SSP21

02/05/20 (r347)

CES-21 TLP WHITE

ORIG: SDG&E / Automatak

Contents

1 Version Information

SSP21 specifies both a `major` and `minor` version number on the wire so that implementations can unambiguously determine compatibility. Changes to the major version indicate that the protocol is no longer wire-compatible with older versions. Changes to the minor version indicate the the protocol was extended in some manner, e.g. additional cryptographic primitives were added to extensible enumerations. Implementations with the same major version and different minor versions will inter-operate so long as only optional modes defined in the earlier minor version are used.

Changes to this document that do not increment either the major or minor version do not affect wire-level compatibility in any way. They are grammatical or meant to clarify the standard.

The version specified in this document has major version 0 and minor version 1, i.e. `0.1`.

2 Introduction

Secure SCADA Protocol (SSP) is cryptographic wrapper designed to secure point-to-multipoint serial protocols, or to act as a security layer for new SCADA applications. It is intended to fill a gap where existing technologies like TLS are not applicable, or require too much processing power or bandwidth. It can be used as a protocol agnostic bump in the wire (BitW) at initiator endpoints or as a bump in the stack (BitS) on the master or the outstation. No provision is made for retrofitting masters with a BitW as we assume that masters can be much more easily upgraded than outstations.

3 Requirements

The following requirements guided the design of the specification and the selection of appropriate companion standards.

3.1 Simplicity of implementation

The encodings, state machines, and other technical details of the protocol shall, above all else but without sacrificing security, endeavor to be as simple to implement as possible. Complexity, bells and whistles, and unneeded features inevitably lead to bugs both in specification and implementation.

A reference implementation shall be developed to inform the evolving design, and shall not be an afterthought. Too often standardization efforts spend too much time on paper, only to lead to designs that are difficult to implement correctly.

3.2 Use only strong vetted cryptography

SSP21 shall only use algorithms that have received prolonged and intense scrutiny from the crypto community. This does not mean that all algorithms need to be NIST approved. Algorithms that are simpler to implement and/or have variations with provably constant-time implementations should be preferred.

3.3 Flexible Basis of trust

SSP21 shall allow for trust to be anchored in multiple ways, including:

- Shared secrets, i.e. symmetric cryptography
- Shared public keys, i.e. key-server style key management based on asymmetric cryptography
- Public Key Infrastructure (PKI) wholly controlled by the asset owner, with an efficient certificate format.

The trade-offs associated with each of these domains is discussed in a later section.

3.4 Asymmetric Certificate Revocation

Master certificates (certificates that identify masters to outstations), will support a fast expiration scheme in addition to explicit revocation. This works well in an operational environment where the utility has a reliable and isolated IP network between an online authority and multiple master stations. An intermediate authority private key can be used to periodically renew master certificates. Using certificate revocation lists (CRLs) with outstations may be undesirable as outstations may not be able to reach them on a serial channel, and masters would have to push revocation notifications down to each endpoint and ensure that they arrive. Outstations would then have to persist these CRLs in non-volatile memory.

This scheme requires that outstations have access to an out-of-band time synchronization mechanism such as GPS, local NTP via GPS in a substation, or WWVB. Alternatively, over TCP networks, outstations could check an online CRL.

Outstation certificates (certificates that identify outstations to masters) will be longer lived, and will be revoked using an online CRL accessible to the masters in the system over a traditional TCP network.

3.5 Extensible only to the extent necessary

- Endpoints shall be able to identify the protocols version to each other during key exchange.
- Must be secure against protocol downgrade attacks via a mechanism that fully authenticates the handshake.
- The protocol shall use security-suite specifications to allow new algorithms to be used in future versions, or to provide more than one option for when algorithms, inevitably, are compromised.
- The number of initial security suites will be limited to one or two, and will only support authentication.

3.6 Authentication

All messages shall be authenticated. Each endpoint in the session shall be able to unambiguously determine that a session message comes from the other endpoint. The authentication mechanism shall automatically provide message integrity and protection from spoofing and MitM attacks.

This authentication will ensure that a particular SCADA master is talking to a particular outstation. In other words, it shall only secure the communication link and will not authenticate individual users or operators of the system. Role Based Access Control (RBAC) and auditing of users is best done at the platform level, and is outside the scope of SSP21. Individual SCADA vendors are free to use different technologies (such as Active Directory, RSA, LDAP, Kerberos, etc.) to manage users at the platform level.

Particular BitS implementations could potentially use some metadata in certificates to limit or constrain what is allowed during a particular communication session. How this metadata is used or configured to limit permissions for a particular protocol is outside the scope of SSP21.

3.7 Protection from replay

Both endpoints of the session shall be able to detect replayed session messages. Although the protocol needs to be secure from replay, it does not necessarily need to ensure that all messages are delivered. SCADA protocols such as DNP3 automatically handle retries at a higher level. The protocol will support two modes: one that strictly enforces packet order over (TCP) and a more tolerant mode that allows any new (non-replayed) packet to pass over serial or UDP.

3.8 Session message time-to-live (TTL)

Since SSP21 is designed to protect control protocols with particular timing constraints, undesirable behavior could occur if an attacker held back one or more authenticated control messages and then replayed them in rapid succession. To reduce the effectiveness of this mode of attack, both parties record their relative time base during session negotiation. Session messages then include a timestamp in milliseconds since the beginning of the session that indicates the last possible moment when the packet should be accepted.

Implementations will have to make these timing parameters configurable so that they can be tuned for the latency of particular networks. As relative clock drift can occur, sessions may need to be renegotiated more frequently or the configurable validity window of session messages increased appropriately.

3.9 Optional encryption

The secure operation of SCADA systems does not require confidentiality of session traffic under all, or even most, circumstances. Reasons to prefer unencrypted sessions include the ability to inspect traffic with IDS/IPS and denying a potentially opaque tunnel to an adversary.

Certain systems may exchange sensitive information and require session confidentiality. SSP21 shall use a security suite specification and encodings that allow for encrypted sessions in the future. The session key exchange mechanism shall support forward secrecy.

3.10 Perfect Forward Secrecy

The protocol shall provide a mode that provides perfect forward secrecy. Namely, if an adversary compromises the long-term private key of an endpoint, they shall not be able to decrypt past sessions.

This mode is only useful for encrypted session modes, and does not offer any benefit to authentication-only modes.

3.11 Support bump-in-the-wire retrofits

Outstation implementations of the protocol shall be capable of being deployed as a bump-in-the-wire (BitW) or integrated into endpoints as a bump-in-the-stack (BitS). BitS integration is preferred, but it is understood that BitW implementations are necessary to retrofit legacy components during transitions.

Requiring a BitW implementation only for outstations and not masters simplifies requirements as the BitW needn't be protocol-aware. It can be configured with the static addresses of the outstation and master, and ignore protocol messages addressed for other nodes. In BitW and BitS implementations, all cryptographic operations including key negotiation and authentication will occur at the bump.

3.12 Support serial and IP

Supporting multi-drop serial means that frames must be addressed in some manner. SSP21 will use 16-bit addressing as this accommodates the addressing scheme used for common existing SCADA protocols. SSP21 will have its own delimiters or length fields, and will use some type of non-cryptographic error detection so that environmental noise is probabilistically filtered out at a level below cryptographic checks for deliberate tampering.

For some protocols, this new secure serial layer could act as a replacement for redundant functionality in existing protocols. For example, the DNP3 link-layer and transport function could be completely removed in BitS implementations and replaced with the SSP21 crypto and framing layers. SSP21 could also fully wrap the existing protocols, but removing redundancy in certain implementations could provide significant bandwidth savings.

Out-of-band messages like session key establishment, heartbeats, etc. can only be initiated from the SCADA master side when it attempts to send a normal protocol message. This is because in half-duplex communications the wrapper cannot squelch a reply from a remote by inappropriately using the channel.

3.13 Low overhead

Security is not a zero-cost protocol feature. Inevitably adding a security sub-layer will require a few more bytes on the wire, increase latency, and put a computational burden on endpoints. SSP21 will endeavor to minimize these overheads and provide modes with varying requirements on hardware.

- **reduced latency** – BitS implementations have a significant advantage in this regard over BitW. HMAC hold back can double latencies in BitW integrations as the entire packet must be received and verified before the first payload byte can be emitted. Some tricks could be played with asymmetric baud rates to minimize this effect. MAC algorithms should be used for which hardware acceleration exists.

- **reduced bandwidth** – It is not uncommon for serial SCADA systems to operate at rates as low as 1200 bits per second. Cryptographic encodings need to be sensitive to tight polling margins. HMACs can be truncated (per NIST guidelines) to reduce overhead. BitS integration may be able to remove redundant layers provided by both the SSP21 and the wrapped protocol. An efficient certificate format that utilizes Elliptic Curve Cryptography (ECC) public keys will be used to reduce certificate sizes.

4 Handshake mode

While the primary aim of this specification is describe the protocol in sufficient detail that it can be faithfully implemented, it is important to describe the trade-offs for the various handshake modes that are supported in the protocol. Handshake modes may differ in either the way trust is anchored or in the security properties they provide. This non-normative section of the document describes the relative pros and cons of each mode.

SSP21 sends the same handshake messages, in the same order, regardless of the handshake mode specified. The messages have roughly the same meaning, but certain fields are interpreted in different ways depending on the mode. The handshake has two request-response phases that can be roughly summarized as follows:

- A single round-trip request/response (1-RTT) to perform key negotiation (phase 1 - key negotiation)
- Each party then transmits its first session data message to authenticate (phase 2 - authentication and optional payload)

Only phase 1 differs depending on the handshake mode. The authentication in phase 2 and the session itself are identical in all handshake modes. The modes are described informally in the following sections, mostly for the purposes of analyzing the benefits and short-comings of each mode.

4.1 Shared secrets

In this mode, each pair of parties that wishes to communicate must have a shared-secret that both parties possess prior to establishing a session. This secret may be installed manually, or distributed securely using emerging technologies like Quantum Key Distribution (QKD). Security is achieved in knowing that only the other party possesses the same key. This shared secret, along with a random nonce from each endpoint, is then used to establish a set of shared session keys.

This mode of operation uses symmetric cryptography only, and consequently has a number of advantages:

- The entire protocol can be implementing using only a secure hash function if confidentiality is not required.
- It can be implemented on deeply embedded systems that might not be powerful enough for asymmetric cryptography.
- It remains secure when a practical quantum computer is developed. The effectiveness of 256-bit shared-secrets will be reduced to 128-bits by Grover's algorithm. All other modes will be vulnerable until practical quantum-resistant public-key algorithms are available.

Despite these advantages, there are considerable challenges:

- The shared secret must leave the secure location where it is generated to be shared with the other party. This likely entrusts secrets to additional staff members or contractors in the absence of something like QKD.
- There is no support for perfect forward secrecy since there are no ephemeral keys exchanged. If a shared secret is ever disclosed, any saved traffic can be decrypted.
- If multiple masters are needed for redundancy purposes, the keys must be shared with each master increasing the attack surface and the risk of compromise, or the number of keys in the system must be doubled from N to 2N.
- Compromise of a field asset always requires that the channel be re-keyed. Full compromise of the master requires that the entire system be re-keyed.

4.2 Pre-shared public keys

In this architecture, each communication node has an asymmetric key pair. It is free to disseminate the public key, and each node must possess the public key for every other node with which it communicates. It might be pre-configured with these peer public keys, or it might retrieve them from a key server using a secure out-of-band mechanism. This architecture better addresses some of the concerns presented with the symmetric key only architecture, namely:

- Multiple masters can be commissioned without doubling the number of keys in the system, however, each outstation must possess the public key of each master with which it must communicate.
- Only the master's public key(s) need to be shared with commissioning personnel. Each outstation can also secure its private key, and only share the public key. This makes tampering from insiders slightly more difficult than in the symmetric only scheme.

A number of potential problems still remain:

- Compromise of a master still results in having to update the master's public key on each outstation.
- Installing or authorizing additional masters requires either sharing the master private key with the backup master, or installing an additional master public key on all outstations.

SSP21 is able to operate without an authority by using the pre-shared key mode.

4.3 Public keys authenticated from a root certificate

The recommended mode for managing trust is the PKI mode. In this mode, trust is anchored by a private authority controlled by the utility. Ideally, SCADA masters and field assets (RTUs, gateways, IEDs, etc.) generate a key pair locally, never share the private key with another entity (human or machine), and can freely disseminate the public key for the purposes of certificate generation. The primary role of any PKI is to reduce the complexity of key management by requiring parties to only place their trust in a central signing authority. The identity of all other parties is then established via digital signatures issued by this authority.

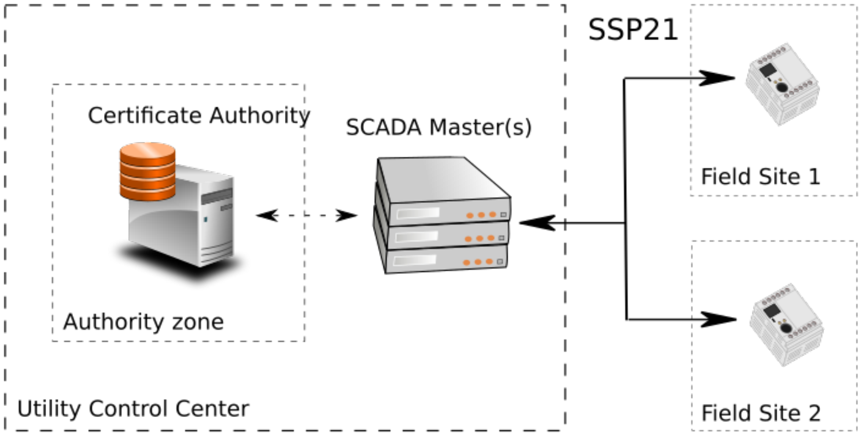

Figure 1: Components of the system relevant to SSP21

The challenges of running a PKI largely revolve around managing enrollment/revocation and keeping the authority secure. How this PKI is managed is outside the scope of SSP21, however, the following sub-sections discuss such a PKI notionally.

4.3.1 The role of the authority

The authority in the system possesses a private asymmetric key that it uses to sign certificates. Certificates consist of the following elements:

- A public asymmetric key
- Metadata associated with the public key (e.g. id, validity windows, serial numbers, etc)
- A digital signature over all other data calculated using the authority private key.

Creating and signing certificates is one of the primary roles of the authority. In its simplest form, this might consist of some cryptographic command line tools on a properly isolated server with a private key and a set of humans with access to this server. Such a basic system might work for small systems.

4.3.2 Issuing outstation certificates

There are far more outstations in any given SCADA system than the number of masters. Such a statement might seem trivial, however, it is an important insight into how the process of enrollment needs to be streamlined for large systems. In such systems, the authority is envisioned to have a hardened web portal accessible from the corporate LAN. This level of access allows authorized personnel to reach the portal using cellular IP technologies and a VPN.

The web portal would likely be secured using a commodity TLS certificate and the users authenticated using strong passwords and a second factor like a security token. The authority itself would likely reside in the DMZ, thus proper procedures will need to be followed to provide this access. Prior to commissioning a new field asset, a privileged user would grant the user commissioning the field asset the permission to generate a certificate for the asset. Thus the authority would maintain a database of a few items:

- An editable set of field assets that will require enrollment.
- A means of authenticating users and roles/permissions. This information is likely to come from an external identity management system.

The database will already be configured by the system administrator with all of the authorized metadata for each certificate in question. The only piece of information the person generating the outstation certificate needs to provide once properly logged in is the outstation public key. Outstation certificates will be very long lived, likely for the lifetime of the equipment. A cryptographic break in the algorithm underlying the certificate signature will require that a new certificate be installed, so this algorithm should be chosen prudently.

It's important to note that while the authority could be a standalone application with its own complete database of outstations, masters, and users, it might also leverage data available in other systems. For instance, an LDAP server or other enterprise identity system could be used to establish the identity and permissions of users. The authority might also be capable of keeping its database of outstations in the system synchronized with a utility EMS.

Allowing system administrators to pre-configure which users can generate certificates for which outstations and providing access to this part of the authority through proper safeguards will substantially streamline the process of enrollment and reduce the extent to which adding security impacts operations. The portal approach also limits direct access to signing keys and provides a central point for creating an audit trail regarding certificate generation.

4.3.3 Revoking outstation certificates

The master(s) will be capable of reaching a CRL on the authority and will be responsible for checking it at a reasonable interval. The compromise of a single outstation private key is small breach compared to other attack scenarios. Nevertheless, a mechanism must be in place to allow for revocation.

4.3.4 Issuing master certificates

The recommended way to issue master certificates is machine-to-machine (M2M) communication directly from the master to the authority. The reason for this asymmetry is that on a serial network, there is no path for an outstation to reach a certificate revocation list (CRL), and thus a fast expiration scheme allows master certificates to be "revoked" by virtue of the fact that the authority can refuse to renew them. This expiration should happen on the timescale of hours and not days.

The compromise of a master private key is a significant security event, since that master may be authorized to control a significant amount of field equipment. There is no fast mechanism for informing outstations on a serial network that a master has been compromised, thus some other mitigation will be needed until the affected certificate expires naturally.

The communication link between the authority and the masters can be secured using a separate, more-traditional PKI. Since the number of masters in the system is low, it could even use pairs of self-signed certificates where the authority has the public key of every master it needs to authorize. This public key would be used to authenticate the certificate sign request for the certificate to be used to authenticate the master to the outstation.

5 The Link Layer

SSP21 specifies a two layer architecture. The non-cryptographic link-layer provides three services to the layers above it:

- **Framing** - A procedure is defined to identify a frame from a stream of bytes.

- **Addressing** - The frame contains source and destination addresses for the transmitter and receiver.
- **Error detection** - All of the header fields and payload are covered by a cyclic redundancy check (CRC).

The link-layer defined in this document should only be considered a default that can be deployed when useful. The core of SSP21 is the message-oriented cryptographic layer. Other layers, such as UDP, could provide all the required services provided by the link-layer.

Since the link-layer does not have any cryptographic protections, it is designed with simplicity in mind and is completely stateless. The CRC is important at this layer to detect data corruption from random sources (EMF, cosmic rays, etc). This check is intended to prevent randomly corrupted payloads from reaching the cryptographic layer. This prevents "tampering" false positives from occurring at the cryptographic layer which would require a completely different organizational response than occasional randomly corrupted frames.

```
[start][destination][source][length][crc-h][payload][crc-p]
```

The frames consist of the following fields. All multi-byte integer fields (including the CRCs) are encoded in little little format.

destination (2-bytes) - The destination field encodes the address of the intended recipient of the frame. Devices shall always set this field to the address of the intended recipient when transmitting. When receiving a frame, devices shall not do any further processing of frames with an unknown destination address.

source (2-bytes) - The source field encodes the address of the transmitting party. The usage of this field may depend on the application layer of wrapped protocol.

length (2-bytes) - Length of the message, including the header and CRC, in bytes.

crc-h (4-bytes) - A 32-bit CRC value calculated over the header (start, destination, source, and length fields). The CRC polynomial is described in detail in the next section.

payload (0 to 4092 bytes) - An opaque payload that is passed to the cryptographic layer. The length is determined by the *length* field in the header. This length shall never exceed 4092 bytes.

crc-p (4-bytes) - A 32-bit CRC value calculated over the payload bytes.

5.1 CRC Polynomial

The CRC polynomial for the SSP21 link frame was selected based on the Hamming distance (HD) offered by several candidate polynomials at different payload lengths. Our candidates included the following polynomials:

notation	DNP3	IEEE 802.3	**Castagnoli**	Koopman
msb first	0x3d65	0x04c11db7	0xf4acfb13	0x32583499
Koopman	0x9eb2	0x82608edb	0xfa567d89	0x992c1a4c

The polynomials provide the following maximum payload lengths (in bytes) at various Hamming distances:

HD	DNP3	IEEE 802.3	Castagnoli	Koopman
8	0	11	34	16
7	0	21	34	16
6	16	33	4092	4092
5	16	371	4092	4092
4	16	11450	8187	8188

Four byte polynomials can provide significantly better error detection across longer payload lengths. The Koopman and Castagnoli polynomials were discovered using exhaustive search techniques and have significantly longer runs of HD = 6 protection than IEEE 802.3. We selected the Castagnoli polynomial because of slightly better HD=8 coverage for very short frames. The error detection properties of this polynomial have also been independently verified by at least two researchers.

The maximum HD=6 payload length of 4092 determines the bound for the maximum link layer frame size allowed by the standard.

6 Cryptographic Layer

The cryptographic layer is messaged-oriented, meaning that framing of the message is accomplished by the layer(s) beneath it. SSP21 uses a handful of message types to establish secure associations, and then transport user data securely from one party to the other.

6.1 Terminology

The key agreement handshake in SSP21 is a request-reply protocol, thus are two parties: an `initiator` and a `responder`. Normally, the initiator is expected to be the SCADA master, and the responder is expected to be an outstation. It's perfectly possible, however, to flip this relationship in certain circumstances, and have the outstation initiate the key agreement. To preserve the generality of the specification the terms `initiator` and `responder` are used in place of master and outstation.

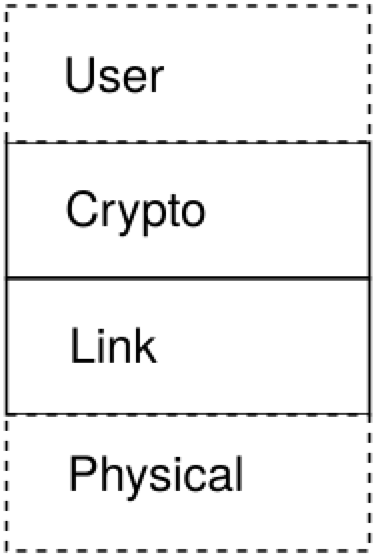

Figure 2: SSP21 stack - The link and crypto layers are defined in this specification

6.2 Algorithms

SSP21 uses a number of cryptographic algorithms. They are described here within the context of the functionality they provide. The initial SSP21 specification contains a minimal subset of algorithms, but the protocol is capable of extension.

The following notation will be used in algorithm pseudo-code:

- The `||` operator denotes the concatenation of two byte sequences.
- The `[b1, b2, .. bn]` syntax denotes a, possibly empty, byte sequence.

6.2.1 Diffie-Hellman (DH) functions

SSP21 currently only supports Curve25519 for session key agreement. It is described in detail in RFC 7748.

DH Curve	length (*DHLEN*)
Curve22519	32

All DH curves will support the following two algorithms with the key lengths specified above.

- `generate_key_pair() -> (private_key, public_key)` - Generate a random public/private key pair.
- `dh(private_key, public_key) -> (result_bytes)` - Given a local private key and remotely supplied public key, calculate bytes of length *DHLEN*.

6.2.2 Hash Functions

SSP21 currently only supports SHA256 described in FIPS 190-4. SHA512 and/or hash function from the BLAKE family will likely be supported in the future. The hash function serves two roles:

- Maintain a hash of all data sent and received during the key negotiation sequence. This running hash is then incorporated into the authentication signatures and makes any tampering of handshake data detectable.
- Used as a sub-function of HMAC to produce authentication tags and derive session keys.

Hash Function	Hash Length (*HASHLEN*)
SHA256	32

6.2.3 Hashed Message Authentication Code (HMAC)

HMAC provides an authentication tag given a shared key and an input message. It is described in RFC 2104. Any hash algorithm described above can be used in conjunction with this construct, and the corresponding HMAC function will produce a tag with the same length as the underlying hash function.

`HMAC(key, message) -> AuthTag` - Calculate an authentication tag from an arbitrary length symmetric key and message bytes.

6.2.4 Authenticated Encryption with Associated Data (AEAD)

This specification describes how to apply AEAD constructs to both authenticate and encrypt session data. AEAD constructs are used in a so called "detached" more when the authentication tag is placed in separate field from the encrypted data. This specification only describes the usage of AEAD constructs that use CTR mode for the encryption operation, and therefore do not require any additional padding. These algorithms include AES-GCM and ChaCha20-Poly1305.

AEAD functions typically accept a nonce larger than the 16-bit nonce used in SSP21. When mapping an AEAD to the SSP21 nonce, the 16-bit nonce shall be placed in the lowest (first) 2 bytes of the nonce buffer in big endian format, and higher bytes shall be set to zero.

6.2.5 Key Derivation Function (KDF)

SSP21 has extensible support for an abstract KDF used during the handshake process.

All KDFs take `salt` and `input_key_material` parameters and return two keys, each 32-bytes in length.

```
  KDF(salt, input_key_material) -> (key1, key2)
```

The `salt` shall always be a publicly known, yet randomized value not controlled by either party independently. The `input_key_material` shall always be, at least partially, a high entropy secret value known only to the two parties.

6.2.5.1 HKDF

The default KDF is HKDF defined in RFC 5869.

HKDF is always be used to derive a pair of 256-bit session keys. This is simple to implement when the block size of the hash function is also 32-bytes (e.g. SHA-256):

```
HKDF(salt, input_key_material) -> (key1, key2)
{
    // extract
    set temp_key = HMAC(salt, input_key_material)
    // specialized expand
    set key1 = HMAC(temp_key, [0x01])
    set key2 = HMAC(temp_key, key1 || [0x02])
    return (key1, key2)
}
```

6.2.6 CSPRNG

A cryptographically secure pseudo-random number generator (CSPRNG) is required for different functions, depending on the handshake mode:

- the generation of ephemeral DH keys in all public-key modes that provide forward secrecy
- the generation of random nonces in symmetric key modes.

Any secure RNG will do, put implementers should err on the side of caution and prefer one provided by the operating system, or a wrapper provided by a trusted library.

6.3 Messages

Every message at the cryptographic layer begins with a one octet message type identifier called the `Function` enumeration. The remaining octets are interpreted according the defined structure of that type.

6.3.1 Syntax

SSP21 uses a formal syntax to define the contents of messages. Using such a syntax has a number of advantages:

- The syntax separates the contents of messages from how they are serialized on the wire
- It ensures that messages are always defined in a consistent way and composed from the same primitives
- Code generation can be more easily leveraged to create encoders and decoders

The SSP21 message syntax is similar to other message definition schemes such as ASN.1, but is specifically designed for security-oriented applications instead of generic application layer messaging, namely:

- The syntax is simple, limited, and only accommodates the requirements of this specification.
- Any given message has one and only one valid serialization, similar to ASN.1 DER.
- String types are intentionally not provided as they tend to lead to abuse and vulnerabilities.

- Self-describing serialization is not an objective like the tag, length, value (TLV) serialization in ASN.1 BER or DER.
- The amount of memory a message will require to deserialize is always a constant known at compile-time.

6.3.1.1 Structs

Groupings of fields are called Structs. Structs use the following syntax:

```
struct <struct-name> {
  <field1-name> : <field1-type>
  <field2-name> : <field2-type>
  ...
  <field3-name> : <field3-type>
}
```

`Messages` are special `Structs` whose first field is always a constant value of the `Function` enumeration.

```
message <message-name> {
  function : enum::Function::<function-name>
  <field1-name> : <field1-type>
  <field2-name> : <field2-type>
  ...
  <field3-name> : <field3-type>
}
```

The following primitive types are defined. All multi-byte integers are serialized in network byte order.

- `U8` - 8-bit unsigned integer.
- `U16` - 16-bit unsigned integer.
- `U32` - 32-bit unsigned integer.

The following example defines a struct that provides counts of various types of flowers:

```
struct Flowers {
  num_roses : U8
  num_violets : U16
  num_petunias : U32
}
```

The serialized size of a `Flowers` struct would always be 7 bytes:

```
sizeof(U8) + sizeof(U16) + sizeof(U32) == 7
```

6.3.1.2 Enumerations

Single byte enumerations are defined with the following syntax:

```
enum <enum-name> {
  <name1> : <value1>
  <name2> : <value2>
  ...
  <nameN> : <valueN>
}
```

The following example defines an enumeration of 3 possible color values:

```
enum COLOR {
  RED : 0
  GREEN : 1
  BLUE : 2
}
```

Enumeration types can be referenced from within a *Struct* or *Message* definition using the following notation:

```
struct <struct-name> {
  <enum-field-name> : enum::<enum-name>
}
```

Using the COLOR example above we could define a *Struct* that represents the intensity of a single color:

```
struct Intensity {
  color : enum::COLOR
  value : U8
}
```

6.3.1.3 Bit fields

Bit fields are single-byte members of *Structs* or *Messages* that encode up to eight boolean values, one value for each bit using the following syntax:

```
bitfield <bitfield-name> { <top-bit-name>, ... <bottom-bit-name> }
```

Bit fields can have zero to eight member bits. The top bit (0x80) is always implicitly defined first in the list of bit names. Unspecified bits shall always be encoded as zero. Parsers shall fail parsing if any unspecified bit is set in the input.

```
bitfield Flags { "flag1", "flag2", "flag3" }
```

The bit-field above with flag1 = true, flag2 = false, and flag3 = true would have a serialized representation of 0b10100000 (0xA0). An example of input that would fail a parsing for this bit field is 0b10100010 (0XA2).

6.3.1.4 Sequences

`Sequences` are variable length lists of a particular type. There are two categories of sequences:

- **`SeqOf[Byte]`** - Denotes a sequence of bytes, like a key, signature, etc.
- **`SeqOf[<struct type>]`** - Denotes a sequence of a defined structure type.

6.3.1.4.1 Variable Length Count

When serialized, all sequence types are prefixed with a variable length count of objects up to 2^32 - 1 (unsigned 32-bit integer). This encoding is similar to the ASN.1 DER encoding of length, but places a constraint on the maximum value. The table below summarizes boundary values and their encodings.

num bytes	value	encoding
1	0 (min)	[0x00]
1	127 (max)	[0x7F]
2	128 (min)	[0x81, 0x80]
2	255 (max)	[0x81, 0xFF]
3	256 (min)	[0x82, 0x01, 0x00]
3	65535 (max)	[0x82, 0xFF, 0xFF]
4	65536 (min)	[0x83, 0x01, 0x00, 0x00]
4	16777215 (max)	[0x83, 0xFF, 0xFF, 0xFF]
5	16777216 (min)	[0x84, 0x01, 0x00, 0x00, 0x00]
5	4294967295 (max)	[0x84, 0xFF, 0xFF, 0xFF, 0xFF]

The following rules apply to encoding and decoding:

1) The first byte may represent a number in the interval [0,127] OR if the top bit is set, the lower 7 bits encode the count of bytes that follow.
2) If the top bit of the first byte is set, the lower 7 bits may only be one of a value in the range 1 to 4.
3) Values shall always be encoded in the fewest bytes possible, and parsers shall always reject such invalid encodings.

6.3.1.4.2 Examples

An example of a struct containing a sequence of bytes:

```
struct ByteSequence {
  value : SeqOf[Byte]
}
```

Given the message definition above, the ByteSequence with value equal to {0xCA, 0xFE} would be encoded as:

```
[0x02, 0xCA, 0xFE]
```

An example of a struct containing a sequence of structs:

```
struct NumberPair {
  first : U8
  second : U8
}

struct Pairs {
  values : SeqOf[struct::NumberPair](max = 5)
}
```

Note that the count of NumberPair structures limited to 5.

Suppose that we wish to encode the following sequence of number pairs:

```
{ {0x07, 0x08}, {0x08, 0x09}, {0x0A, 0xCC} }
```

The serialized Pairs message would be encoded as:

```
[0x03, 0x07, 0x08, 0x08, 0x09, 0x0A, 0xCC]
```

The first value of `0x03` is the variable length and indicates that there are 3 NumberPair structs in the sequence. The encoded NumberPair structs directly follow this count of objects.

6.3.1.5 Constraints

Certain field types have optional or mandatory constraints placed on their contents. Constraints on a field are expressed with the following notation:

```
<field-name> : <field-type>(<id-1> = <value-1>, ..., <id-N> = <value-N>)
```

Any field without the trailing constraint syntax (...) is implicitly defined to have no constraints.

For example, a sequence of bytes may have a `count` constraint that defines the required number of elements for the sequence.

```
struct SomeStruct {
  id : SeqOf[U8](count = 16)
}
```

Parsers should always enforce constraints internally and signal errors whenever a constraint has been violated. The table defines the allowed constraints and the field types to which they apply.

Field Type(s)	Constraint ID	Value semantics	Mandatory
SeqOf[U8]	min	minimum number of elements	no
SeqOf[U8]	max	maximum number of elements	no
SeqOf[U8]	count	required number of elements	no
SeqOf[struct::?]	max	maximum number of elements	yes

6.3.2 Definitions

6.3.2.1 Enumerations

Common enumeration types that are used in one or more messages are defined here.

6.3.2.1.1 Function

SSP21 message definitions always begin with a fixed value of the *Function* enumeration. This fixed value allows a parser to determine the type of the message by inspecting the first byte of an opaque message delivered by the link layer. The correct message-specific parser can then be invoked.

```
enum Function {
    REQUEST_HANDSHAKE_BEGIN  : 0
    REPLY_HANDSHAKE_BEGIN    : 1
    REPLY_HANDSHAKE_ERROR    : 2
    SESSION_DATA             : 3
}
```

6.3.2.1.2 Session Nonce Mode

The `SessionNonceMode` enumeration specifies how the nonce (aka message counter) is verified to protect packets from replay.

```
enum SessionNonceMode {
    STRICT_INCREMENT : 0
    GREATER_THAN_LAST : 1
}
```

- **`STRICT_INCREMENT`** - The receiver of a session message will verify that each received nonce is strictly equal to the last valid nonce plus one. This is the default mode and should always be used in session oriented environments like TCP that provide stream integrity and ordering guarantees.
- **`GREATER_THAN_LAST`** - The receiver of a session message will verify that each received is greater the last valid nonce. This mode is intended to be used in session-less environments like serial or UDP and allows for loss of authenticated packets, but also relaxes security allowing a MitM to selectively drop messages from a session. The protocol being protected by SSP21 is then responsible for retrying transmission in session-less environments.

6.3.2.1.3 Handshake Ephemeral

The `HandshakeEphemeral` enumeration specifies what the contents of the `ephemeral_data` field of the handshake request/reply contain.

```
enum HandshakeEphemeral {
    X25519 : 0
    NONCE : 1
    NONE : 2
}
```

- `X25519` - A x25519 DH public key
- `NONCE` - A 32-byte random nonce
- `NONE` - Empty value

6.3.2.1.4 Handshake Hash

The `HandshakeHash` enumeration specifies which hash algorithm will be used during the key agreement handshake to prevent tampering.

```
enum HandshakeHash {
    SHA256 : 0
}
```

- `SHA256` - Use SHA256

6.3.2.1.5 Handshake KDF

The `HandshakeKDF` enumeration specifies which KDF is used to derive session keys and intermediate keys.

```
enum HandshakeKDF {
    HKDF_SHA256 : 0
}
```

- `HKDF_SHA256` - Use HKDF where the HMAC is HMAC-SHA256

6.3.2.1.6 Session Crypto Mode

The `SessionCryptoMode` enumeration specifies the complete set of algorithms used to authenticate (and optionally encrypt) the session.

```
enum SessionCryptoMode {
    HMAC_SHA256_16 : 0
    AES_256_GCM    : 1
}
```

- `HMAC_SHA256_16` - Cleartext user data with the authentication tag set to HMAC-SHA256 truncated to the leftmost 16 bytes.
- `AES_256_GCM` - AES-256 in GCM mode.

Note: AEAD modes have their nonce's length extended as required by the particular algorithm, with the 16-bit SSP21 nonce in the lowest order 2 bytes in big-endian format. Higher order bytes are set to zero.

6.3.2.1.7 Handshake Mode

The `HandshakeMode` enumeration specifies which procedure both parties use to derive session keys. Each mode interprets certain fields in the handshake messages in different ways.

```
enum HandshakeMode {
    SHARED_SECRET : 0
    PUBLIC_KEYS : 1
    QUANTUM_KEY_DISTRIBUTION : 2
    INDUSTRIAL_CERTIFICATES : 3
}
```

Note: Refer to the handshake section for how each mode shall interpret handshake message fields.

- `SHARED_SECRET` - Both parties possess a shared-secret.
- `PUBLIC_KEYS` - Both parties have out-of-band knowledge of each other's public DH key.
- `QUANTUM_KEY_DISTRIBUTION` - Single-use shared secrets delivered via quantum key distribution (QKD).
- `INDUSTRIAL_CERTIFICATES` - Both parties use an authority certificate to authenticate each other's public DH key from a certificate chain.

6.3.2.1.8 Handshake Error

The `HandshakeError` enumeration denotes an error condition that occurred during the handshake process.

```
enum HandshakeError {
    BAD_MESSAGE_FORMAT                : 0
    UNSUPPORTED_VERSION               : 1
    UNSUPPORTED_HANDSHAKE_EPHEMERAL   : 2
    UNSUPPORTED_HANDSHAKE_HASH        : 3
    UNSUPPORTED_HANDSHAKE_KDF         : 4
    UNSUPPORTED_SESSION_MODE          : 5
    UNSUPPORTED_NONCE_MODE            : 6
    UNSUPPORTED_HANDSHAKE_MODE        : 7
    BAD_CERTIFICATE_FORMAT            : 8
    BAD_CERTIFICATE_CHAIN             : 9
    UNSUPPORTED_CERTIFICATE_FEATURE   : 10
    AUTHENTICATION_ERROR              : 11
    NO_PRIOR_HANDSHAKE_BEGIN          : 12
    KEY_NOT_FOUND                     : 13
    UNKNOWN                          : 255
}
```

Note: Implementations shall **NEVER** define custom error codes as this can allow implementation fingerprinting.

- `BAD_MESSAGE_FORMAT` - A received handshake message was malformed in some manner, i.e. it was improperly encoded.
- `UNSUPPORTED_VERSION` - The specified protocol version is not supported.
- `UNSUPPORTED_HANDSHAKE_EPHEMERAL` - The requested handshake ephemeral is not supported or doesn't match the handshake mode.
- `UNSUPPORTED_HANDSHAKE_HASH` - The requested hash algorithm is not supported.
- `UNSUPPORTED_HANDSHAKE_KDF` - The requested KDF algorithm is not supported.
- `UNSUPPORTED_SESSION_MODE` - The requested session mode is not supported.
- `UNSUPPORTED_NONCE_MODE` - The requested session nonce mode is not supported.
- `UNSUPPORTED_HANDSHAKE_MODE` - The requested handshake mode is not supported.
- `BAD_CERTIFICATE_FORMAT` - One of the received certificates was improperly encoded.
- `BAD_CERTIFICATE_CHAIN` - The certificate chain contains an authentication or other issue,
- `UNSUPPORTED_CERTIFICATE_FEATURE` - One of the received certificates uses a feature not supported by this implementation.
- `AUTHENTICATION_ERROR` - The responder was unable to authenticate the initiator.
- `NO_PRIOR_HANDSHAKE_BEGIN` - The initiator requested handshake auth, but no prior handshake begin was received.
- `KEY_NOT_FOUND` - In QKD mode, the requested key id was not found.
- `UNKNOWN` - A error code for any unforeseen condition or implementation specific error.

6.3.2.2 Handshake Messages

6.3.2.2.1 Version

All handshake messages contain a version field that allows implementations to determine compatibility and to aid in debugging remotely when compatibility issues arise.

```
struct Version {
   major : U16
   minor : U16
}
```

6.3.2.2.2 Request Handshake Begin

The initiator starts the process of establishing a new session by sending the RequestHandshakeBegin message. This message contains a specification of all of the abstract algorithms to be used during the handshake and the session.

```
struct CryptoSpec {
   handshake_ephemeral      : enum::HandshakeEphemeral
   handshake_hash           : enum::HandshakeHash
   handshake_kdf            : enum::HandshakeKDF
   session_nonce_mode       : enum::SessionNonceMode
   session_crypto_mode      : enum::SessionCryptoMode
}
```

- **handshake_ephemeral** - Specifies the nonce or DH algorithm to be used during the handshake, and implicitly determines the expected length of ephemeral_data.
- **handshake_hash** - Specifies which hash algorithm is used to prevent tampering of handshake data.
- **handshake_kdf** - Specifies which KDF is used for handshake key derivation.
- **session_nonce_mode** - Mode describing how session messages are protected against replay with differing security properties.
- **session_crypto_mode** - Specifies the full set of algorithms used to authenticate (and optionally encrypt) the session

The message also includes some constraints on the session to be established.

```
struct SessionConstraints {
   max_nonce : U16
   max_session_duration : U32
}
```

- **max_nonce** - The maximum allowed value of either the transmit or receive nonce.
- **max_session_duration** - The maximum allowed session duration in seconds after which messages are no longer considered valid.

```
message RequestHandshakeBegin {
   function                 : enum::Function::REQUEST_HANDSHAKE_BEGIN
   version                  : struct::Version
   crypto_spec              : struct::CryptoSpec
   constraints              : struct::Constraints
   handshake_mode           : enum::HandshakeMode
   ephemeral_data           : SeqOf[U8]
   mode_data                   : SeqOf[U8]
}
```

- **version** - Identifies the version of SSP21 in use. Only new versions that introduce non-backward compatible changes to the specification which cannot be mitigated via configuration will increment this number.
- **crypto_spec** - Struct that specifies the various abstract algorithms to be used.
- **constraints** - Struct that specifies constraints on the session.
- **handshake_mode** - Determines how session keys are derived by interpreting `ephemeral_data` and `mode_data`.
- **ephemeral_data** - An ephemeral nonce or public DH key corresponding to the `handshake_ephemeral` in the `CryptoSpec` and possibly constrained by the `handshake_mode`.
- **mode_data** - Additional data interpreted according to the `handshake_mode`. Whether this field is empty or not depends on the mode.

6.3.2.2.3 Reply Handshake Begin

The responder replies to `RequestHandshakeBegin` by sending `ReplyHandshakeBegin`, unless an error occurs in which case it responds with `ReplyHandshakeError`.

```
message ReplyHandshakeBegin {
   function       : enum::Function::REPLY_HANDSHAKE_BEGIN
   version        : struct::Version
   ephemeral_data : SeqOf[U8]
   mode_data      : SeqOf[U8]
}
```

- **ephemeral_data** - An ephemeral nonce or public DH key interpreted according to the `handshake_mode` and corresponding to the `handshake_ephemeral` in the `CryptoSpec` and interpreted according
- **mode_data** - Additional data data interpreted according to the `handshake_mode`.

6.3.2.2.4 Reply Handshake Error

The outstation shall reply to a `RequestHandshakeBegin` with a *ReplyHandshakeError* message if an error occurs. This message is for debugging purposes only during commissioning and cannot be authenticated.

```
message ReplyHandshakeError {
   function : enum::Function::REPLY_HANDSHAKE_ERROR
   version  : struct::Version
   error    : enum::HandshakeError
}
```

- **`error`** - An error code that enumerates possible error conditions that can occur during the handshake.

6.3.2.2.5 Session Data

Session data messages perform two functions:

- They authenticate the initial handshake when transmitted with nonce == 0.
- Any session message (including the initial authentication message) may also transfer authenticated (and possibly encrypted) user data to the other party.

The message uses the following sub-fields:

```
struct AuthMetadata {
    nonce : U16
    valid_until_ms : U32
}
```

- **`nonce`** - An incrementing nonce that provides protection from replay of session messages.
- **`valid_until_ms`** - A relative millisecond timestamp since session initialization as defined in the section on key negotiation.

```
message SessionData {
   function : enum::Function::SESSION_DATA
   metadata : struct::AuthMetadata
   user_data : SeqOf[U8]
   auth_tag : SeqOf[U8]
}
```

- **`metadata`** - The Metadata struct is covered by the authentication mechanism of the negotiated *Session Mode*.
- **`user_data`** - A blob of (possibly encrypted) user data.
- **`auth_tag`** - A tag used to authenticate the message.

6.4 Key Agreement Handshake

Key agreement in SSP21 is a single request/response message exchange whereby both parties derive a common set of session keys using a procedure determined by the `HandshakeMode` specified by the initiator. This initial message exchange does not authenticate the parties to each other. The parties must then prove to each-other that they derived the same keys by then transmitting an initial `SessionData` message in each direction. A successful handshake involves the exchange of the four messages depicted in figure 3.

The same messages are exchanged in the same order, regardless of which `HandshakeMode` is in use. Only the interpretation of certain fields and the procedure for deriving sessions keys differs between modes. The authentication step is always identical for every mode. The steps for a successful handshake are summarized below.

- Mode specification and key derivation (1-RTT)
  - The initiator sends a `RequestHandshakeBegin` message specifying the `HandshakeMode` and `CryptoSpec`.
  - The responder replies with a `ReplyHandshakeBegin` or a `ReplyHandshakeError` message.
  - Both parties derive session keys according to the procedure specified by the initiator
- Authentication and optional data transfer (1-RTT)
  - The initiator sends a `SessionData` message with nonce equal to zero.
  - The responder authenticates the message and replies with a `SessionData` message with nonce equal to zero.

These initial `SessionData` messages with nonce equal to zero are syntactically identical to other `SessionData` messages, however, the following differences apply:

- The nonce of zero identifies that they are a special case, and are processed according to special rules.
- A responder may reply to an initiator's initial `SessionData` message with a `ReplyHandshakeError`.

Because of their special status and processing rules, we define aliases for these messages:

- A `SessionData` message with a nonce equal to zero sent by an initiator is called a `SessionAuthRequest` message.

- A `SessionData` message with a nonce equal to zero sent by a responder is called a `SessionAuthReply` message.

These aliases do not define new wire-level message types. That are purely used as a shorthand for the purpose of specification. Implementations will want to direct parsed `SessionData` messages to the correct handler if the nonce is zero or greater than zero. A `SessionData` message for a previously authenticated session shall always use a nonce greater than zero, whereas the session authentication messages shall always use a nonce equal to zero.

Initiators and responders may optionally transfer user data in these messages. This mechanism effectively makes the handshake process a single round trip (1-RTT) request and response. Certain implementations may not wish to transfer user data until fully authenticated. Such implementations may send zero-payload session authentication messages and remain wire-level compatible.

A previously valid session (keys, nonce values, start time, etc) shall not be invalidated until a session authentication message is received and authenticated using the new session keys. Implementations may wish to implement this behavior using two data structures, one for an `active session` and one for a `pending session`.

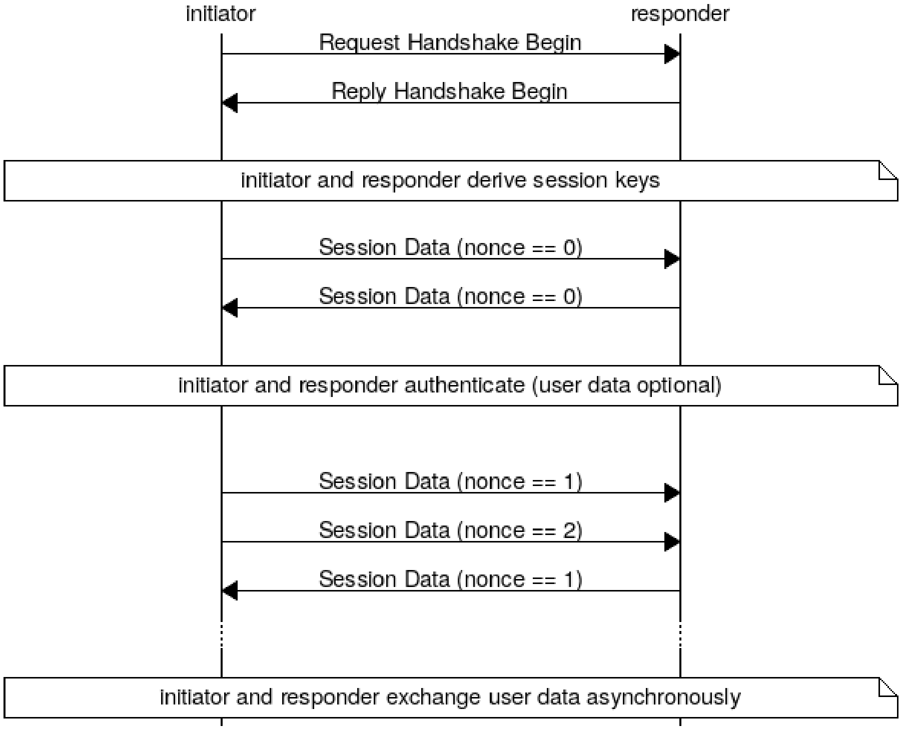
initiator
responder
Request Handshake Begin
Reply Handshake Begin
initiator and responder derive session keys
Session Data (nonce == 0)
Session Data (nonce == 0)
initiator and responder authenticate (user data optional)
Session Data (nonce == 1)
Session Data (nonce == 2)
Session Data (nonce == 1)
initiator and responder exchange user data asynchronously

Figure 3: Successful session establishment

6.4.1 Timing Considerations

In the procedure that follows, the initiator and responder establish a common relative time base so that future session messages can be stamped with a time-to-live (TTL) value since the beginning of the session. This TTL value allows either side of the connection to detect delays induced by a man-in-the-middle. This TTL is has its limitations. An attacker with the ability to delay the handshake messages, can manipulate the common time base calculated by the initiator within the configured response timeout for initiator.

All initiator implementations shall implement a response timeout to the handshake message(s). This timeout shall default to 2 seconds. If a timeout occurs before receiving a valid response, the current handshake attempt shall be aborted. This ensures that attackers cannot skew the common time base by more than this timeout parameter. Implementations should enforce a relatively low maximum value for this parameter to ensure that users do not accidentally deploy systems vulnerable to large session time manipulations

6.4.2 Abstract Handshake Interfaces

Mode specific interfaces are defined in the next two sections. The term *interface* is used to describe an abstract implemenation consisting of both data and functions that operate on the data and inputs. They are described using a pseudo-code. Implementations are not required to use these abstractions when implementing the standard. They are provided as a mechanism to concisely specify the required behaviors and error handling in a manner that separates the generic handshake proceedure from the specifics of any particular mode.

6.4.2.1 Initiator Handshake Interface (IHI)

The IHI consists of two abstract methods:

```
interface InitiatorHandshake
{
    /*
    * Returns the ephemeral_data and mode_data for the this handshake_mode
    *
    * Both of the return values are sequences of byte, possibly empty
    */
    abstract initialize() -> (ephemeral_data, mode_data)

    /*
    * Validate the reply and return the IKM or ABORT
    */
    abstract validate(reply: ReplyHandshakeBegin) -> IKM or ABORT
}
```

ABORT means “abort the handshake” and retry later if configured to do so.

6.4.2.2 Responder Handshake Interface (RHI)

The RHI consists of a single method:

```
interface ResponderHandshake
{
    /*
    * Partially validate the request and return the pair
    * (ReplyHandshakeBegin, IKM) or an error
    */
    abstract validate(request: RequestHandshakeBegin)
        -> (ReplyHandshakeBegin, IKM) or HandshakeError
}
```

HandshakeError is the enumeration with the same name. Returning this enumeration means that the responder will abort the handshake proceedure and return the ReplyHandshakeError message reporting the HandshakeError that occured.

The ResponderHandshake interface always validates the following parts of RequestHandshakeBegin:

1. The mode_data bytes must conform to the handshake_mode.
2. The ephemeral_data bytes must conform to the handshake_mode.
3. The spec.handshake_ephemeral enumeration must agree with the handshake_mode.

Other validation is left to the generic handshake proceedure in the following sections.

6.4.3 Generic Handshake Procedure

The steps in this section are always performed during a successful handshake, regardless of which HandshakeMode is requested by the initiator. The generic procedure reference the mode-specific interfaces. It is important to understand that while the specifics of the mode sub-procedures vary, the following properties always hold for any given mode:

- ephemeral_data is always generated dynamically or it is empty.
- mode_data is always a deterministic value or it is empty.

Notation:

- Both parties maintain a handshake hash denoted by the variable **h** which is HASH_LEN in length.
- The HASH() function always refer to the hash function requested by the master in the ‘RequestHandshakeBegin message.
- NOW() is a function that returns the current value of a relative monotonic clock as a 64-bit unsigned count of milliseconds.
- RANDOM(count) is a function that returns an array of cryptographically random bytes which is count in length.
- [] denotes an empty array or sequence
- || denotes the concatenation operator

Symmetric session keys in this this section use the following abbreviations:

- tx_sk - transmit session key
- rx_sk - receive session key

6.4.3.1 Initiator Handshake Procedure

1. The initiator prepares a `RequestHandshakeBegin` message to send to the responder.

   ```
   set request = IHI.initialize()
   ```

2. The initiator sets `h` to the hash of the fully-initialized serialized request:

   ```
   set h = HASH(request)
   ```

3. The initiator transmits the message and records the time of transmission for future use:

   ```
   set time_tx = NOW()
   ```

4. The initiator starts a response timer that will be used to abort the handshake if a `ReplyHandshakeBegin` message is not received before the timeout occurs.

5. Upon receiving the `ReplyHandshakeBegin` message before the timeout:

   - The initiator cancels the response timer.
   - The initiator estimates the session start time:

     ```
     set session_start_time = time_tx + ((NOW() - time_tx)/2)
     ```

   - The initiator mixes the entire received response into *h*.

     ```
     set h = HASH(h || response)
     ```

   - The initiator uses the IHI to validate the response and obtain the IKM:

     ```
     set IKM = IHI.validate(response)
     ```

     If an error occurs, the initiator aborts the handshake.

     Otherwise, the initiator performs session key derivation using the KDF requested in `RequestHandshakeBegin`:

     ```
     set (tx_sk, rx_sk) = KDF(h, IKM)
     ```

     The initiator then initializes the `pending session` with the session keys, requested algorithms, and `session_start_time`.

6. The initiator uses the `pending session` to transmit a `SessionAuthRequest` message. The initiator may optionally transfer user data in this message. If no user data is available, then the user data shall be empty. All of the fields are calculated in the same manner as an active session, with the exception that the nonce is fixed to zero.

7. The initiator starts a response timer that will be used to terminate the handshake if a `SessionAuthReply` message is not received before the timeout occurs.

8. Upon receiving a `SessionAuthReply` message before the timeout, the initiator uses the `pending session` to validate and authenticate the message. This procedure is identical to an active session, with the exception that the nonce is required to be zero.

9. If the reply authenticates, the initiator replaces any `active session` with the `pending session`. Any previously active session is invalidated.

6.4.3.2 Responder Handshake Procedure

The responder handshake is relatively stateless compared to the initiator. It does not require the use of a timer, and a single flag can be used to track whether the `pending session` is initialized or not.

6.4.3.2.1 Processing RequestHandshakeBegin

Upon receiving a `RequestHandshakeBegin` message:

1. The responder records the time the request was received:

   ```
   set session_start_time = NOW()
   ```

2. The responder sets *h* equal to the hash of the entire received request:

   ```
   set h = HASH(request)
   ```

3. The responder examines the `Version` field of the received request and if the major version does not match the responder's major version it replies with `HandshakeError` set to `UnsupportedVersion` and does no further processing.

4. The responder uses the RHI to validate the request and prepare a response:

   ```
   set (IKM, ReplyHandshakeBegin) = RHI.validate(request)
   ```

   If an error is returned instead, the responder replies with ReplyHandshakeBegin(error) and performs no further actions.

   The responder mixes the fully prepared response into *h*:

   ```
   set h = HASH(h || response)
   ```

   The responder performs session key derivation using the KDF requested in the `RequestHandshakeBegin` message.

   ```
   set (rx_sk, tx_sk) = KDF(h, IKM)
   ```

   Note: The receive and transmit keys are reversed for the initiator and responder.

   The initiator initializes the pending session with the session keys, requested algorithms, and `session_start_time`.

The responder transmits the `ReplyHandshakeBegin` obtained from the RHI.

6.4.3.2.2 Handling Session Auth Request

1. The responder verifies that the `pending session` is initialized and valid.
2. The responder uses the `pending session` to validate the message. The validation procedure is identical to an active session, with the exception that the nonce is required to be zero.
3. The responder uses the `pending session` to transmit a `SessionAuthReply` message with nonce equal to zero. The responder may optionally transfer user data in this message. If no user data is available, then the user data shall be empty. All of the fields are calculated in the same manner as an active session, with the exception that the nonce is fixed to zero.
4. The responder replaces any `active session` with the `pending session`. Any previously active session is invalidated.

6.4.4 Security Properties

If any of the following properties do not hold, then initiator and responder will not agree on the same pair of session keys.

- If a MitM tampers with the contents of either the `RequestHandshakeBegin` message or the `ReplyHandshakeBegin`, the two parties will have different `h` values which will produce different keys when feed into the KDF.
- If either party is unable to calculate the `input_key_material`, they will be unable to calculate the same keys using the KDF.
- A MitM cannot tamper with the common `time_session_init` by delaying messages by more than whatever timeout setting the initiator uses while waiting for replies from the responder. This ensures that the common time-point, in two separate relative time bases, is at least accurate to within this margin when the session is first initialized.

6.4.5 Message Exchanges

The responder may signal an error after receiving a `RequestHandshakeBegin`:

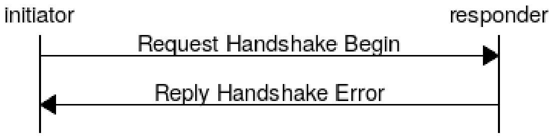

Figure 4: Error in Request Handshake Begin

6.4.6 Handshake Modes

This section defines the various handshake modes that can be used to perform key derivation. The table below summarizes the modes, how they interpret fields in the handshake messages, and whether session modes that encrypt have forward secrecy (FS) if the long-term keys are later compromised.

Handshake Mode	ephemeral data	mode data	key material	FS
shared secret (SS)	random nonces	none	SS + nonces	no
QKD	none	key identifier	qkd key	yes
public keys	DH keys	nonce	triple DH	yes
certificates	DH keys	certificate chain	triple DH	yes

Note: QKD mode provides forward secrecy in that there is no long term key to compromise. Keys used to establish sessions are only used once and then discarded.

A `triple DH` operation performs three DH calculations using both the static and ephemeral DH keys to calculate a shared secret for the two parties. This shared secret is different for every session since the ephemeral keys are different.

Some patterns are apparent in the table:

- Shared-secret mode may only use a random nonce for ephemeral data and never provides forward secrecy.
- The ephemeral data in public-key modes is an ephemeral DH public key. These modes always provide forward secrecy when paired with an encrypting session mode.
- Both public-key modes calculate the input key material in the same manner. In certificate mode, the remote public key is obtained by authenticating the certificate chain and extracting the public key from the peer certificate.

6.4.6.1 Shared secret mode

In shared secret mode, each party possesses the same static 256-bit key.

The `input_key_material` parameter to the KDF is a concatenation of the shared secret and nonces provided by both the initiator and the responder.

6.4.6.1.1 IHI Implementation

The following pseudo-code implements the `InitiatorHanshake` interface for the shared-secret mode:

```
class SharedSecretInitiatorHandshake implements InitiatorHandshake
{
    // class member variables
    set nonce = []
    set shared_secret = <initialized by constructor>

    initialize() -> (ephemeral_data, mode_data) {
        set this.nonce = RANDOM(256) // save the nonce for later
        return (this.nonce, []) // the mode_data is empty
    }

    validate(reply: ReplyHandshakeBegin) -> IKM or ABORT {

        /*
        * this is just a guard that prevents the method
        * from being before initialize() has been called
        */
        if(this.nonce.length == 0) {
            return ABORT("nonce not initialized")
        }

        if(reply.ephemeral_data.length != 32) {
            return ABORT("bad nonce length in reply")
        }

        if(reply.mode_data.length != 0) {
            return ABORT("mode_data must be empty")
        }

        return (this.shared_secret || this.nonce || reply.ephemeral_data)
    }
}
```

6.4.6.1.2 RHI Implementation

The following pseudo-code implements the `ResponderHanshake` interface for the shared-secret mode:

```
SharedSecretResponderHandshake implements ResponderHandshake
{
    // class member variables
    set shared_secret = <initialized by constructor>

    abstract validate(request: RequestHandshakeBegin)
        -> (ReplyHandshakeBegin, IKM) or HandshakeError
    {
       if(request.crypto_spec.handshake_ephemeral != HandshakeEphemeral::NONCE) {
           // doesn't match the handshake mode
           return HandshakeError::UNSUPPORTED_HANDSHAKE_EPHEMERAL;
       }
```

```
        if(request.ephemeral_data.length != 32) {
            return HandshakeError::BAD_MESSAGE_FORMAT;
        }

        if(request.mode_data.length != 0) {
            return HandshakeError::BAD_MESSAGE_FORMAT;
        }

        set nonce = RANDOM(256)

        return (
            ReplyHandshakeBegin(ephemeral_data = nonce, mode_data = []),
            // the IKM
            this.shared_secret || request.ephemeral_data || nonce
        );
    }
}
```

6.4.6.2 Quantum Key Distribution (QKD) mode

In QKD mode, both parties continuously receive a stream of 256-bit keys. A single one of these keys is used to establish an SSP21 session. The initiator signals which key to use using the `mode_data` parameter in `RequestHandshakeBegin` as a key identifier. This key identifer is implementation specific, but must be capable of unambiguously identifying a key without revealing any information about the key. Possible valid key identifiers include:

1. A fingerprint of the key, e.g. a secure hash of the key itself.
2. A sufficiently large increasing counter, with care given to the behavior if the counter ever rolls over due to restart.

The handshake interface definitions that follow make use of two abstract functions to obtain and find keys from some abstract store of keys delivered via QKD:

```
// Used by the initiator to retrieve the next key and the key's identifier
get_next_key() -> (key, identifier)

/**
 * Used by the consumer to retrieve the next key and the key's identifier
 * The Option in the return type denotes that the key may or may not be found
 */
find_key(identifier: Seq[Byte]) -> Option[key]
```

Important: Both of these functions **consume** the key from the key store, i.e. the key in question will never be available again after a call to either get_next_key or find_key.

6.4.6.2.1 IHI Implementation

The following pseudo-code implements the InitiatorHanshake interface for the QKD mode:

```
class QKDInitiatorHandshake implements InitiatorHandshake
{
    // the key is empty until the first call to initialize
    set key = []

    initialize() -> (ephemeral_data, mode_data) {
        set (this.key, identifier) = get_next_key()
        return ([], identifier) // the ephemeral_data is empty
    }

    validate(reply: ReplyHandshakeBegin) -> IKM or ABORT {

        /*
        * this is just a guard that prevents the method
        * from being called before initialize() has been called
        */
        if(this.key.length == 0) {
            return ABORT("key not initialized")
        }

        if(reply.ephemeral_data.length == 0) {
            return ABORT("ephemeral data is not empty")
        }

        if(reply.mode_data.length == 0) {
            return ABORT("mode_data is not empty")
        }

        // the IKM is just the one-time key
        return this.key
    }
}
```

6.4.6.2.2 RHI Implementation

The following pseudo-code implements the ResponderHanshake interface for the QKD mode:

```
QKDResponderHandshake implements ResponderHandshake
```

```
{
    abstract validate(request: RequestHandshakeBegin)
        -> (ReplyHandshakeBegin, IKM) or HandshakeError
    {
       if(request.crypto_spec.handshake_ephemeral != HandshakeEphemeral::NONE) {
           // must be NONE in QKD mode
           return HandshakeError::UNSUPPORTED_HANDSHAKE_EPHEMERAL;
       }

       // use the mode data to lookup the key
       set optional_key = find_key(request.mode_data)

       if(!optional_key.exists) {
           return HandshakeError::KEY_NOT_FOUND;
       }

       return (
           ReplyHandshakeBegin(ephemeral_data = [], mode_data = []), // empty reply
           // the IKM
           optional_key.get_value
       );
    }
}
```

6.4.6.3 Pre-shared public key mode

In pre-shared public key mode, each party has out-of-band prior knowledge of the other party's static public DH key. The `ephemeral_data` field in this mode is an ephemeral public DH key. The `mode_data` field is always empty.

DH keys in this section use the following abbreviations:

- **`ls_pk`** - local static public key
- **`ls_vk`** - local static private key
- **`le_pk`** - local ephemeral public key
- **`le_vk`** - local ephemeral private key
- **`rs_pk`** - remote static public key
- **`re_pk`** - remote ephemeral public key

6.4.6.3.1 IHI Implementation

The following pseudo-code implements the `InitiatorHanshake` interface for the pre-shared public key mode:

```
class PresharedPublicKeyInitiatorHandshake implements InitiatorHandshake
{
    set ls_pk = <initialized by constructor>
    set ls_vk = <initialized by constructor>
    set rs_pk = <initialized by constructor>

    // local ephemeral private key initialized for every handshake
    set le_vk = []

    initialize() -> (ephemeral_data, mode_data) {
        // see section on DH algorithms
        set (le_pk, this.le_vk) = generate_key_pair()
        return (le_pk, []) // mode is empty
    }

    validate(reply: ReplyHandshakeBegin) -> IKM or ABORT {

        if(reply.ephemeral_data.length != LEN_DH) {
            return Abort("ephemeral_data length doesn't match DH algorithm")
        }

        if(reply.mode_data.length != 0) {
            return Abort("mode_data is not empty")
        }

        set dh1 = DH(le_vk, re_pk)
        set dh2 = DH(ls_vk, re_pk)
        set dh3 = DH(le_vk, rs_pk)

        return (dh1 || dh2 || dh3)
    }
}
```

6.4.6.3.2 RHI Implementation

The following pseudo-code implements the `ResponderHanshake` interface for the QKD mode:

```
QKDResponderHandshake implements ResponderHandshake
{
    abstract validate(request: RequestHandshakeBegin)
        -> (ReplyHandshakeBegin, IKM) or HandshakeError
    {
       // validate that the initiator is requesting the same DH algorithm
       if(request.spec.handshake_ephemeral != DH_TYPE) {

           return HandshakeError::unsupported_handshake_ephemeral;
       }

       // mode data must be empty
```

```
        if(request.mode_data.length != 0) {
            return HandshakeError::bad_message_format;
        }

        // validate that the length of the field is as expected
        if(request.ephemeral_data.length != LEN_DH) {
             return HandshakeError::bad_message_format;
        }

        set dh1 = DH(le_vk, re_pk)
        set dh2 = DH(le_vk, rs_pk)
        set dh3 = DH(ls_vk, re_pk)

        return (dh1 || dh2 || dh3)
    }
}
```

Important: This procedure omits validation for every possible type of DH keys. Implementations must validate DH keys are valid for any specific DH algorithm.

Note: The `dh2` and `dh3` calculations are reversed for the initiator and responder, but result in the same values.

6.4.6.3.3 Input Key Material (IKM) Procedure (Responder)

1. Verify that the *handshake ephemeral* is a DH key type.
2. Verify that the length of the `ephemeral_data` field matches the length of the requested DH type.
3. Verify that the *mode data* field is empty.
4. Calculate the `input_key_material`:
    - set dh1 = DH(le_vk, re_pk)
    - set dh2 = DH(le_vk, rs_pk)
    - set dh3 = DH(ls_vk, re_pk)
    - return (dh1 || dh2 || dh3) as the IKM.

Note: The dh2 and dh3 calculations are reversed for the initiator and responder.

6.4.6.4 Industrial Certificate Mode

Certificate mode is similar to the pre-shared public key mode. Instead of having prior knowledge of the remote party's public static DH key, it is embedded in a certificate that is authenticated using the public key of a trusted authority. The calculations of the IKM are identical between the two modes. Instead of being empty, the `mode_data` field contains a certificate chain.

TODO: Decide whether to mostly duplicate the prodcedure or find a way define common points with pre-shared public key mode.

6.5 Sessions

6.5.1 Initialization

Upon completion of a successfully authenticated handshake, the communication session is initialized (or possibly reinitialized) with the following arguments:

- `rx_sk` - A session key used to authenticate/decrypt received messages.
- `tx_sk` - A session key used to sign/encrypt transmitted messages.
- `time_session_init` - The time the session was considered initialized in the local relative time base.
- `read` - A function corresponding to the specified `SessionCryptoMode` used to process a received message's payload:

```
read(key: Key, message: SessionData) -> (cleartext) or Error
```

- `write` - A function corresponding to the specified `SessionCryptoMode` used to prepare a transmitted message's payload:

```
write(key: Key, ad: Seq[Byte], plaintext: Seq[Byte])
    -> (user_data: Seq[Byte], auth_tag: Seq[Byte])
```

- `verify_nonce` - A function used to verify the message nonce:

```
verify_nonce(last_nonce: U16, new_nonce: U16) -> bool
```

The session shall also always maintain a few additional variables initialized internally:

- A 2-byte incrementing nonce (`n`) always initialized to zero, one for each session key.
- A configurable session termination timeout after which the session will no longer be considered valid.

6.5.2 Invalidation

Sessions will only become invalidated after one of the following conditions occurs:

- The transmit or receive nonce reaches the maximum value of 2ˆ16 - 1.

- A configurable amount of time elapses. This session timeout shall default to 1 day and shall not be configurable to be greater than 30 days (the maximum session TTL of a message since initialization is ~49.7 days).
- A complete, authenticated handshake occurs replacing any previously valid session.
- When using session-oriented transports such as TCP, closing the underlying communication layer will invalidate the SSP21 cryptographic session.

Under no condition will malformed packets, unexpected messages, authentication failures, partial handshakes, or any other condition other than the ones listed above invalidate an existing session.

6.5.3 Sending `SessionData`

The following procedure is followed to transmit a `SessionData` message:

- Ensure that the transmit nonce is not equal to the maximum value.
- Increment the transmit nonce by 1 and set this new value on the message.
- Set the `valid_until_ms` to `NOW()` plus the message `TTL`
- Set the `user_data` and `auth_tag` fields using the `write` function with which the session was initialized.

Note: The first transmitted session message from each party always has `n` = 1.

Note: See the TTL session for advice on how to set appropriate TTLs.

6.5.4 Validating `SessionData`

The following procedure is followed to validate a received `SessionData` message:

- Verify the authenticity of the message using the `read` function with which the session was initialized. Upon successful authentication, the cleartext payload is returned.
- Verify that `valid_until_ms <= NOW()`.
- Check the nonce using the `verify_nonce` function with which the session was initialized.
- Set the current nonce equal to the value of the received nonce.

6.5.5 Time-to-live

Each session message contains a time-to-live (TTL) parameter called `valid_until_ms`. This parameter is a count of milliseconds since session initialization (as defined in the handshake section), after which, the receiving party shall not accept an authenticated packet. This TTL prevents attackers from holding back a number of valid session messages and then replaying them in rapid succession.

While this feature provides an additional security guarantee absent in most IT security protocols, it must be configured appropriately.

6.5.5.1 Clock drift

SSP21 Implementations are required to have an accurate relative clock with at least millisecond precision. This standard does not require a particular minimum drift rate from real-time, however, hardware solutions deploying SSP21 should publish information about the maximum possible clock drift.

6.5.5.2 Setting the TTL

A strategy for setting the TTL in each transmitted message must take into account the following factors:

- `session key change interval` (`I`) - The longer the interval between session key changes, the more the relative clocks can drift apart.
- `maximum relative drift rate` (`R`) - The maximum possible drift rate expressed as a number greater than 1. For example, 1.0001 specifies that clocks can diverge by as much as 1/100th of 1% of the time elapsed.
- `initiator handshake response timeout` (`T`) - The longer the response timeout in the handshake, the greater potential mismatch in the session initialization time on the initiator and responder.
- `maximum network latency` (`L`) - The maximum amount of time it might take for a message to reach its destination under normal conditions.

A simple scheme would be to add a fixed value to the current session time as specified below.

```
set current_session_time = session_init_time - NOW()
set max_drift = I * R
set TTL = current_session_time + max_drift + T + L
```

Schemes where the maximum drift dead-band is calculated dynamically based on the elapsed session time are also possible:

```
set current_session_time = session_init_time - NOW()
set max_drift = current_session_time * R
set TTL = current_session_time + max_drift + T + L
```

Regardless of the scheme chosen, implementations shall document whatever method they use for determining the TTL on transmitted packets.

6.5.5.3 Disabling Support

In applications that do not require a TTL, or where no accurate clock is available, implementations may optionally disable support for the TTL.

- In the receive direction, implementations may be configurable to ignore the received TTL entirely.
- In the transmit direction, implementations may be configurable to set the TTL to the maximum value of 2^32 - 1.

6.5.6 Session Crypto Modes

The `SessionCryptoMode` specified by the initiator determines the concrete `read` and `write` functions with which the session is initialized. These functions fall into two general categories: MAC-based functions that only provide authentication, and Authenticated Encryption with Associated Data (AEAD) algorithms that encrypt the payload and additionally authenticate both the payload and associated data in the message.

AEAD implementations such as AES-GCM or ChaCha20-Poly1305 already provide functions that largely match the `read` and `write` methods described here. The only difference being that they also take the nonce as a parameter.

Note: Empty session messages are explicitly disallowed, with the exception of the initial handshake authentication messages. Even authenticated empty messages with `n >= 1` should be treated and logged as an error, and never passed to user layer.

6.5.6.1 MAC Modes

MAC session modes are based on some kind of MAC function, like a truncated HMAC. The write function of these modes can be specified generically in terms of the MAC function.

```
write(key: Key, ad: Seq[Byte], plaintext: Seq[Byte])
    -> (user_data: Seq[Byte], auth_tag: Seq[Byte])
{
    set auth_tag = MAC(key, ad || len(cleartext) || cleartext)
    return (cleartext, auth_tag)
}
```

The MAC is calculated over the concatenation of the following parameters:

- The serialized form of the `AuthMetadata` field (`ad`).
- The length of the cleartext as an unsigned big-endian 16-bit integer
- The cleartext itself

Note: Incorporating the length of the cleartext provides domain separation between `ad` and `cleartext`. This future proofs the specification in the event that `ad` is ever becomes a variable length structure.

The corresponding `read` function calculates the same MAC, compares it for equality using a constant-time comparison, and returns the `user_data` field of the message as the cleartext:

```
read(key: Key, message: SessionData) -> (cleartext) or Error
{
    set tag = MAC(key, message.user_data)
    if(!equals_constant_time(tag,  message.auth_tag)) {
        return Error("authentication failure")
    }

    return messsage.user_data
}
```

Important: Using a constant-time comparison is critical to guard against time-based side channel attacks.

6.6 Certificates

SSP21 defines its own certificate format, the Industrial Certificate Format (ICF). It leverages the definition and serialization rules used for messages within the cryptographic layer. The primary design goals of the ICF include:

- Simplicity - The format and its encodings should be easy to understand and implement, where ASN.1 and its BER/DER encodings are considered undesirably complex for this application.
- Efficiency/Size - The format shall be encodable in the low hundreds of bytes, rather than thousands of bytes to enable usage in bandwidth-limited applications.
- Relevant Metadata - Only metadata relevant to ICS should be included. This explicitly means that much of the metadata in x.509 related to domains/web is not pertinent here.
- Extensible in terms of algorithms - The format shall allow for migration to stronger digital signature algorithms (DSA) in the event of deprecation or break of any particular algorithm.
- Extensible in terms of metadata - It is desirable in certain applications to sign and include additional metadata in the certificate beyond what was defined in the original specification. An example of such an extension would be role based access control (RBAC) permissions for a specific application protocol. The certificate format shall provide the ability to define extensions and define the required behavior when undefined extensions are encountered.

Defining an additional certificate format does not preclude SSP21 from being extended in the future to use X.509.

6.6.1 ICF Definition

The certificate format consists of three components:

- An outer envelope that contains unauthenticated metadata about the issuer, algorithm, and the signature value.
- Inner certificate data fully covered by specified the digital signature algorithm.
- Optional extensions contained within the certificate data itself.

6.6.1.1 Enumerations

The following enumerations are used within the various ICF message definitions.

6.6.1.1.1 PublicKeyType

This enumeration defines the type of the public key embedded in the certificate body.

```
enum PublicKeyType {
    Ed25519 : 0
    X25519  : 1
}
```

- **`Ed25519`** - The key is an Ed25519 DSA public key.
- **`X25519`** - The key is an x25519 Diffie-Hellman key.

6.6.1.2 Certificate Envelope

The certificate envelope is defined as follows:

```
message CertificateEnvelope {
    issuer_id           :      SeqOf[U8](count = 16)
    signature           :      SeqOf[U8]
    certificate_body    :      SeqOf[U8]
}
```

- **`issuer_id`** - A 16-byte digest of the issuer's public key. This digest shall always be the leftmost 16 bytes of the SHA-256 hash of the public key.
- **`signature`** - The value of the signature.
- **`certificate_body`** - The certificate body covered by the specified digital signature algorithm. This data shall not be parsed until the authenticity is established by verifying the signature.

The following digital signature algorithms (DSA) are defined for usage with the ICF.

6.6.1.2.1 Security Discussion

Attackers may freely manipulate the `issuer_id` field with the following impacts:

- The verifying party would be unable to find the corresponding public key in which case the verification would immediately fail.
- The verifying party would apply the wrong public key to the DSA verification, in which case the verification would fail with similar probability as attempting to brute force the signature value.

This `issuer_id` digest is not cryptographic in nature. It merely acts as convenient fixed-length digest for public keys of any length. It must be well distributed against random inputs (i.e. public key values), but need not be cryptographically secure. A break in the underlying SHA-256 hash function does not require changing how the issuer_id is calculated.

The truncated hash used in the issuer id is not cryptographic in nature, and merely needs to be collision resistant against the possible random public keys deployed in the system. It serves to provide a fixed size id for public keys of any size or length.

Attackers may also manipulate the `algorithm` field. Such a manipulation would fail either due to the length of the signature being incorrect, or an signature value.

6.6.1.3 Certificate Body

The certificate body is defined as follows:

```
message CertificateBody {
    serial_number     :      U32
    valid_after       :      U64
    valid_before      :      U64
    signing_level     :      U8(max = 6)
    public_key_type   :      enum::PublicKeyType
    public_key        :      SeqOf[U8]
    extensions        :      SeqOf[struct::ExtensionEnvelope](max = 5)
}
```

- **`serial_number`** - An incrementing serial number assigned by the issuer
- **`valid_after`** - Number of milliseconds since Unix epoch, before which, the certificate shall be considered invalid.
- **`valid_before`** - Number of milliseconds since Unix epoch, after which, the certificate shall be considered invalid.

- **signing_level** - A signing level of zero indicates that the certificate is for an endpoint. Otherwise the certificate is for an authority which may produce any certificate type with signing_level less than its own.
- **public_key_type** - The type of the public key that follows.
- **public_key** - The public key value defined by the public_key_type
- **extensions** - An optional sequence of extensions that define additional required behaviors like application protocol specific whitelists.
- **authority** - The holder of the certificate may produce endpoint certificates or authority certificates with a signing_level less than its own. They may not directly participate as endpoints.
- **endpoint** - The holder of the certificate may act as an endpoint within the system, but may not sign other certificates.

6.6.1.4 Extensions

Certificate extensions have their own envelope that shall be readable by all implementations.

```
message ExtensionEnvelope {
  identifier      : U32
  extension_body  : SeqOf[U8]
}
```

The identifier for each extension shall be unique, and all extensions shall be registered and approved for completeness and suitability with the body maintaining the SSP21 standard. Proprietary extensions are explicitly forbidden. An unknown extension shall always fail verification.

6.6.2 Certificate/Chain Validation

An endpoint certificate may be presented during the SSP21 handshake by itself or as part of a chain. This section describes the validation process for a chain. A standalone certificate is just the special case of a chain where the endpoint certificate is directly signed by the trust root. In the descriptions that follow a single certificate is just a chain of length one.

In PKI mode, trust is fully rooted in the public key of (typically) one authority. This authority may then optionally delegate it's authority to an intermediate signing authorities, or it may directly sign endpoint certificates. End users shall maintain separate signing authorities for masters and outstations so that the compromise of an outstation's private key doesn't enable attacker control of other outstations.

6.6.2.1 Trust Anchors

The device or application running an SSP21 implementation must store the trust anchors in the system: the long-term public keys of one or more authorities

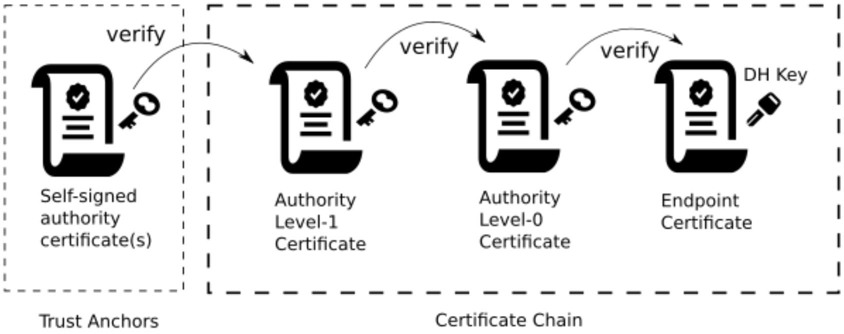
verify
verify
verify
DH Key
Self-signed authority certificate(s)
Authority Level-1 Certificate
Authority Level-0 Certificate
Endpoint Certificate
Trust Anchors
Certificate Chain

Figure 5: Verification of a certificate chain (depth == 3)

embedded in self-signed certificates. How this trust store is implemented is not defined, but some examples include:

- A single certificate loaded into memory during program initialization
- A folder of such certificates with file names corresponding to their `issuer_id`, i.e. SHA-2 hash of the public key

The certificates in the trust store are wholly trusted and used to establish the authenticity of other certificates. Certificates in the trust store are selected based on the `issuer_id` of the first certificate in any chain.

6.6.2.2 Self-signed Certificates

The certificates in the trust store are self-signed, i.e. their signature is computed by the private key corresponding to the public key they contain. As such, no external party verifies their authenticity. Being the trust anchor, they derive their authority merely by their presence on the device.

Even though this could be accomplished with a bare public key, the self-signed certificate approach is preferred for a number of reasons:

- Symmetry - It allows for authority, intermediate signing authority, and endpoint certificates to be dealt with uniformly.
- Bundled configuration - It allows for configuration parameters to be bundled with the authority public key such as validity times and signing depths. For instance, a root authority certificate with signing depth == 1 would not allow for intermediate certificates during verification.

6.6.2.3 Verification Procedure

The goal of certificate chain verification is to ultimately authenticate the final certificate via the chain of trust. Any other result than ending up with a verified certificate at the end of the chain is considered a failed verification. This verification function has the following signature in pseudo code. It will return an error condition if a failure occurs instead of returning a valid terminal certificate.

```
verify(anchors : Seq[CertificateBody], chain: Seq[CertificateEnvelope])
    -> CertificateBody or HandshakeError
```

- **`anchors`** - One or more trusted root certificates.
- **`chain`** - A chain of one or more certificates to be validated against one of the anchors.

If no error occurs, `verify` returns the last parsed certificate in the chain.

The following steps are performed to verify the chain:

1) Upon receiving a certificate chain (certificate count >= 1), the receiving party shall first examine the `issuer_id` in the first (possibly only) certificate in the chain and identify the requested anchor certificate. If the anchor certificate cannot be found, return `HandshakeError::BAD_CERTIFICATE_CHAIN`.

2) Iterate over adjacent certificate pairs using a general purpose function, begining with the selected anchor certificate (A) and the first certificate (C1) in the chain.

```
Parent  Child
|       |
V       V

A       C1    C2    ...     C(n-1)   Cn
```

Verification proceeds rightwards, one certificate at a time, until an error occurs and is returned, or the final pair is validated:

```
                       Parent  Child
                       |       |
                       V       V

A    C1    C2    ... C(n-1)   Cn
```

This sub-function has the the following signature:

```
verify(parent : CertificateBody, child: CertificateEnvelope)
    -> CertificateBody or HandshakeError
```

- **`parent`** - The parsed and verified `body` of the parent certificate.
- **`child`** - The parsed but unverified `envelope` of the child certificate.

The function returns the verified body of the child certificate or an error.

The following sub-steps are performed for each invocation of `verify_pair`:

A) Compare the `issuer_id` in the child envelope to the value calculated over the parent's public key. If they do not match, return `HandshakeError::BAD_CERTIFICATE_CHAIN`.

B) Examine the `public_key_type` in the parent body. If it is not a DSA public key (e.g. it is a DH key), return `HandshakeError::BAD_CERTIFICATE_CHAIN`.

C) Verify that the length of the signature in the child envelope matches the output length of the DSA algorithm specified the parent `public_key_type`. If it does not, return `HandshakeError::BAD_CERTIFICATE_CHAIN`.

D) Verify the DSA signature value in the child envelope using the public key in parent body and the raw bytes of `certificate_body` field of the child envelope. If verification fails, return `HandshakeError::AUTHENTICATION_ERROR`.

E) Fully parse the child certificate body. If parsing fails, return `HandshakeError::BAD_CERTIFICATE_FORMAT`.

F) Verify that parent.valid_after >= child.valid_after. If it is not, return `HandshakeError::BAD_CERTIFICATE_CHAIN`.

G) Verify that parent.valid_before <= child.valid_before. If it is not, return `HandshakeError::BAD_CERTIFICATE_CHAIN`.

H) Verify that parent.signing_level > child.signing_level. If it is not, return `HandshakeError::BAD_CERTIFICATE_CHAIN`.

I) Return the fully verified child body for the next iteration.

3) Return `HandshakeError::BAD_CERTIFICATE_CHAIN` if the `signing_level` if the terminal certificate is not zero.

4) Return HandshakeError::BAD_CERTIFICATE_CHAIN if the `public_key_type` of the terminal certificate is not a DH key.

5) Return the fully verified terminal certificate.