on any artefact (§17.5).
- No FIPS validation is claimed or approached (§17.7).

D.5 Requirement coverage

Requirement	Addressed	Test that would establish it
F1, C2	Ch. 8	L1 AEAD vectors; §19.5.1 record corruption
F2, C3	Ch. 9, 15	L1 boundary lengths; L2 duplication; L3 schedule
F3, F4	Ch. 10	L1 key exchange; §19.5.1 invalid-curve and small-subgroup
F5, F10	Ch. 14	§19.5.1 provider deactivation
F6, C6	Ch. 13	§19.5.1 wrong library context
F7, C4	Ch. 12, 16	L4 handshake; L5 interoperability
F8	Ch. 7, App. C	L2 parameter completeness
F9	Ch. 7	L2 coexistence
N1	Ch. 18	M5 prefetch benefit
N2, C5	Ch. 17	Binary-level verification (not proposed in detail)
N3–N6	Ch. 7	L2 teardown under a leak checker

Appendix E. Glossary

Definitions of OpenSSL terms follow the project's own glossary where one exists; those entries are marked (*OpenSSL*). Protocol terms follow RFC 8446.

AEAD

Authenticated Encryption with Associated Data. The only form of record protection in TLS 1.3. Provides confidentiality for the plaintext and integrity for both the plaintext and additional data that is transmitted in the clear.

Capability

A named set of declarations a provider makes to the core, answered by invoking a callback once per item with an `OSSL_PARAM` array. `TLS-GROUP` and `TLS-SIGALG` are the two the TLS layer consumes (§4.3).

Cipher suite

In TLS 1.3, a code point naming an AEAD algorithm and a handshake hash. Unlike TLS 1.2 suites, it does not encode key exchange or authentication.

Code point

A numeric identifier carried on the wire, drawn from an IANA registry. Groups, signature algorithms and cipher suites each have their own registry.

Dispatch table

An array of `OSSL_DISPATCH` entries pairing function identifiers with function pointers, terminated by a zero entry. The provider architecture's ABI (§3.2).

ENGINE

The pre-3.0 extension mechanism: a container for method structures indexed by NID. Deprecated in OpenSSL 3.0 and replaced by providers (claim 10, Appendix D).

Explicit fetching

(*OpenSSL*) Obtaining an algorithm object by an explicit call such as `EVP_MD_fetch()`. The returned object is owned by the caller.

Fetching

(*OpenSSL*) Resolving an algorithm name and property query to an implementation supplied by some activated provider.

Handshake hash

The hash function named by the cipher suite, used for the transcript hash and, through HMAC and HKDF, for the entire key schedule. Its output length parameterises the width of every secret (Chapter 15).

HKDF

HMAC-based key derivation function, structured as Extract and Expand. The basis of the TLS 1.3 key schedule.

Implicit fetching

(*OpenSSL*) Use of an algorithm object with no associated implementation, such as the return value of `EVP_sha256()`, with an implementation resolved automatically on first use using default criteria.

Invalid-curve attack

An attack in which a peer supplies a point that is not on the intended curve, causing operations with the victim's private key in a weaker group and leaking key material over successive handshakes. Prevented by the validation of §10.4.

Key schedule

The sequence of HKDF operations deriving every secret in a TLS 1.3 connection from the shared secret and the transcript.

Key share

The extension carrying a public key-exchange value, encoded as an opaque octet string whose interpretation is defined per group.

Library context

(*OpenSSL*) `OSSL_LIB_CTX`: a scope within which configuration applies and providers are activated. Represented by `NULL` for the default context.

OSSL_ALGORITHM

The structure by which a provider advertises one implementation: a colon-separated name list, a property definition, a dispatch table and a description (§3.3).

OSSL_PARAM

The key/typed-value structure by which data crosses the core/provider boundary (§3.4).

Property / property query

(*OpenSSL*) A key/value pair classifying an implementation, and a string of such assertions used to select among implementations. Required and preferred forms differ by one character and by a great deal of meaning (§14.1).

Provider

(*OpenSSL*) A component grouping together algorithm implementations, from OpenSSL itself or from a third party.

Small-subgroup attack

An attack supplying a point of small order, so that the resulting shared secret takes few possible values and reveals the private key modulo the subgroup order.

Transcript hash

A hash over all handshake messages so far, consumed at several points without being finalised — hence the requirement for context duplication (§9.2).

E.1 Abbreviations

Abbreviation	Expansion
AAD	Additional Authenticated Data
ABI	Application Binary Interface
AEAD	Authenticated Encryption with Associated Data
CMVP	Cryptographic Module Validation Program
ECDH	Elliptic-Curve Diffie-Hellman
GCM	Galois/Counter Mode
HKDF	HMAC-based Key Derivation Function
HMAC	Keyed-Hash Message Authentication Code
IANA	Internet Assigned Numbers Authority
KEM	Key Encapsulation Mechanism
ML-DSA	Module-Lattice-based Digital Signature Algorithm
NID	Numeric Identifier (OpenSSL's internal algorithm numbering)
OID	Object Identifier
TEE	Trusted Execution Environment

Abbreviation	Expansion
TLS	Transport Layer Security

References

OpenSSL manual snapshots are pinned to the source tags shown. Documents marked `papers/` or `references/` are stored beside this paper in the project tree, with SHA-256 hashes recorded in the corresponding manifest.

1. OpenSSL Project. *crypto.pod*. OpenSSL_1_1_1g. [Online source](#). Local file: `openssl-1.1.1g/01-crypto.pod`.
2. OpenSSL Project. *ENGINE_add.pod*. OpenSSL_1_1_1g. [Online source](#). Local file: `openssl-1.1.1g/02-ENGINE_add.pod`.
3. OpenSSL Project. *EVP_DigestInit.pod*. OpenSSL_1_1_1g. [Online source](#). Local file: `openssl-1.1.1g/03-EVP_DigestInit.pod`.
4. OpenSSL Project. *EVP_EncryptInit.pod*. OpenSSL_1_1_1g. [Online source](#). Local file: `openssl-1.1.1g/04-EVP_EncryptInit.pod`.
5. OpenSSL Project. *config.pod*. OpenSSL_1_1_1g. [Online source](#). Local file: `openssl-1.1.1g/05-config.pod`.
6. OpenSSL Project. *OPENSSL_init_crypto.pod*. OpenSSL_1_1_1g. [Online source](#). Local file: `openssl-1.1.1g/06-OPENSSL_init_crypto.pod`.
7. OpenSSL Project. *README*. OpenSSL_1_1_1g. [Online source](#). Local file: `openssl-1.1.1g/07-README`.
8. OpenSSL Project. *ossl-guide-libcrypto-introduction.pod*. openssl-3.5.5. [Online source](#). Local file: `openssl-3.5.5/01-ossl-guide-libcrypto-introduction.pod`.
9. OpenSSL Project. *ossl-guide-libraries-introduction.pod*. openssl-3.5.5. [Online source](#). Local file: `openssl-3.5.5/02-ossl-guide-libraries-introduction.pod`.
10. OpenSSL Project. *ENGINE_add.pod*. openssl-3.5.5. [Online source](#). Local file: `openssl-3.5.5/03-ENGINE_add.pod`.
11. OpenSSL Project. *provider-keymgmt.pod*. openssl-3.5.5. [Online source](#). Local file: `openssl-3.5.5/04-provider-keymgmt.pod`.
12. OpenSSL Project. *provider-signature.pod*. openssl-3.5.5. [Online source](#). Local file: `openssl-3.5.5/05-provider-signature.pod`.
13. OpenSSL Project. *openssl-core.h.pod*. openssl-3.5.5. [Online source](#). Local file: `openssl-3.5.5/06-openssl-core.h.pod`.
14. OpenSSL Project. *openssl-glossary.pod*. openssl-3.5.5. [Online source](#). Local file: `openssl-3.5.5/07-openssl-glossary.pod`.
15. OpenSSL Project. *openssl-list.pod.in*. openssl-3.5.5. [Online source](#). Local file: `openssl-3.5.5/08-openssl-list.pod.in`.
16. OpenSSL Project. *openssl-engine.pod.in*. openssl-3.5.5. [Online source](#). Local file: `openssl-3.5.5/09-openssl-engine.pod.in`.

17. Billy Bob Brumley. *SoK: The Constant Time Model*. 2606.13000v1. [Online source](#). Local file: `arxiv/2606.13000-sok-the-constant-time-model.pdf`.
18. Ananya Kudaloor, Adnan Aijaz. *Toward Quantum-Safe 6G: Experimental Evaluation of Post-Quantum Cryptography Techniques*. 2605.06881v1. [Online source](#). Local file: `arxiv/2605.06881-toward-quantum-safe-6g-experimental-evaluation-of-post-quantum-cryptog.pdf`.
19. José Luis Delgado Jiménez. *Signature Placement in Post-Quantum TLS Certificate Hierarchies: An Experimental Study of ML-DSA and SLH-DSA in TLS 1.3 Authentication*. 2604.06100v3. [Online source](#). Local file: `arxiv/2604.06100-signature-placement-in-post-quantum-tls-certificate-hierarchies-an-exp.pdf`.
20. Javier Blanco-Romero, Yuri Melissa Garcia-Niño, Florina Almenares Mendoza et al.. *Post-Quantum Entropy as a Service for Embedded Systems*. 2603.10274v1. [Online source](#). Local file: `arxiv/2603.10274-post-quantum-entropy-as-a-service-for-embedded-systems.pdf`.
21. Juyoul Lee, Sanzida Hoque, Abdullah Aydeger et al.. *Quantum-Resistant Domain Name System: A Comprehensive System-Level Study*. 2506.19943v1. [Online source](#). Local file: `arxiv/2506.19943-quantum-resistant-domain-name-system-a-comprehensive-system-level-stud.pdf`.
22. Javier Blanco-Romero, Pedro Otero García, Daniel Sobral-Blanco et al.. *QKD-KEM: Hybrid QKD Integration into TLS with OpenSSL Providers*. 2503.07196v1. [Online source](#). Local file: `arxiv/2503.07196-qkd-kem-hybrid-qkd-integration-into-tls-with-openssl-providers.pdf`.
23. Christian N  ther, Daniel Herzinger, Jan-Philipp Stegh  fer et al.. *Toward a Common Understanding of Cryptographic Agility -- A Systematic Review*. 2411.08781v3. [Online source](#). Local file: `arxiv/2411.08781-toward-a-common-understanding-of-cryptographic-agility-a-systematic-re.pdf`.
24. Jihoon Cho, Changhoon Lee, Eunkyung Kim et al.. *Software-Defined Cryptography: A Design Feature of Cryptographic Agility*. 2404.01808v2. [Online source](#). Local file: `arxiv/2404.01808-software-defined-cryptography-a-design-feature-of-cryptographic-agilit.pdf`.
25. Alessio Claudio, Andrea Vesco. *A Novel DID Method Leveraging the IOTA Tangle and its Integration into OpenSSL*. 2310.01087v2. [Online source](#). Local file: `arxiv/2310.01087-a-novel-did-method-leveraging-the-iota-tangle-and-its-integration-into.pdf`.
26. Daniel J. Bernstein, Billy Bob Brumley, Ming-Shing Chen et al.. *OpenSSLNTRU: Faster post-quantum TLS key exchange*. 2106.08759v3. [Online source](#). Local file: `arxiv/2106.08759-opensslntru-faster-post-quantum-tls-key-exchange.pdf`.
27. James Walden. *The Impact of a Major Security Event on an Open Source Project: The Case of OpenSSL*. 2005.14242v1. [Online source](#). Local file: `arxiv/2005.14242-the-impact-of-a-major-security-event-on-an-open-source-project-the-cas.pdf`.
28. OpenSSL Project. *provider*. openssl-3.5.5. [Online source](#). Local file: `01-provider.pod`.
29. OpenSSL Project. *provider-base*. openssl-3.5.5. [Online source](#). Local file: `02-provider-base.pod`.

30. OpenSSL Project. *provider-digest*. openssl-3.5.5. [Online source](#). Local file: 03-provider-digest.pod.
31. OpenSSL Project. *config*. openssl-3.5.5. [Online source](#). Local file: 04-config.pod.
32. OpenSSL Project. *OSSL_PROVIDER*. openssl-3.5.5. [Online source](#). Local file: 05-OSSL_PROVIDER.pod.
33. OpenSSL Project. *EVP_DigestInit / EVP_MD_fetch*. openssl-3.5.5. [Online source](#). Local file: 06-EVP_DigestInit.pod.
34. OpenSSL Project. *property*. openssl-3.5.5. [Online source](#). Local file: 07-property.pod.
35. OpenSSL Project. *OSSL_LIB_CTX*. openssl-3.5.5. [Online source](#). Local file: 08-OSSL_LIB_CTX.pod.
36. OpenSSL Project. *openssl-core_dispatch.h*. openssl-3.5.5. [Online source](#). Local file: 09-openssl-core_dispatch.h.pod.
37. OpenSSL Project. *OSSL_PARAM*. openssl-3.5.5. [Online source](#). Local file: 10-OSSL_PARAM.pod.
38. OpenSSL Project. *fips_module*. openssl-3.5.5. [Online source](#). Local file: 11-fips_module.pod.
39. OpenSSL Project. *ossl-guide-migration*. openssl-3.5.5. [Online source](#). Local file: 12-ossl-guide-migration.pod.
40. NIST. *Secure Hash Standard (FIPS PUB 180-4)*. August 2015. [Online source](#). Local file: 13-NIST-FIPS-180-4.pdf.
41. Max Ammann, Fredrik Dahlgren, Spencer Michaels, and Jim Miller, Trail of Bits. *OpenSSL Security Assessment*. 18 April 2024. [Online source](#). Local file: 14-OpenSSL-security-audit-2024.pdf.