# Performance, Regression & Interoperability Testing for the miTLS component and miPKI library

## Introduction

Development of the miTLS component has continued for some years now, for more information, see https://github.com/project-everest which is the open source project for the development of the component and its support tools and languages. A python-based test harness exists to provide some interoperability testing, but it needs further development, as the feature set and API of the miTLS component frequently changes. There is a need to extend the testing to cover all the TLS versions (including different drafts), extensions, cipher suites, encryption algorithms and named groups which the component now supports.

The original component only supported TLS but now also supports QUIC and the TLS version has moved on from draft 22 to draft 28 to the now official standard. There is also a requirement to include the test harness as part of the CI process. To update the test harness, we need the following features to be present: -

- Support for testing the component through its API
- Support for interoperability testing with other TLS implementations (NSS, OpenSSL, mbedTLS etc)
- Support for measurements
- Support for the gathering of statistics to identify trends in the performance and stability of the component
- Support for all cipher suites, signature algorithms, named groups and other TLS Parameters
- Protocol Verification

Note that the component is provided as a shared library and the test harness makes use of it in this form. There are also several test applications making use of the component, but these are not required for the test harness and in some ways duplicate its operation.

## Support for testing the component through its API

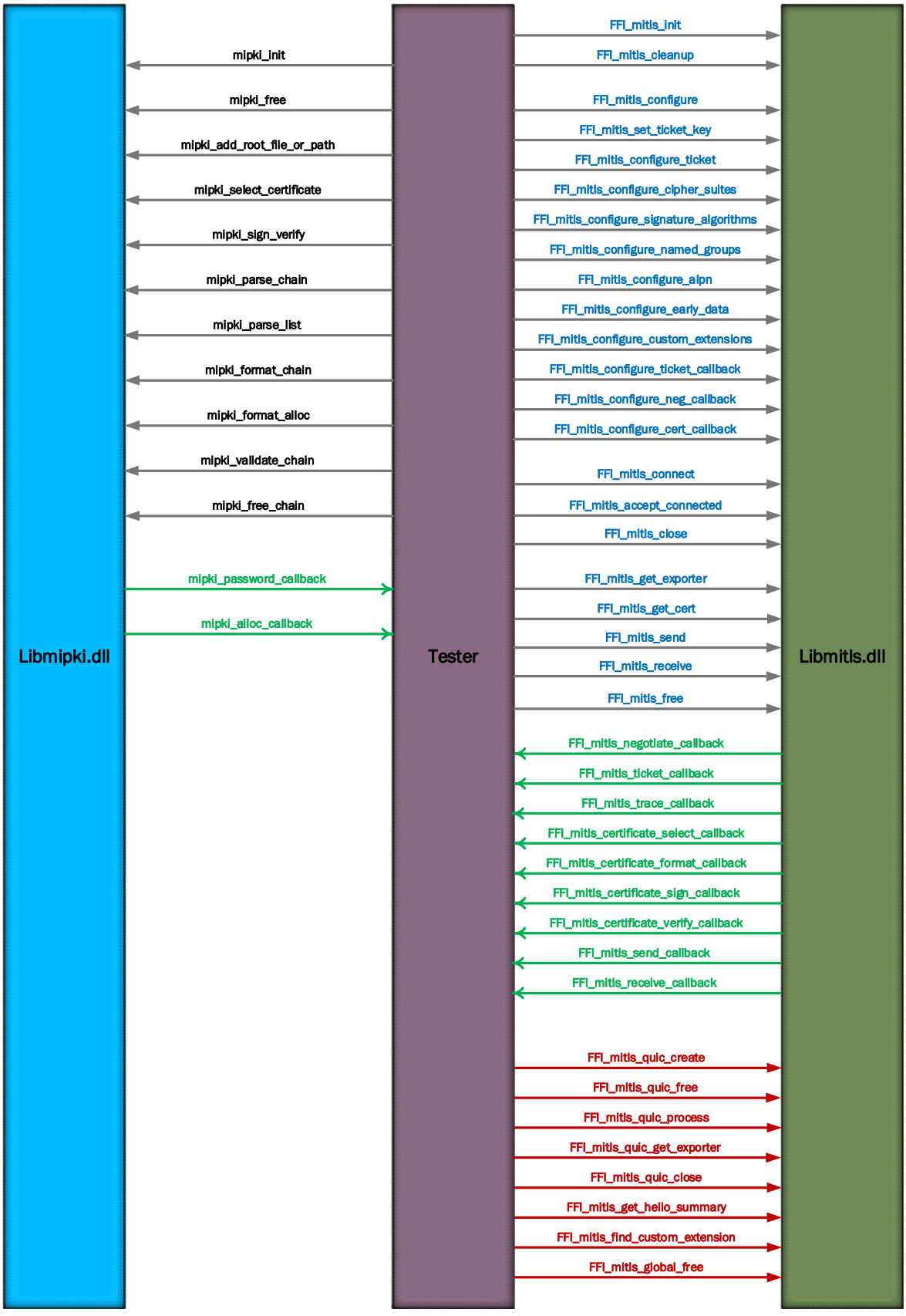
The component is built using a set of development tools which are also undergoing continuous development, especially the F* tool and the HACL, Vale and Kremlin tools. These tools generate source code which is then compiled/assembled/translated into the final component and so the performance of the tools have a bearing on the performance and quality of the component. It is necessary to regularly monitor the performance of the component to ensure it does not deteriorate over time as the performance of the individual tools is not measured.

To this end the test harness will need to frequently exercise the component using its API. The component normally fits into a TCP/IP stack implementation inside layer 4 but it is in fact designed to be transport independent. Therefore, the component needs top, bottom and side interfaces in addition to a configuration API. The top interface is needed so that the application can send data through the component. The bottom interface is so that the component can send the handshake records and the encrypted data to the peer. The side interfaces are required for certificate handling. To ensure platform and transport independence, the component interacts with its environment, using call-back functions, which must be supplied by the environment. In the context of the test harness, the test harness is the environment.

The component API is grouped loosely into two parts, the original TLS API and an additional QUIC API. QUIC also shares the TLS API for configuration. Also shown is the certificate library “libmipki.dll” required for the Tester to implement the certificate call-back functions required by the component which has its own API. The tester does not have to use this library, but since it is provided along with the component, the assumption is that they will be used together in some use cases and the performance of one will impact the performance of the other. Note however that currently, the library is not implemented using verified code but that it is planned to implement it this way.

**Figure 1** below shows the component "libmitls.dll" and its context within the test harness. The names in blue are the TLS API. The names in red are for the QUIC API and the names in green are for the call-back functions. The names in black are for the library API.

Figure 1 Context Diagram for the Tester showing the miTLS Component and the miPKI Library



# Support for Interoperability testing

The component can act as either a client or a server. The peer component may be another miTLS component but is more likely to be a different implementation. It is therefore necessary to test the component in both client and server modes and in conjunction with other implementations. The other implementations of interest are: -

- OpenSSL - a frequently used open source implementation of the predecessor of TLS
- BoringSSL - a fork of OpenSSL from Google, used in the chrome browser.
- mbedTLS - from ARM Limited, as part of their mbed IOT system, previously called PolarSSL
- NSS - a GPL compliant open source library typically used by Google, Mozilla, Oracle etc
- SChannel/SSPI - Microsoft's implementation as used in Windows, an internal competitor to miTLS!
- Fizz - Facebook's C++ implementation (new)

# Support for Measurements

The component implements a well-defined internet protocol. The implementation exchanges messages with its peer (a handshake) to set up the transport layer security and this will take a measurable time. To verify the performance of the component, each defined protocol exchange should be measured.

The component is generated in one of two ways, using the Ocaml compiler or compiled 'c' source generated from the F* tool. There are likely to be performance differences between the two versions, even if they are compliant with each other. Both versions should be tested.

The TLS protocol has two major variants, the full handshake with a corresponding exchange of security key information, and a shorter variant where pre-shared keys are used. Both must be measured. The QUIC protocol also has multiple variants such as 0-RTT with early data and 1-RTT where the server has not been contacted recently.

Then of course there are release and debug versions of the component. Ideally both should be measured, but this may be very difficult for the release version as the existing test harness requires debug output from the component.

## Connections Per Second

The component sets up the transport layer security for more than one connection and so the rate at which these can be established needs to be measured.

## Latency

The time required from the first request from layer five to establish a connection to the first byte of data sent is the latency of the connection. The time taken between the end of the handshake and the first byte of data is also a latency.

## Data Throughput

Once the connection is secured, the data throughput of the component needs to be assessed as additional processing is performed to provide security. The component packages up all data into records rather than as individual bytes, so this is also a measure of data records per second. The measurement should be performed for both transmit and receive as additional processing is also required for deciphering.

## Resources

The component will consume memory when loaded and during operation. Although a heap memory or pre-allocated memory manager is being used, the actual amount of memory used from these memory pools should be measured to verify that the right amount of memory is being allocated.

It is not clear whether other processor resources are to be measured.

# Support for the Gathering of Statistics

The implementation is based on source code generated from other tools and so its performance will vary depending on the quality of those tools. The performance of the component needs to be measured regularly and statistics recorded to identify whether the performance is improving or deteriorating. All the measurements identified above will need to be recorded in a simple database during each test run. To make it easier to identify trends, graphs should be produced for all measurements, but the test harness itself does not need to be graphical.

# Protocol Verification

The TLS protocol is an official internet standard defined in RFC 5246 for Version 1.2 and a draft version of the RFC for Version 1.3 and as such any implementation must conform to the standard. The tester should verify that all messages send and received by the component are compliant with the standard including all known extensions. The component has been designed to be robust against malformed TLS records, but this should also be verified. The following aspects are to be verified: -

1) Fragmentation mechanism when used in conjunction with the max_fragment_size extension.
2) Robustness against deliberately malformed TLS Records. These can be constructed and injected for some messages.
3) Robustness against deliberately malformed extensions.
4) Verification of timeouts when the server is delayed in responding.
5) Version Intolerance. See https://www.thesslstore.com/blog/google-wants-grease-chrome.

## Support for all TLS Parameters

The complete set of cipher suites, signature algorithms and named groups as well as other parameters supported by all versions of SSL/TLS are formally defined by the Internet Assigned Names Authority (IANA) and listed on the webpage https://www.iana.org/assignments/tls-parameters/tls-parameters.xhtml#tls-parameters-8 which also lists the other TLS parameters. Note that each parameter is given formal two-part hexadecimal code.

### Cipher Suites

The cipher suite parameter describes the encryption algorithm to be used for the later parts of the handshake and then the following data. The component does not support every cipher suite mentioned in the IANA list, as some are now so old and insecure that they should not be used. However, TLS is supposed to be backwards compatible, so these algorithms may be supported by an older server. If the component does not support them, for a specific TLS version, then it does not offer them in the ***ClientHello*** message. In this situation, the server is supposed to close the connection with an optional ***Alert*** message since it cannot establish a secure connection with the client, since neither party has an algorithm the other supports. Some servers do not close the connection gracefully and just close the network socket immediately with no reason given.

The cipher suite names passed to the **FFI_mitls_configure_cipher_suites ()** function are not quite the same as those listed in the IANA webpage. In the worst case, the component responds badly (the DLL terminates) if the wrong name is given and will take the calling application or thread with it. In the best case it gives a warning on standard output and then terminates. Either way the function does not return FAILURE (0) as it is supposed to. It's therefore important that the correct names are used.

In addition, more than one suite can be configured by concatenating each cipher suite name required in the same string and separating each name with a colon ":" with the most important algorithm listed first. For example if "TLS_AES_128_GCM_SHA256:TLS_AES_128_CCM_SHA256" is given, this will configure the two variants GCM and CCM of the AES 128 cipher suite. If the configuration function is never called and the TLS version is set to "1.3", the **default** set of cipher suites will be offered as shown in the following table in the same order as they appear in the cipher suites field of the ***ClientHello*** message.

**Table 1 Default Cipher Suites for TLS 1.3**

| Name used in configure () | Formal Name (IANA Name) | Code |
|---|---|---|
| TLS_AES_128_GCM_SHA256 | TLS_AES_128_GCM_SHA256 | 0x1301 |
| TLS_AES_256_GCM_SHA384 | TLS_AES_256_GCM_SHA384 | 0x1302 |
| TLS_CHACHA20_POLY1305_SHA256 | TLS_CHACHA20_POLY1305_SHA256 | 0x1303 |
| ECDHE-ECDSA-AES128-GCM-SHA256 | TLS_ECDHE_ECDSA_AES128_GCM_SHA256 | 0xC02B |
| ECDHE-RSA-AES128-GCM-SHA256 | TLS_ECDHE_RSA_AES128_GCM_SHA256 | 0xC02F |

| | | |
|---|---|---|
| DHE-RSA-AES128-GCM-SHA256 | TLS_DHE_RSA_AES128_GCM_SHA256 | 0x009E |
| ECDHE-ECDSA-AES256-GCM-SHA384 | TLS_ECDHE_ECDSA_AES256_GCM_SHA384 | 0xC02C |
| ECDHE-RSA-AES256-GCM-SHA384 | TLS_ECDHE_RSA_AES256_GCM_SHA384 | 0xC030 |
| DHE-RSA-AES256-GCM-SHA384 | TLS_DHE_RSA_AES256_GCM_SHA384 | 0x009F |
| ECDHE-ECDSA-CHACHA20-POLY1305-SHA256 | TLS_ECDHE_ECDSA_CHACHA20_POLY1305_SHA256 | 0xCCA9 |
| ECDHE-RSA-CHACHA20-POLY1305-SHA256 | TLS_ECDHE_RSA_CHACHA20_POLY1305_SHA256 | 0xCCA8 |
| DHE-RSA-CHACHA20-POLY1305-SHA256 | TLS_DHE_RSA_CHACHA20_POLY1305_SHA256 | 0xCCAA |

The cipher suites "TLS_AES_128_CCM_SHA256 (0x1304)" and "TLS_AES_128_CCM_8_SHA256 (0x1305)" are not offered by default but will be offered if configured.

## Support for all Signature Algorithms

The signature algorithm parameter is used to specify what algorithm should be used in the digital certificates for signing. Since the digital certificates must be decoded and the signature verified to ensure that the certificate is valid, the correct algorithm for the root certificates and intermediate certificates used by servers need to be supported. Just as there are old cipher suites, there are old and proven insecure signature algorithms. The full list described by IANA is not supported by the component and only the algorithms listed in the table below can be used.

Just as for the cipher suites, the signature algorithm names passed to the **FFI_mitls_configure_signature_algorithms ()** function are not quite the same as those listed in the IANA webpage. The component will not respond well to the wrong name being given so the proper names are given in the following table. Similarly, more than one algorithm can be specified by separating the names.

The basic ***ClientHello*** message does not support signature algorithms, so an extension must be used. If the signature algorithm is not set by the configuration function, then a default set will be specified in the "signature_algorithms" extension. The order of algorithms in the extension is the same as the order in the table below.

**Table 2 Default and Supported Signature Algorithms**

| Name used in configure () | Formal Name (IANA Name) | Code | Comment |
|---|---|---|---|
| RSA+SHA1 | RSA_PKCS1_SHA1 | 0x0201 | |
| ECDSA+SHA1 | ECDSA_SHA1 | 0x0203 | |
| RSA+SHA256 | RSA_PKCS1_SHA256 | 0x0401 | |
| RSA+SHA384 | RSA_PKCS1_SHA384 | 0x0501 | |
| RSA+SHA512 | RSA_PKCS1_SHA512 | 0x0601 | |
| RSAPSS_SHA256 | RSA_PSS_RSAE_SHA256 | 0x0804 | |
| RSAPSS_SHA384 | RSA_PSS_RSAE_SHA384 | 0x0805 | |
| RSAPSS_SHA512 | RSA_PSS_RSAE_SHA512 | 0x0806 | |
| ECDSA+SHA256 | ECDSA_SECP256R1_SHA256 | 0x0403 | |
| ECDSA+SHA384 | ECDSA_SECP384R1_SHA384 | 0x0503 | |
| ECDSA+SHA512 | ECDSA_SECP512R1_SHA512 | 0x0603 | |
| RSAPSS_SHA256 | RSA_PSS_SHA256 | 0x0000 | Not sure if these are the same |
| RSAPSS_SHA384 | RSA_PSS_SHA384 | 0x0000 | As 0x0804 etc above |
| RSAPSS_SHA512 | RSA_PSS_SHA512 | 0x0000 | |

## Support for all Named Groups

Named groups are a mechanism for specifying elliptic curve formats and other cryptographic properties. The **ClientHello** message must use extensions to specify the named group and there are several extensions that support this. The main one is the "supported groups" extension which allows the client to list the groups supported by that client. A named group will also be mentioned in the "key share" extension and the component uses both these extensions. The full list described by IANA is not supported by the component and only the algorithms listed in the table below can be used.

Just as for the cipher suites, the group names passed to the **FFI_mitls_configure_named_groups ()** function are not quite the same as those listed in the IANA webpage. The component will not respond well to the wrong name being given, so the correct

names are given in the following table. Similarly, more than one group can be specified by separating the names. If the configuration function is never called then only one supported group is offered, "X25519 (0x001D)".

**Table 3 Default and Supported Named Groups**

| Name used in configure () | Formal Name (IANA Name) | Code | Comment |
|---|---|---|---|
| X25619 | X25519 | 29 | Elliptic Curve groups |
| P-256 | SECP256R1 | 23 | |
| P-384 | SECP384R1 | 24 | |
| P-521 | SECP521R1 | 25 | |
| X448 | X448 | 30 | |
| FFDHE4096 | FFDHE4096 | 258 | Finite Field Diffie Hellman groups |
| FFDHE3072 | FFDHE3072 | 257 | |
| FFDHE2048 | FFDHE2048 | 256 | |

# Component TLS API

Since the component API is not documented anywhere, except for some function declarations in a 'c' header file "mitls-fstar\libs\ffi\mitlsffi.h" and function definitions in a 'c' source file "mitls-fstar\libs\ffi\ffi.c", the API is documented here in considerable detail. Every function in the API is prefixed with the term "FFI_" which denotes the foreign function interface used between programming languages. The component is implemented as a shared library (lib, so, dll) so it must export functions to provide an API. It also requires the environment to provide certain call back functions that the component needs to use. Two main groups of API functions are provided, the first for the original TLS and a newer set for the QUIC enhancement. The TLS API functions can be grouped together as follows: -

- Component Initialisation and Termination functions
  - FFI_mitls_init
  - FFI_mitls_cleanup
- Configuration functions prior to connection
  - FFI_mitls_configure
  - FFI_mitls_set_ticket_key
  - FFI_mitls_configure_ticket
  - FFI_mitls_configure_cipher_suites
  - FFI_mitls_configure_signature_algorithms
  - FFI_mitls_configure_named_groups
  - FFI_mitls_configure_alpn
  - FFI_mitls_configure_early_data
  - FFI_mitls_configure_custom_extensions
  - FFI_mitls_configure_ticket_callback
  - FFI_mitls_configure_nego_callback
  - FFI_mitls_configure_cert_callbacks
- Connection functions
  - FFI_mitls_connect
  - FFI_mitls_accept_connected
  - FFI_mitls_close
- Query functions
  - FFI_mitls_get_exporter
  - FFI_mitls_get_cert
- Data Transfer functions
  - FFI_mitls_send
  - FFI_mitls_receive
  - FFI_mitls_free
- Callback functions (provided by environment)
  - FFI_ticket_cb
  - FFI_cert_select_cb
  - FFI_cert_format_cb
  - FFI_cert_sign_cb
  - FFI_cert_verify_cb
  - FFI_nego_cb
  - FFI_send_callback
  - FFI_receive_callback
  - FFI_trace_callback
- Debug functions
  - FFI_mitls_set_trace_callback

## FFI_mitls_init ()

This function must be called before any other function is called in the component. It performs one-time initialization. There are no parameters and the return value is 0 for failure and 1 for success. The corresponding function to be used when the component is no longer required is **FFI_mitls_cleanup** ().

```
int MITLS_CALLCONV FFI_mitls_init(void);
```

## FFI_mitls_cleanup ()

This function must be called when the component is no longer required. It performs one-time termination. There are no parameters and no return value.

```
void MITLS_CALLCONV FFI_mitls_cleanup(void);
```

## FFI_mitls_configure ()

The component will construct the TLS handshake messages, but to send them to the peer, it requires a network connection to that peer. If no information was provided about that connection, then the TLS messages could not be generated correctly, especially where the Server Name Indicator (SNI) extension is used. This function is used to specify the server (peer) name and the TLS version to be used by the component for this connection.

In addition, each API call from this point forward requires the right internal state information to be provided so that the correct actions are performed. This first call gets back the pointer to that internal state so that it can be passed back to those other functions. Note that the API header file hides all the internal detail that exists in the state variable structure. This is because the internal details could change from version to version, so it is best not to include them. Doing so would tie the user of the component to a specific version of that component, unless it can be guaranteed that the structure will always be backwards compatible.

However other API changes from version to version could render this point moot.

At this stage, it is almost possible for the component to successfully connect to a server if the certificate configuration allowed it because it has default values for the cipher suites, signature algorithms and named groups which work correctly for the server "google.com". However, it is normally necessary at this point to configure those values and then set up the certificate configuration using the **FFI_mitls_configure_cert_callbacks ()** function.

```
int MITLS_CALLCONV FFI_mitls_configure(mitls_state **state, const char *tls_version, const char *host_name);
```

## FFI_mitls_configure_cipher_suites ()

The full list of cipher suites offered by TLS since it was first defined is very long. Many of the cipher suites are now considered unusable since they are proven to be breakable or have known security flaws. Cipher suites have a natural length associated with them and this refers to the size of the keys used rather than the size of the output which can vary depending on the input. For example, the cipher suite called TLS_AES_128_GCM_SHA256 uses a 128 bit key, whereas the suite called TLS_AES_256_GCM_SHA384 uses a 256 bit key and is therefore considered more secure. The names also imply the hashing algorithm to be used to provide message authentication for encrypted messages in the form of the HMAC.

The cipher suites supported by the component are selectable and this function is used to specify them. Note that the API specifies the cipher suites by name rather than using the standardised codes for them. Note also that the component itself does not contain an implementation of the cipher suites as this is provided by an external crypto library. This function only enables the component to make use of one. The cipher suites are specified by name, but more than one cipher suite can be specified by separating them with colons ":" where the first one in the list is the highest priority one followed by successively lower priority ones.

If this function is never called, the component will use the defaults and offer them in the ClientHello. The defaults are specified in the table above.

## FFI_mitls_configure_signature_algorithms ()

Certificates exchanged during the handshake are verified and trusted because they are "signed". This means that a signature algorithm has been run over the content to produce a hash and this hash is added to the message. Different algorithms may be chosen by the server including older, less secure ones and so the client must support these to verify those certificates.

The algorithms supported by the component are selectable and this function is used to specify them. Note that the API specifies the algorithm by name rather than using the standardised codes for them. Note also that the component itself does not contain an implementation of the algorithm as this is provided by an external crypto library. This function only enables the component to make use of one. The algorithms are specified by name but more than one algorithm can be specified by separating

them with colons ":" where the first one in the list is the highest priority algorithm followed by successively lower priority ones e.g. "ECDSA+SHA256:RSA+SHA256" means both algorithms are to be supported with the ECDSA taking priority.

If this function is never called, the component will use the defaults and offer them in the ClientHello extensions. The defaults are specified in the table above.

```
int MITLS_CALLCONV FFI_mitls_configure_signature_algorithms(mitls_state *state, const char *sa);
```

## FFI_mitls_configure_cert_callbacks ()

The component does not handle certificates directly, it expects to make use of a third-party implementation of a certificate manager. Normally the MIPKI Library (described below) would be used for this purpose but that is not mandatory. This library needs to be configured with certificates before this function can be called and the state of the library is passed in as the **cb_state** parameter.

For the component to manage certificates correctly, it needs four functions to be specified, which it can then call back at the appropriate stage in the handshake to select, format, sign and verify the certificates passed in the handshake messages. Rather than have 4 parameters added to the function call, a structure is populated with the call-back function addresses. The address of this structure is then provided in the **cert_cb** parameter. It is up to the user of the component, how these call-back functions are to implement the require operations, but it is assumed that the Library will provide appropriate functions.

When the ***ServerHello*** is received, it will be followed by the ***Certificate*** message which contains a list of one or more X.509 certificates. The "Certificate Verify Call-back" will be invoked to verify each of these certificates. If verification of any certificate fails, then the handshake will also fail. This will manifest as an immediate return from the **FFI_mitls_connect ()** call with a result of 0. Before that, an ***Alert*** message will be sent to the server to signify that the handshake has failed and why.

```
int MITLS_CALLCONV FFI_mitls_configure_cert_callbacks(mitls_state *state, void *cb_state, mitls_cert_cb *cert_cb);
```

## FFI_mitls_connect ()

Once configuration has been performed, the component can be used initially to perform the handshake protocol. The component is designed so that it can be used in any environment and with any transport mechanism. To achieve this independence, it makes use of two call back functions to send and receive the actual TLS records. When these call-backs are called for a connection, there must be some way for the calling system to identify which connection is required since there can be many concurrent connections.

Therefore, a context parameter is required to be passed to the connect () call to provide this context. Note that this value can be anything at all as it is passed as a void pointer. This means it could be a socket identifier, for example, but it could also be a "**this"** pointer for a C++ object or something else entirely. The component makes no assumptions about the context and just assumes that the call-backs will achieve the correct result. Note that the function call does not return until the handshake is finished, so between calling the **FFI_mitls_connect** () function and returning, many calls to the send and receive call-back functions will be made.

It has been observed that the component calls the receive call-back twice for each TLS Record. The first time is to request the 5-octet record header which contains the length of the record to follow and the second time for the record body. It is not yet known how the component will respond when fragmentation occurs.

```
int MITLS_CALLCONV FFI_mitls_connect(void *send_recv_ctx, pfn_FFI_send psend, pfn_FFI_recv precv, mitls_state *state);
```

## FFI_send_callback ()

The component uses this call back function to send the records from the record layer of TLS both during the handshake protocol and to send the application data itself. This function will typically be called twice for each record. The first invocation is for the record header which then contains the record length to follow. The second invocation will then send the rest of the record. During the initial part of the handshake, the TLS records are unencrypted and decodable. When the change cipher key message is received, the component will switch to encrypted mode and all further messages will be encrypted. Note that the record header is still in clear, but the message (fragment) part is encrypted.

TLS supports a compression mechanism, but this is almost never used as compression of encrypted information is technically impossible as most compression algorithms rely on removing redundant or repeated information. IP Header compression may be enabled in the TCP/IP stack, but this is outside the scope of the component.

Notice that no state information is passed back to the environment as it already has access to this after the **FFI_mitls_init** ().

```
typedef int (MITLS_CALLCONV *pfn_FFI_send)(void *ctx, const unsigned char *buffer, size_t buffer_size);
```

### FFI_receive_callback ()

The component uses this call back function to request the records for the record layer of TLS both during the handshake protocol and to receive the application data itself. This function will typically be called twice for each record. The first invocation is for the record header which then contains the record length. This is indicated by the buffer size set to 5. The second invocation will then typically request the rest of the record by setting the buffer size parameter to the record length indicated in the header. The record body then contains all or part of the handshake message.

Because of fragmentation, it is possible that the network will not have provided sufficient content to return the full message yet, so several calls to this function may be made until the complete message is received over multiple records. When only part of the record or message is available, the function returns the amount actually transferred into the buffer. Normally this would be requested amount.

Notice that no state information is passed back to the environment as it already has access to this after the **FFI_mitls_init** ().

```
typedef int (MITLS_CALLCONV *pfn_FFI_recv)(void *ctx, unsigned char *buffer, size_t buffer_size);
```

# Component QUIC API

QUIC stands for Quick UDP Internet Connections and was invented by Google as a way of overcoming several issues with using TCP for downloading webpage content. It uses UDP, as it is not practical to update (fix) the TCP protocol due to lack of upgradability of network infrastructure and middleboxes and other deployment problems. The component does not have a full implementation of the QUIC protocol, just the cryptographic parts. The full implementation makes streaming reliable by implementing TCP like acknowledgements etc. This is an application layer mechanism in the QUIC protocol as defined by Google and the standard. The list of API functions is: -

- Initialisation and Termination Functions
  - FFI_mitls_quic_create
  - FFI_mitls_quic_free
- Connection Functions
  - FFI_mitls_quic_process
  - FFI_mitls_quic_get_exporter
  - FFI_mitls_quic_close
- Miscellaneous Functions
  - FFI_mitls_get_hello_summary
  - FFI_mitls_find_custom_extension_
  - FFI_mitls_global_free

### FFI_mitls_quic_create

```
int MITLS_CALLCONV FFI_mitls_quic_create(quic_state **state, quic_config *cfg);
```

### FFI_mitls_quic_process

```
quic_result MITLS_CALLCONV FFI_mitls_quic_process(quic_state *state, const unsigned char* inBuf, size_t *pInBufLen, un
signed char *outBuf, size_t *pOutBufLen);
```

### FFI_mitls_quic_get_exporter

```
int MITLS_CALLCONV FFI_mitls_quic_get_exporter(quic_state *state, int early, quic_secret *secret);
```

### FFI_mitls_quic_close

```
void MITLS_CALLCONV FFI_mitls_quic_close(quic_state *state);
```

### FFI_mitls_get_hello_summary

```
int MITLS_CALLCONV FFI_mitls_get_hello_summary(const unsigned char *buffer, size_t buffer_len, mitls_hello_summary *su
mmary, unsigned char **cookie, size_t *cookie_len);
```

### FFI_mitls_find_custom_extension

```
int MITLS_CALLCONV FFI_mitls_find_custom_extension(int is_server, const unsigned char *exts, size_t exts_len, uint16_t
 ext_type, unsigned char **ext_data, size_t *ext_data_len);
```

### FFI_mitls_quic_free

```
void MITLS_CALLCONV FFI_mitls_quic_free(quic_state *state, void* pv);
```

### FFI_mitls_global_free

```
void MITLS_CALLCONV FFI_mitls_global_free(void* pv);
```

# miPKI Library API

To provide security and authentication on the internet, digital certificates are used. The system of storing, managing and providing certificates is called the Public Key Infrastructure (PKI). The library file “libmipki.dll” is a reference implementation of an X.509 certificate verification system. It does not have to be used with the component, as the component is designed to use any certificate library, but it is built alongside it and is the suggested solution. The library is used only indirectly by the component as the API functions are accessed via call-back functions from the component after configuration of the library and the component. However, it is included in the test harness because the performance of the library will have an impact on the performance of the component and therefore it must be instrumented and measured. The current library is not generated from verified code, but it is planned to do this later in the project.

Certificates are sent over the connection between the client and the server and are encoded using ASN.1 DER encoding which is multiple octet based. To be useful, the certificates must be decoded, and the library does this internally. However, the required storage space for the decoded certificate is variable and so the library does not allocate memory for the certificate directly but makes use of the **mipki_alloc_callback** () to allocate the required amount. When long certificate chains are involved, the amount of memory required can become significant. When configured as a server, the component has its own set of certificates and keys stored in the filesystem in Base64 format usually and passed in during configuration using the **mipki_init** () function.

The TLS handshake requires the transmission and reception of multiple digital certificates. The server will respond to the ***ClientHello*** with a ***ServerHello*** and this will contain a certificate list with one or more certificates. The client can also provide the server with a certificate list so that the client can be authenticated. It is also possible for intermediate parts of the chain between client and server, such as a firewall, to add to the chain or modify it in some way.

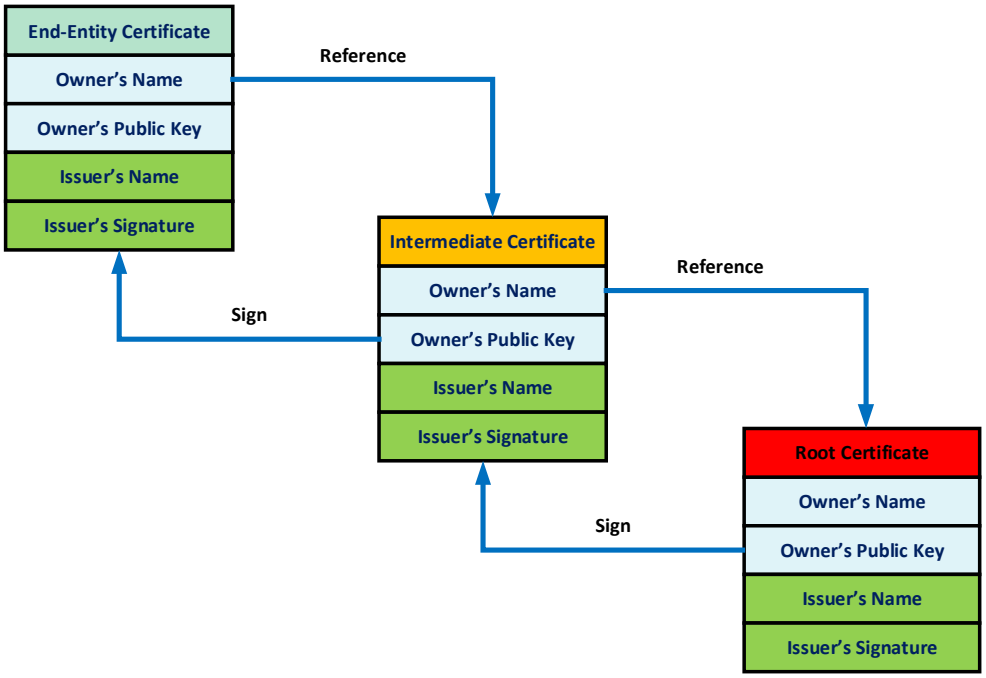
These lists form a certificate chain of trust where each certificate refers to another more trustworthy certificate which refers to another until the root certificate is reached. The root certificate is issued by the Certificate Authority (CA) and represents the ultimate authority in the chain. Root certificates are usually provided with operating systems and can therefore be implicitly trusted.

Certificates are stored in a certificate store and how this is implemented will vary with different operating systems. The structure of a certificate is defined in the X.509 standard, of which V3 is the latest version. Since the component and library are

intended to work on any platform, a specific implementation cannot be assumed. The most common fields in any X.509 certificate are: -

- **Serial Number**: Used to uniquely identify the certificate within a Certificate Authority's systems.
- **Subject**: The entity a certificate belongs to: a machine, an individual, or an organization.
- **Issuer**: The entity that verified the information and signed the certificate.
- **Not Before**: The earliest time and date on which the certificate is valid.
- **Not After**: The time and date past which the certificate is no longer valid.
- **Key Usage**: The valid cryptographic uses of the certificate's public key.
- **Extended Key Usage**: The applications in which the certificate may be used.
- **Public Key**: A public key belonging to the certificate subject.
- **Signature Algorithm**: The algorithm used to sign the public key certificate.
- **Signature**: A signature of the certificate body by the issuer's private key.

In the chain of trust, the owner's name and server signature of a certificate are linked together in a chain. The owner's name refers to the higher (more trusted) authority. The Issuer's signature of the higher authority is used to sign the certificate. The root certificate is signed by itself or Self-Signed. The intermediate certificate(s) will usually have to be installed by the application as they will not come with the operating system. The root certificates usually come with the operating system.




Validating a certificate chain means checking the fields of every certificate in the chain for validity. The API functions can be grouped together as follows: -

- Library Initialisation and Termination functions
  - mipki_init
  - mipki_free
- Configuration functions
  - mipki_add_root_file_or_path
- Certificate functions
  - mipki_select_certificate
  - mipki_sign_verify
  - mipki_parse_chain
  - mipki_parse_list
  - mipki_format_chain
  - mipki_format_alloc

    - mipki_validate_chain
    - mipki_free_chain
- Call-back functions (provided by environment)
    - mipki_password_callback
    - mipki_alloc_callback

### mipki_init ()

The header file describes this as a server function. Every server must provide a valid certificate which can be verified back to a root certificate. This function is used to specify that certificate when the component is used in server mode. However, in client mode, the client must also provide a valid certificate and since there are many concurrent connections to different servers through an instance of the client, then multiple certificates must be provided. This function therefore takes an array of configuration entries in the ***config*** parameter where each entry specifies a single certificate and key and an indication of how the certificate may be used. The number of entries (rather than the size in octets) is indicated by the ***config_len*** parameter. Private keys can be protected with a password and therefore a call-back function must be provided, which the library can invoke, to ask for this password. This call-back function is specified in the ***pcb*** parameter. The library makes use of a certificate store and uses the operating systems implementation of this which can return errors. To get the error back when the initialisation fails, the ***erridx*** address parameter is required since the function returns a pointer to the library's PKI state variable. Note that no information is currently provided on what errors can be returned, or their meaning, but it is assumed that these are windows error codes, when the component is running under Windows.

The PKI state variable is used as the first parameter in most of the other functions in the library. The header file for the library redefines the *mipki_state* structure so that any implementation specific details are hidden.

```
mipki_state* MITLS_CALLCONV mipki_init(const mipki_config_entry config[], size_t config_len, password_callback pcb, i
nt *erridx);
```

### mipki_sign_verify ()

There are many parameters to this function. The last parameter is used to signify if the function should perform signature verification or certificate signing. The second last parameter may catch you out! When verifying, you must pass in the address of a variable containing the length of the unverified signature. This will then be overwritten by the length of the verified signature. Verification will fail if you do not do this.

The first parameter is the usual state variable.

```
int MITLS_CALLCONV mipki_sign_verify(mipki_state *st, mipki_chain cert_ptr, const mipki_signature sigalg, const char *
tbs, size_t tbs_len, char *sig, size_t *sig_len, mipki_mode m);
```

## References


[1**Error! No bookmark name given.**] RFC 5246 - The Transport Layer Security (TLS) Protocol Version 1.2

[2] Transport Layer Security (TLS) Extensions

[3] Transport Layer Security (TLS) Parameters

[4] RFC 5246 Draft 28 - The Transport Layer Security (TLS) Protocol Version 1.3

[5] How TLS Works – An Overview Based on RFC 2246

[6] The TLS Protocol Version 1.0

## Appendix 1 Wireshark decode of ClientHello message

Wireshark is very useful for decoding TLS connections as it has very good and up to date support for SSL Decoding. The listing below is a Wireshark decode of a ***ClientHello*** message going between the component and "google.com" as part of a test run. The content of the message in earlier versions of TLS was relatively simple. To support the needs of "TLS 1.3", a lot of extensions have been added to the message to provide additional information and override some of the values specified in the basic message that must be given for compatibility reasons. The blue sections are the original message as defined in older versions of the specification and the green sections are the new extensions needed to supply the extra information required to support "TLS 1.3".

Note that even if "TLS 1.3" is configured, the ***ClientHello*** message TLS version will be set to "TLS 1.2". In fact, the protocol version of the record header is also set to "TLS 1.2" and the actual version required is specified in the "Supported Versions" extension. Note that in the decode below, the first version value given, "0x7F1C", is actually a draft version (draft 28) of "TLS 1.3" and it is assumed that the server knows how to parse this value since the first octet "0x7F" indicates an experimental or private version.

```
Frame 2199: 250 bytes on wire (2000 bits), 250 bytes captured (2000 bits) on interface 0
Ethernet II, Src: Microsof_2c:0a:32 (00:0d:3a:2c:0a:32), Dst: 12:34:56:78:9a:bc (12:34:56:78:9a:bc)
Internet Protocol Version 4, Src: 20.0.0.4 (20.0.0.4), Dst: google.com (172.217.17.46)
Transmission Control Protocol, Src Port: 51928 (51928), Dst Port: https (443), Seq: 6, Ack: 1, Len: 196
[2 Reassembled TCP Segments (201 bytes): #2197(5), #2199(196)]
Secure Sockets Layer
  TLSv1.2 Record Layer: Handshake Protocol: Client Hello
    Content Type: Handshake (22)
    Version: TLS 1.2 (0x0303)
    Length: 196
    Handshake Protocol: Client Hello
      Handshake Type: Client Hello (1)
      Length: 192
      Version: TLS 1.2 (0x0303)
      Random: 5b7af8f59be0452538342524f639891c6d3f170b98108f53...
        GMT Unix Time: Aug 20, 2018 17:23:01.000000000 Coordinated Universal Time
        Random Bytes: 9be0452538342524f639891c6d3f170b98108f535161a92b...
      Session ID Length: 0
      Cipher Suites Length: 24
      Cipher Suites (12 suites)
        Cipher Suite: TLS_AES_128_GCM_SHA256 (0x1301)
        Cipher Suite: TLS_AES_256_GCM_SHA384 (0x1302)
        Cipher Suite: TLS_CHACHA20_POLY1305_SHA256 (0x1303)
        Cipher Suite: TLS_ECDHE_ECDSA_WITH_AES_128_GCM_SHA256 (0xc02b)
        Cipher Suite: TLS_ECDHE_RSA_WITH_AES_128_GCM_SHA256 (0xc02f)
        Cipher Suite: TLS_DHE_RSA_WITH_AES_128_GCM_SHA256 (0x009e)
        Cipher Suite: TLS_ECDHE_ECDSA_WITH_AES_256_GCM_SHA384 (0xc02c)
        Cipher Suite: TLS_ECDHE_RSA_WITH_AES_256_GCM_SHA384 (0xc030)
        Cipher Suite: TLS_DHE_RSA_WITH_AES_256_GCM_SHA384 (0x009f)
        Cipher Suite: TLS_ECDHE_ECDSA_WITH_CHACHA20_POLY1305_SHA256 (0xcca9)
        Cipher Suite: TLS_ECDHE_RSA_WITH_CHACHA20_POLY1305_SHA256 (0xcca8)
        Cipher Suite: TLS_DHE_RSA_WITH_CHACHA20_POLY1305_SHA256 (0xccaa)
      Compression Methods Length: 1
      Compression Methods (1 method)
        Compression Method: null (0)
```

```
Extensions Length: 127
Extension: supported_versions (len=5)
   Type: supported_versions (43)
   Length: 5
   Supported Versions length: 4
   Supported Version: TLS 1.3 (draft 28) (0x7f1c)
   Supported Version: TLS 1.2 (0x0303)
Extension: key_share (len=38)
   Type: key_share (51)
   Length: 38
   Key Share extension
      Client Key Share Length: 36
      Key Share Entry: Group: x25519, Key Exchange length: 32
         Group: x25519 (29)
         Key Exchange Length: 32
         Key Exchange: 38c6a6fcd33e2aea898d4040df295f2d98303fc8e5d70244...
Extension: server_name (len=15)
   Type: server_name (0)
   Length: 15
   Server Name Indication extension
      Server Name list length: 13
      Server Name Type: host_name (0)
      Server Name length: 10
      Server Name: google.com
Extension: SessionTicket TLS (len=0)
   Type: SessionTicket TLS (35)
   Length: 0
   Data (0 bytes)
Extension: extended_master_secret (len=0)
   Type: extended_master_secret (23)
   Length: 0
Extension: signature_algorithms (len=24)
   Type: signature_algorithms (13)
   Length: 24
   Signature Hash Algorithms Length: 22
   Signature Hash Algorithms (11 algorithms)
      Signature Algorithm: rsa_pkcs1_sha1 (0x0201)
         Signature Hash Algorithm Hash: SHA1 (2)
         Signature Hash Algorithm Signature: RSA (1)
      Signature Algorithm: ecdsa_sha1 (0x0203)
         Signature Hash Algorithm Hash: SHA1 (2)
         Signature Hash Algorithm Signature: ECDSA (3)
      Signature Algorithm: rsa_pkcs1_sha256 (0x0401)
         Signature Hash Algorithm Hash: SHA256 (4)
         Signature Hash Algorithm Signature: RSA (1)
      Signature Algorithm: rsa_pkcs1_sha384 (0x0501)
         Signature Hash Algorithm Hash: SHA384 (5)
         Signature Hash Algorithm Signature: RSA (1)
      Signature Algorithm: rsa_pkcs1_sha512 (0x0601)
         Signature Hash Algorithm Hash: SHA512 (6)
         Signature Hash Algorithm Signature: RSA (1)
      Signature Algorithm: rsa_pss_rsae_sha256 (0x0804)
         Signature Hash Algorithm Hash: Unknown (8)
         Signature Hash Algorithm Signature: Unknown (4)
      Signature Algorithm: rsa_pss_rsae_sha384 (0x0805)
         Signature Hash Algorithm Hash: Unknown (8)
         Signature Hash Algorithm Signature: Unknown (5)
      Signature Algorithm: rsa_pss_rsae_sha512 (0x0806)
         Signature Hash Algorithm Hash: Unknown (8)
         Signature Hash Algorithm Signature: Unknown (6)
```

```
            Signature Algorithm: ecdsa_secp256r1_sha256 (0x0403)
               Signature Hash Algorithm Hash: SHA256 (4)
               Signature Hash Algorithm Signature: ECDSA (3)
                     Signature Algorithm: ecdsa_secp384r1_sha384 (0x0503)
               Signature Hash Algorithm Hash: SHA384 (5)
               Signature Hash Algorithm Signature: ECDSA (3)
            Signature Algorithm: ecdsa_secp521r1_sha512 (0x0603)
               Signature Hash Algorithm Hash: SHA512 (6)
               Signature Hash Algorithm Signature: ECDSA (3)

         Extension: ec_point_formats (len=2)
            Type: ec_point_formats (11)
            Length: 2
            EC point formats Length: 1
            Elliptic curves point formats (1)
               EC point format: uncompressed (0)
         Extension: supported_groups (len=4)
            Type: supported_groups (10)
            Length: 4
            Supported Groups List Length: 2
            Supported Groups (1 group)
               Supported Group: x25519 (0x001d)
         Extension: psk_key_exchange_modes (len=3)
            Type: psk_key_exchange_modes (45)
            Length: 3
            PSK Key Exchange Modes Length: 2
            PSK Key Exchange Mode: PSK-only key establishment (psk_ke) (0)
            PSK Key Exchange Mode: PSK with (EC)DHE key establishment (psk_dhe_ke) (1
```

## Appendix 2 Example DLL output

### A good debug sequence for a working TLS test.

```
FFI| Setting up certificate callbacks.
FFI| Setting a new server negotiation callback.
TLS| writeHandshake (plaintext)
TLS| HS.next_fragment PlaintextID?
HS | next_fragment
HS | offering ClientHello 1.3
KS | ks_client_init 1.3
CDH| Keygen (initiator) on X255519
NGO| No PSK or 0-RTT disabled
NGO| offering client extensions [ supported_versions key_share server_name session_ticket extended_master
_secret signature_algorithms ec_point_formats supported_groups psk_key_exchange_modes ]
NGO| offering cipher suites ; TLS_AES_128_GCM_SHA256; TLS_AES_256_GCM_SHA384; TLS_CHACHA20_POLY1305_SHA25
6; TLS_ECDHE_ECDSA_WITH_AES_128_GCM_SHA256; TLS_ECDHE_RSA_WITH_AES_128_GCM_SHA256; TLS_DHE_RSA_WITH_AES_1
28_GCM_SHA256; TLS_ECDHE_ECDSA_WITH_AES_256_GCM_SHA384; TLS_ECDHE_RSA_WITH_AES_256_GCM_SHA384; TLS_DHE_RS
A_WITH_AES_256_GCM_SHA384; TLS_ECDHE_ECDSA_WITH_CHACHA20_POLY1305_SHA256; TLS_ECDHE_RSA_WITH_CHACHA20_POL
Y1305_SHA256; TLS_DHE_RSA_WITH_CHACHA20_POLY1305_SHA256
HSL| emit ClientHello
TLS| HS.next_fragment returned a fragment
TLS| sendHandshake
TLS| send Handshake fragment with index PlaintextID
TLS| Sending fragment of length 206
REC| record headers: 16030300ce
TLS| writeHandshake (plaintext)
TLS| HS.next_fragment PlaintextID?
HS | next_fragment
TLS| HS.next_fragment returned nothing
TLS| sendHandshake
TLS| read: WrittenHS, Ctrl/Ctrl, nothing
TLS| Read fragment at epoch index: -1 of length 4939
TLS| read Handshake fragment
HS | recv_fragment
```

```
HSL| parsed ServerHello
HSL| end of flight (bytes waiting)
HS | client_ServerHello
NGO| processing server extensions [ extended_master_secret ]
NGO| negotiated 1.2 TLS_ECDHE_RSA_WITH_AES_128_GCM_SHA256
HS | Running classic TLS
HS | Offered SID= Server SID=623a00006cbc7de00405515f3fa57fd340979f659731e6344baccc26bfaa45ce
HS | recv_fragment
HSL| parsed Certificate
HSL| parsed ServerKeyExchange
HSL| parsed ServerHelloDone
HSL| end of flight
HS | processing ...ServerHelloDone
NGO| ServerKeyExchange signature: Valid
CDH| Keygen (responder) on X255519
CDH| DH initiator on X255519
KS | Share: f830c397903c19bc8746420b7b4ddca3dbaef3302142f80940366a07d3e0e96d
KS | Share: fd8f6171b97bd2b5c100a1289ea6b0ef4cc6b678adb0c4147dd38f58bf25fb02
KS | PMS: 9afc79326f70e676d83fe888c65c1c59adc23ff1f72d55ff2d2d617f5605db72
HSL| emit ClientKeyExchange and hash
KS | ks_client_12_set_session_hash hashed_log = 276e0af14e6ce97f1ab48c180a556a9d7f273dcdfa72bc45e4822dcc6
0369167
KS | extended master secret:a0dc417057fa4d35f76d74305d481c08320f837747344c95f2684e1c0ab40b58c3e763bd5fbec
13f627b6ea55f7c3aa6
KS | ks_12_record_key
KS | keystring (CK, CIV, SK, SIV) = 759c0e74e0ec09fd500eda32e8995dc5241de4a93cc36dcf88b1b4f607a9ca48ec704
1483f89c123
AEP| LowProvider: COERCE(K=759c0e74e0ec09fd500eda32e8995dc5)
AEP| LowProvider: COERCE(K=241de4a93cc36dcf88b1b4f607a9ca48)
HS | digest is 276e0af14e6ce97f1ab48c180a556a9d7f273dcdfa72bc45e4822dcc60369167
TLS| readOne ReadAgain
TLS| writeHandshake (plaintext)
TLS| HS.next_fragment PlaintextID?
HS | next_fragment
TLS| HS.next_fragment returned a fragment; CCS; next_keys (for handshake only
TLS| sendHandshake
TLS| send Handshake fragment with index PlaintextID
TLS| Sending fragment of length 37
REC| record headers: 1603030025
TLS| send CCS fragment with index PlaintextID
TLS| Sending fragment of length 1
REC| record headers: 1403030001
EPO| writer++ -1/0
TLS| writeHandshake (encrypted)
TLS| HS.next_fragment ID12?
HS | next_fragment
TLS| HS.next_fragment returned a fragment
TLS| sendHandshake
TLS| send Handshake fragment with index ID12
TLS| Sending fragment of length 40
REC| record headers: 1603030028
TLS| writeHandshake (encrypted)
TLS| HS.next_fragment ID12?
HS | next_fragment
TLS| HS.next_fragment returned nothing
TLS| sendHandshake
TLS| read: WrittenHS, Ctrl/Ctrl, new writer: handshake-only
TLS| ignore handshake-specifc key change; calling read again
TLS| writeHandshake (encrypted)
TLS| HS.next_fragment ID12?
HS | next_fragment
TLS| HS.next_fragment returned nothing
TLS| sendHandshake
TLS| read: WrittenHS, Ctrl/Ctrl, nothing
TLS| Read fragment at epoch index: -1 of length 1
TLS| read CCS fragment
HS | recv_ccs
HS | Processing server CCS (full handshake). No ticket sent.
EPO| reader++ 0/0
TLS| readOne ReadAgainFinishing
TLS| writeHandshake (encrypted)
```

```
TLS| HS.next_fragment ID12?
HS | next_fragment
TLS| HS.next_fragment returned nothing
TLS| sendHandshake
TLS| read: WrittenHS, Ctrl/Ctrl, nothing
TLS| Read fragment at epoch index: 0 of length 40
TLS| StAE decrypt correct.
TLS| read Handshake fragment
HS | recv_fragment
HSL| parsed Finished
HSL| end of flight
TLS| readOne Complete
FFI| Read returned Complete
```

## A debug sequence for a failing TLS test.

Note that the server sends an alert message right after receiving the client hello and then immediately closes the network connection.

```
FFI| Setting up certificate callbacks.
FFI| Setting a new server negotiation callback.
TLS| writeHandshake (plaintext)
TLS| HS.next_fragment PlaintextID?
HS | next_fragment
HS | offering ClientHello 1.3
KS | ks_client_init 1.3
CDH| Keygen (initiator) on P521
NGO| No PSK or 0-RTT disabled
NGO| offering client extensions [ supported_versions key_share server_name session_ticket extended_master
_secret signature_algorithms supported_groups psk_key_exchange_modes ]
NGO| offering cipher suites ; TLS_AES_128_GCM_SHA256
HSL| emit ClientHello
TLS| HS.next_fragment returned a fragment
TLS| sendHandshake
TLS| send Handshake fragment with index PlaintextID
TLS| Sending fragment of length 247
REC| record headers: 16030300f7
TLS| writeHandshake (plaintext)
TLS| HS.next_fragment PlaintextID?
HS | next_fragment
TLS| HS.next_fragment returned nothing
TLS| sendHandshake
TLS| read: WrittenHS, Ctrl/Ctrl, nothing
TLS| sendAlert AD_internal_error (Transport.recv)
TLS| send Alert fragment with index PlaintextID
TLS| Sending fragment of length 2
REC| record headers: 1503030002
TLS| readOne ReadError Transport.send(payload) returned -1
FFI| Read returned ReadError Transport.send(payload) returned -1
FFI| returning error: (None) Transport.send(payload) returned -1
```