TÜV NORD GROUP

RUB

RUHR-UNIVERSITÄT BOCHUM

Development and Evaluation of a TLS-Testsuite

Philipp Nieting

Master's Thesis – May 22, 2022.
Chair for Network and Data Security.

Supervisor: Prof. Dr. Jörg Schwenk
Advisors: M. Sc. Robert Merget
Prof. Dr.-Ing. Juraj Somorovsky
Dipl.-Ing. Eugen Weiss (TÜViT)

hgi Lehrstuhl für Netz- und Datensicherheit

Abstract

This thesis develops a testsuite containing test cases that allow checking the conformance of a TLS implementation to the protocol specification. The testsuite provides the possibility to execute client and server tests and includes test cases for the specifications of TLS 1.2 and TLS 1.3. For high test coverage, a single test case performs multiple handshakes that are automatically derived from a provided one, each negotiating a different TLS cipher suite or applying fragmentation.

For the evaluation, the developed testsuite is executed against the most popular TLS implementations like OpenSSL, BoringSSL and NSS. The results show that there are many different behaviors among the implementations regarding their reaction to TLS messages that do or do not conform to the specification. Such an unspecified behavior can lead to interoperability issues and in specific circumstances even to security problems.

Eidesstattliche Erklärung

Ich erkläre, dass ich keine Arbeit in gleicher oder ähnlicher Fassung bereits für eine andere Prüfung an der Ruhr-Universität Bochum oder einer anderen Hochschule eingereicht habe.

Ich versichere, dass ich diese Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen benutzt habe. Die Stellen, die anderen Quellen dem Wortlaut oder dem Sinn nach entnommen sind, habe ich unter Angabe der Quellen kenntlich gemacht. Dies gilt sinngemäß auch für verwendete Zeichnungen, Skizzen, bildliche Darstellungen und dergleichen.

Ich versichere auch, dass die von mir eingereichte schriftliche Version mit der digitalen Version übereinstimmt. Ich erkläre mich damit einverstanden, dass die digitale Version dieser Arbeit zwecks Plagiatsprüfung verwendet wird.

Official Declaration

Hereby I declare, that I have not submitted this thesis in this or similar form to any other examination at the Ruhr-Universität Bochum or any other institution or university.

I officially ensure, that this paper has been written solely on my own. I herewith officially ensure, that I have not used any other sources but those stated by me. Any and every parts of the text which constitute quotes in original wording or in its essence have been explicitly referred by me by using official marking and proper quotation. This is also valid for used drafts, pictures and similar formats.

I also officially ensure that the printed version as submitted by me fully confirms with my digital version. I agree that the digital version will be used to subject the paper to plagiarism examination.

Not this English translation but only the official version in German is legally binding.

Datum/Date	Unterschrift/Signature

Erklärung

Ich erkläre mich damit einverstanden, dass meine Masterarbeit am Lehrstuhl NDS dauerhaft in elektronischer und gedruckter Form aufbewahrt wird und dass die Ergebnisse aus dieser Arbeit unter Einhaltung guter wissenschaftlicher Praxis in der Forschung weiter verwendet werden dürfen.

Date	Philipp Nieting

Contents

1 Introduction

We use the internet every day for online banking, social media or e-mails and also for parts of our infrastructure. For all of these activities it is important to protect the data sent, to prevent manipulations during the transmission and to preserve privacy. This can be achieved using the transport layer security (TLS) protocol that provides encryption, authentication and integrity to achieve the protection of the data. The protocol is built on top of the transport layer, so it can be used for the majority of application protocols.

The first versions of TLS are called secure sockets layer (SSL) and are cryptographically broken today. The successors, TLS 1.0 and 1.1 are also considered as insecure today, because of the usage of weak algorithms and published attacks [1, 2, 3, 4, 5, 6, 7]. The current versions of TLS that are 1.2 and 1.3 are considered to be secure on up-to-date systems.

1.1 Motivation

Although TLS 1.2 and 1.3 are theoretically secure this does not mean that every communication that uses these protocols is secure. A point of failure is the TLS stack, which is the implementation of the protocol inside a library like OpenSSL, BoringSSL or NSS. These TLS implementations perform all operations that are required by the protocol to perform the handshake, that is, for example, the negotiation of cryptographic keys. They are used by other programs like webservers, browsers, e-mail clients or other software that use the TLS protocol. The advantage of this concept is that the security-related tasks are executed by well-known libraries. Using existing software minimizes, on the one hand, the development time and on the other hand the risk of implementation flaws.

TLS implementations are complex and vulnerable to implementation flaws like every other software. These lead, for example, to security vulnerabilities or interoperability issues between clients and servers. If they are inside a TLS implementation, an attacker might be able to ultimately defeat the protections offered by the TLS protocol [8, 9].

The goal of this thesis is to implement a testsuite for the TLS protocol that is capable of testing the compliance of an implementation with the protocol specification. The testsuite

should be a tool that can be used by developers to test their implementations as well as penetration testers to estimate the quality of a TLS stack. To demonstrate the performance of the developed testsuite, it is used to analyze the most popular TLS implementations that provide an example client or server application.

1.2 Related Work

The idea of a testsuite for the TLS protocol is not new. There are tools available for doing something similar. These projects are either not available as open-source or they are limited in their functionality.

tlsfuzzer. tlsfuzzer [10] is an open-source python project that consists of a fuzzing library and a testsuite. The limitation of this project is, that every test case is a separate program spanning a few hundred lines of code. This makes it complicated to understand the tests and to run all available tests against a target. Another limitation is that tlsfuzzer can only test server implementations.

Achelos TLS Inspector. The TLS Inspector developed by Achelos [11] is a tool to perform automated tests for TLS client and server implementations. These tests cover the RFCs for TLS 1.1, 1.2 and 1.3 as well as additional penetration tests. The tool is not available as open-source. The company also does not provide an overview of security vulnerabilities or other findings that could be found using the testsuite.

Other test tools. Another kind of TLS test tools are compatibility or vulnerability scanners like TLS-Scanner [12], testssl.sh [13], Qualys SSL Labs [14] or How's My SSL [15]. These tools test known vulnerabilities like DROWN [16] or Heartbleed [8], supported cipher suites and other properties of the TLS configurations like HSTS. All the mentioned tools, except How's My SSL, are only able to test TLS server implementations.

For the development and evaluation of the testsuite the following projects are used as the foundation.

- TLS-Attacker
- TLS-Compliance
- TLS-Docker-Library

TLS-Attacker. The TLS-Attacker framework [17] provides a Java implementation of the TLS protocol and complete control over the TLS handshake protocol. This results in the ability to modify every message that is part of the handshake as well as to reorder the messages. TLS-Attacker can act as client and server and thus can be used for writing tests against client and server implementations.

TLS-Compliance. TLS-Compliance is a testsuite developed by Ebert [18] in 2018 as part of a master thesis. This testsuite aimed for developing test cases based on the RFCs describing TLS 1.2 and the belonging extensions. The project only provides a Java program to run the tests against TLS servers that are spawned in Docker containers. The implemented tests are used as the foundation for the developed testsuite.

TLS-Docker-Library. The TLS-Docker-Library project is used for the evaluation part of the thesis. Part of the project are Dockerfiles for more than 20 different TLS implementations of multiple versions. Most of the Docker images run the example TLS server/client applications that are shipped as part of the implementations. This project also contains a Java library to manage the Docker containers.

1.3 Contribution

The main contribution of this thesis is the developed testsuite that provides a high degree of flexibility, an extendable architecture and application programming interfaces (APIs) to write test cases for the TLS protocol efficiently. Furthermore, it is capable of testing TLS 1.2 and TLS 1.3 servers and clients. The testsuite is also designed to be usable with low overhead to test arbitrary TLS implementations.

2 Background

This chapter gives background information to the topics that are required to understand the following chapters. This includes an overview of the TLS protocol in versions 1.2 and 1.3.

Further, the architecture of TLS-Attacker is described that the testsuite uses to perform TLS handshakes. The TLS-Docker-Library is used for the evaluation of server and client implementations.

Since the testsuite is a tool for software testing an introduction to this topic is given as well as to JUnit 5, a testing framework, that the testsuite uses to model the test cases.

2.1 Transport Layer Security Protocol

The Transport Layer Security (TLS) protocol exists to be able to establish a secure communication channel between two peers. The establishment process is called handshake during which messages are exchanged between the peers to negotiate cryptographic parameters. The established channel provides the security properties confidentiality, authenticity and integrity.

Confidentiality is achieved through encryption. TLS uses an asymmetric key exchange mechanism to negotiate a symmetric encryption key between the two communicating parties. For the negotiation of the symmetric encryption key it is possible, to use key agreement or key transport mechanisms. Which mechanism is used, depends on the negotiated cipher suite. A cipher suite specifies which algorithms for key exchange, encryption, signatures and hashing are used.

Authenticity is based on the public key infrastructure (PKI) and signatures that are created during the handshake. This results in the possibility that one or both peers can authenticate towards the other peer. In the World Wide Web most of the time only the server authenticates towards the client so that the client can verify and ensure that it is communicating with the correct server.

Integrity is achieved during the handshake through signatures and after the handshake using message authentication codes (MACs). Since the keys for the MACs are calculated

after the authentication, MACs also provide authenticity. TLS 1.2 supports Encrypt-then-MAC as well as Mac-then-Encrypt which led to problems like padding oracles in the past [19].

The protocol is specified in multiple RFCs. The specification of TLS 1.2 includes more than 10 different RFCs as shown in Figure A.1a. The network of RFCs makes this protocol complex to implement. TLS 1.3 has less RFCs that are needed to be considered for the implementation since it is the newest version of the protocol and therefore a leaf in the network of RFCs (Figure A.1b).

Both versions of TLS use two layers consisting of multiple sub-protocols to abstract the functionality. The foundation is the record layer that belongs to the Record protocol and acts as a data container to encapsulate the payload. The payload is either a message that belongs to a higher level layer that consists of the handshake-, alert-, application- or change cipher spec protocol. Which type of payload it contains, is indicated with a content type field that is part of the record layer. This architecture is visualized in Figure 2.1. After the symmetric encryption keys are negotiated, the encryption is activated on the record layer level. This means that every message that is part of the protocol below the record protocol is encrypted.

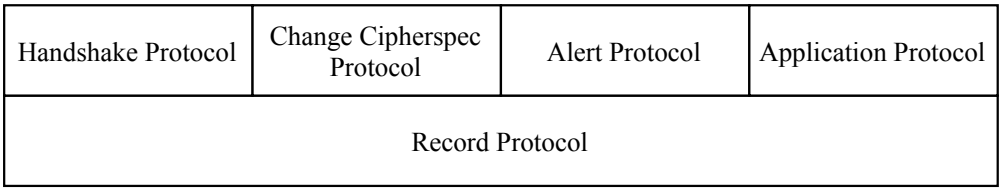


Figure 2.1: Subprotocols of TLS 1.2.

2.1.1 TLS 1.2

TLS 1.2 was specified in 2008. The regular handshake to negotiate cryptographic parameters needs two round trip times (RTT) and involves at least nine messages exchanged between the two peers that are described in the following and visualized in Figure 2.2. Depending on the TLS configuration more messages might be sent. Since the testsuite does not cover every feature of the TLS protocol, such as client authentication, these additional optional messages are not explained in detail.

ClientHello. The ClientHello message is sent by the client and proposes supported TLS properties towards the server. These are, for example, the supported cipher suites and compression modes. In addition, the message contains the latest supported version of TLS as well as a cryptographic secure 32-byte random value that is used later to derive the encryption keys. Optionally, a session id is present that signals to the server that the client wants to reuse a previous established session, which would result in a shorter

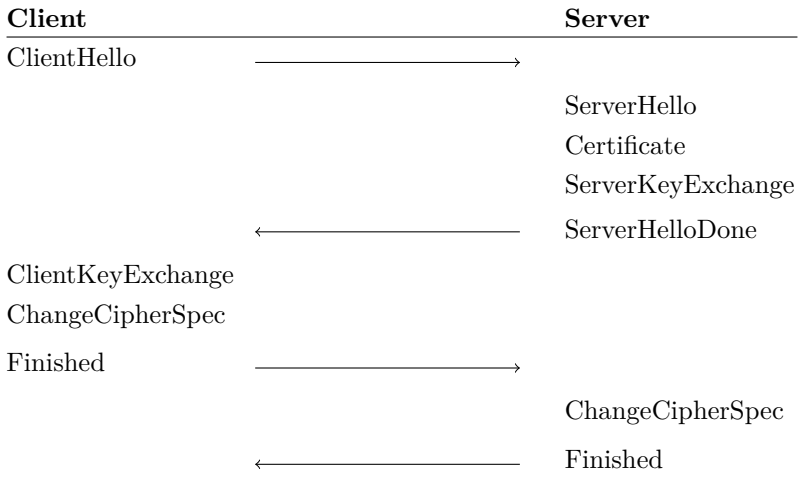


Figure 2.2: Message flow of TLS 1.2

handshake of 1 RTT. A list of extensions can also be part of this message. Since the developed tests also test the implementation of extensions, the most important ones are described in Section 2.1.1.1.

ServerHello. With the ServerHello message, the server selects the algorithms and the TLS version proposed by the client. This message also contains a cryptographic secure random value that is used during the key derivation process. If the client sends a session id that the server recognizes, both parties continue with the Finished message using the same algorithms negotiated in the handshake referenced by the session id.

Certificate. The server usually sends his certificate to the client using the certificate message, except when a cipher suite is negotiated that does not require authentication. The certificate must contain a public key that is compatible with the selected cipher suite. Non-anonymous cipher suites using a key exchange with ephemeral keys require that the key pair can be used for signature operations. For a cipher suite using an RSA key exchange, the key pair must be suitable for encryption operations. When a certificate uses a static Diffie Hellman key, this key must be suited to be used for key agreement.

ServerKeyExchange. The message is only sent by the server if a cipher suite is selected that uses a key agreement key exchange mode with ephemeral keys and thus supports perfect forward secrecy. The key parameters chosen by the server are signed with the private key that belongs to his certificate. If a cipher suite is selected that

does not require server authentication, the chosen parameters are not signed by the server.

ServerHelloDone. This message signals the client that the server is done with the communication and waits for messages sent by the client.

ClientKeyExchange. This message contains the key parameters chosen by the client. In case of an RSA key exchange, the client chooses a premaster secret and encrypts it with the RSA public key of the server that is part of the certificate. If a key agreement cipher suite is selected by the server, the message contains the key parameters of the client. After this message a shared secret is established between both peers, that is the premaster secret. From the premaster secret together with the random values of server and client, the 48 bytes long master secret is derived. The master secret is used together with the random values to derive symmetric encryption and MAC keys as well as initialization vectors for the cipher algorithms. The client and the server use different cryptographic keys.

ChangeCipherSpec. The ChangeCipherSpec (CCS) message contains a single byte set to 1 and tells the receiving peer that the following messages are encrypted. This message is part of an extra protocol and thus needs to be sent in a separate record layer.

Finished. As the final message of the handshake, the Finished message is sent. Since the message is sent after the CCS message, it is encrypted. It contains a Keyed-Hash Message Authentication Code (HMAC) over the messages of the handshake up to this point, with the master secret as HMAC secret. Both peers have to verify the HMAC they receive. This message ensures the integrity and authenticity of the complete handshake.

2.1.1.1 Extensions

For the extensibility of the TLS protocol, an extension model exists that provides the possibility to activate certain features. Extensions also have to be negotiated between both peers. The support for most of these extensions is optional. Therefore, if a server receives an extension from the client that it does not understand, it still must be possible to perform a handshake.

The specification for most of the TLS extensions are not part of the TLS 1.2 RFC. Only the signature algorithms extension is specified in the TLS 1.2 RFC. Other extensions are defined in the following RFCs.

- RFC 6066 - Transport Layer Security (TLS) Extensions: Extension Definitions [20]
  - Server Name Indication (Section 3)
  - Maximum Fragment Length Negotiation (Section 4)
  - Client Certificate URLs (Section 5)
  - Trusted CA Indication (Section 6)
  - Truncated HMAC (Section 7)
  - Certificate Status Request (Section 8)
- RFC 7366 - Encrypt-then-MAC for Transport Layer Security (TLS) and Datagram Transport Layer Security (DTLS) [21]
- RFC 7685 - A Transport Layer Security (TLS) ClientHello Padding Extension [22]
- RFC 8422 (RFC 4492) - Elliptic Curve Cryptography (ECC) Cipher Suites for Transport Layer Security (TLS) Versions 1.2 and Earlier [23, 24]
  - Supported Elliptic Curves Extension (Section 5.1.1)
  - Supported Point Formats Extension (Section 5.1.2)

Extensions that are covered by tests are explained in the following.

Signature Algorithms. The specification of TLS 1.2 requires servers to support this extension. The extension is sent by the client and contains a list of signature and hash algorithm pairs that the client supports to verify. These signatures are, for example, the signature of the server certificate or the key exchange parameters of the ServerKeyExchange message. This extension is new in TLS 1.2 and provides more flexibility for signatures. In previous versions of TLS, signatures for certificates must use the same key type as the public key of the certificate.

Server Name Indication. To be able to operate multiple domains using the same IP address, the server needs to be able to determine the domain to dispatch the messages to the correct process. Since application data is encrypted by TLS, the domain has to be transmitted unencrypted as part of the handshake to the server. The extension is sent in the ClientHello message and contains the domain name of the server.

Maximum Fragment Length Negotiation. This extension allows to limit the maximum size transmitted in a TLS record layer. This is useful for implementations that are constrained by hardware, like memory or bandwidth. The minimum size that can be specified with this extension is 512 bytes. Clients request the negotiation of this extension in the ClientHello message. If the server accepts the request of the client, it includes the extension in the ServerHello message.

Encrypt-then-MAC. If the Encrypt-then-MAC extension is negotiated between both parties, the payload of the TLS record layer is protected by the Encrypt-then-MAC procedure instead of the MAC-then-Encrypt mechanism used per default. This makes the encryption resilient against padding oracle attacks [19] and similar attacks like Lucky 13 [2] that provides a side channel to distinguish a valid MAC from invalid padding.

Padding Extension. The padding extension can be used to control the size of the ClientHello message by including a zero bytes vector of arbitrary size in the extension. This is used to work around implementation bugs of certain TLS implementations [22].

Supported Elliptic Curves. When the client supports elliptic curve cipher suites, this extension is used by the client to send a list of elliptic curves that the client supports to the server. Since every cipher suite using elliptic curves uses key agreement to negotiate the premaster secret, both peers must agree to use the same elliptic curve.

Point Formats Extension. RFC 4492 specifies multiple formats that can be used to represent a point on an elliptic curve. Besides the uncompressed format, two compressed formats exist to save bandwidth. RFC 8422 that obsoletes RFC 4492, deprecates the compressed formats, thus only the uncompressed format remains.

2.1.2 TLS 1.3

TLS 1.3 is the newest version of TLS, released in 2018 and specified in RFC 8446 [25]. In contrast to TLS 1.2, it usually only needs one RTT for its handshake to negotiate cryptographic keys. This is achieved by reordering the messages together with new extensions that enable both peers to calculate a shared secret already after receiving the hello messages. The new message flow is shown in Figure 2.3.

The handshake might not always look like this, because TLS 1.3 tries to negotiate a shared secret already with the ClientHello and ServerHello messages that basically contain the contents of a TLS 1.2 ClientKeyExchange respectively ServerKeyExchange message.

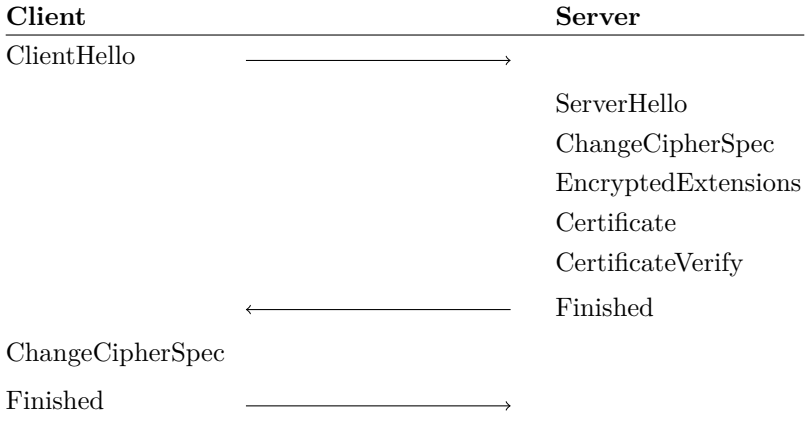


Figure 2.3: Message flow of TLS 1.3

Since the client can only guess which Diffie Hellman or elliptic curve groups the server supports, it might select a group that the server does not support. In this case, an additional round trip is required, consisting of a HelloRetryRequest sent by the server followed by a second ClientHello from the client. This is not described in detail, since it is not covered by test cases (see Section 3.4.1).

Optionally a 0-RTT handshake mode is supported, where the client already sends application data together with the ClientHello message. This is only possible if both peers have performed a regular handshake before and negotiated a pre-shared-key (PSK). This PSK is also used for session resumption. Since these features are not tested with the testsuite, they are not explained in detail.

This version of TLS is designed to be backward compatible. Since the messages are specified differently than the messages of TLS 1.2, this behavior is desired, so that TLS 1.3 clients can negotiate an older TLS version if the server does not support TLS 1.3. This is achieved by keeping the structure of the ClientHello and ServerHello messages the same and adding the new requirements of TLS 1.3 by using extensions.

In addition, TLS 1.3 only supports secure encryption, hash and signature algorithms. All TLS 1.2 cipher suites are removed from the specification. TLS 1.3 supports 5 cipher suites in total, which all provide authenticated encryption. Regarding hash algorithms, only SHA-2 algorithms are supported. In terms of signature algorithms, RSA-PKCS1 as well as ECDSA with SHA-1 are only allowed to be used as certificate signatures, which is needed for backward compatibility. For singing TLS messages, these algorithms are not allowed to be used anymore [25, page 43]. This makes the protocol resilient against Bleichenbacher's attack by design [7, 1].

ClientHello. The ClientHello message contains the same information as the ClientHello of TLS 1.2 except new mandatory extensions that are the supported versions and key share extensions that are explained later in Section 2.1.2.1. The session id only exists for backward compatibility reasons and is not used for session resumption anymore. The same applies to the protocol version which is set to the same value as in TLS 1.2, since the version negotiation for TLS 1.3 is achieved with the supported versions extension. In addition, the compression of TLS messages is deprecated, therefore, the list of supported compression methods must contain the *uncompressed* algorithm.

ServerHello. The ServerHello message is also mainly unchanged to the previous version of TLS. Like in the ClientHello message, it must contain the key share and supported versions extension. The version field is also set to TLS 1.2 and the compression method must be set to *uncompressed*. TLS 1.3 adds a new mechanism to the ServerHello that enables clients to detect a downgrade of the negotiated TLS version. If the server selects a version prior to TLS 1.3, it sets the last 8 bytes of the server random to a static value.

ChangeCipherSpec. This message is only sent in the backward compatibility mode of TLS 1.3. It serves the same purposes as the CCS message of TLS 1.2, that is that the following messages will be encrypted. Because the message is not required, TLS 1.3 implementations should ignore the message and activate the encryption based on the state machine that is part of the specification [25, Sections A.1 and A.2].

EncryptedExtensions. The EncryptedExtensions message is new in TLS 1.3 and provides a way to send extensions encrypted to the client. Because of this, the ServerHello message only contains the key share and supported version extension. Every other extension that is sent by the server and can be encrypted, is part of the EncryptedExtensions message.

Certificate. The Certificate message contains the certificate of the server. This message is mandatory if session resumption is not used, which is at least the case at the first time both peers perform a handshake.

CertificateVerify. This message provides authentication of the handshake by signing the contents of all handshake protocol messages up to this point using the private key of the server. The message is mandatory if a Certificate message was sent before.

Finished. The Finished message contains a MAC computed over every handshake message using a key derived from a shared secret.

2.1.2.1 Extensions

In contrast to TLS 1.2, multiple messages are specified to have extensions. These are the ClientHello, ServerHello, EncryptedExtensions, Certificate, CertificateRequest, NewSessionTicket and HelloRetryRequest. The RFC specifies which extension is allowed in which message. Sending an extension as part of the wrong message is not allowed, so that the receiving peer must terminate the connection [25, page 37].

Signature Algorithms. In contrast to TLS 1.2, the client is only required to send this extension for TLS 1.3 renegotiation. In addition, the client must support one algorithm that is allowed to be used to sign TLS messages. This means that the list of signature and hash algorithms must at least contain one algorithm that does not use SHA-1 as hash algorithm or RSA-PKCS1 as signature schema, since these algorithms are only allowed to create signatures for certificates.

Supported Versions. For TLS 1.3 negotiation, this extension is required. When this is sent by the client as part of the ClientHello message it contains a list of TLS versions the client supports ordered by preference. The server only considers the versions that are contained in this list to select a TLS version. The server only sends the selected TLS version in the supported versions extension as part of the ServerHello message to the client.

Supported Groups. This extension is similar to the supported elliptic curves extension from TLS 1.2, with the difference that also finite field groups are allowed in the case of TLS 1.3. The extension is required if a key share extension is sent, that is if the client wants to achieve a 1-RTT handshake.

Key Share. The key share extension contains the ephemeral public keys that are part of the ClientKeyExchange and ServerKeyExchange message in TLS 1.2. The extension can contain multiple key share entries for different groups, each consisting of a supported group, that is already part of the supported groups extension, and the public key. The server must support at least one group to be able to achieve a 1-RTT handshake. The public key of the server is sent in the key share extension contained inside the ServerHello message.

2.2 TLS-Attacker

TLS-Attacker [17] is a Java framework that allows controlling the TLS protocol on byte level. It can operate as server or client and provides APIs to specify the TLS messages and their content that are going to be sent. This results in a powerful tool that can be

used to specify a sequence of TLS messages performing a regular handshake as well as for specifying a handshake that does not conform to the specification. Being able to model the latter kind of handshakes is useful for detecting bugs in implementations by observing the behavior of the other peer. Since the test cases of the testsuite use TLS-Attacker for sending TLS messages its architecture is explained.

2.2.1 Specifying TLS message flows

Each message specified in TLS is implemented in TLS-Attacker as a separate Java class. The fields of the TLS messages are modeled with the ModifiableVariable [26] project. A `ModifiableVariable` object is a wrapper around a data type and provides additional options to modify the contents during runtime. This is achieved by attaching modifications to the object that are executed when the value of the wrapped data type is accessed using the `getValue()` function. Using different modifications that allow to replace, modify or expand the sent TLS messages, without changing the implementation of the protocol.

For the specification and execution of a sequence of TLS messages with TLS-Attacker, multiple classes are necessary. Figure 2.4 shows how the classes are used to perform a TLS handshake. The classes and their purposes are explained in the following.

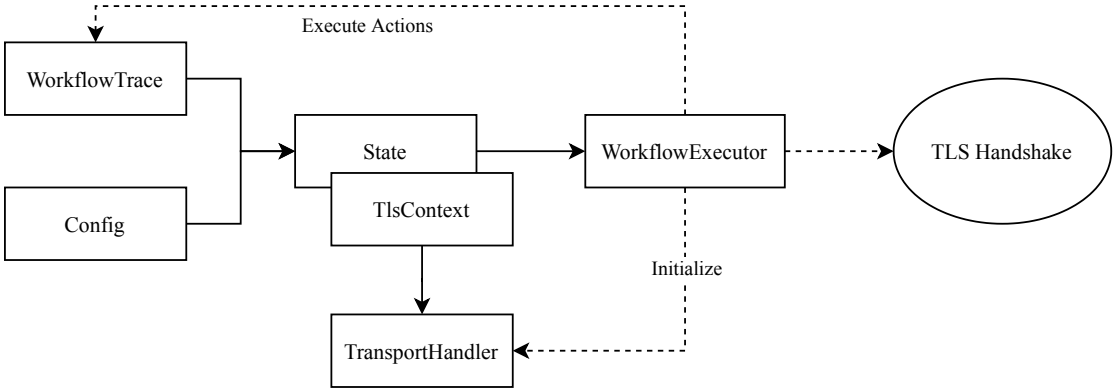


Figure 2.4: Overview of TLS-Attacker classes that are needed to perform a TLS handshake.

Before a handshake can be executed, it first has to be specified. This is achieved using the following two classes.

- `WorkflowTrace`
- `Config`

WorkflowTrace. The `WorkflowTrace` object contains a list of actions that are executed. Actions exist to send and receive TLS messages, such as the `SendAction` and `ReceiveAction`. Both of these actions are initialized with classes that inherit from the `ProtocolMessage` class and thus represent a TLS message. The `WorkflowConfigurationFactory` class provides static methods that create a workflow containing the actions to perform different types of handshakes.

Config. The `Config` object is used for the configuration of the TLS-Attacker framework. Part of the configuration specifies network settings, for example the endpoint for the communication when the framework operates as the client. Or on which interface and port it should listen for incoming connections in case it acts as a server. TLS configurations, such as which cipher suites or elliptic curve the implementation supports, are also part of the `Config`. In addition, it contains several other `Config` flags that control how messages are received and send or what should happen if there is no intersection of supported algorithms. TLS-Attacker is shipped with a default config, that is stored in a XML file `default_config.xml` and can be automatically loaded, using the static `createConfig` function of the `Config` class.

2.2.2 Executing TLS message flows

For the execution of `WorkflowTrace` and `Config` objects, the following classes are important to understand.

- `State`
- `TlsContext`
- `WorkflowExecutor`

State. The `State` object bundles a `WorkflowTrace` object together with a `Config` object and thus forms a well-defined plan how and which messages should be exchanged with another peer. It is also responsible for normalizing the `WorkflowTrace`, that is setting up the configuration for the network connections using the settings from the `Config` if this information is not already attached to the workflow. In addition, it creates a `TlsContext` object for every connection and holds a reference to this context.

TlsContext. The `TlsContext` object is a data store that is used during the TLS handshake. It stores the negotiated parameters like the TLS version or cipher suite, as well as cryptographic keys like the premaster secret.

WorkflowExecutor. The `WorkflowExecutor` is an abstract class that is used to execute a `State` object. A often used concrete subclass is the `DefaultWorkflowExecutor` class initialized with the `State` specifying a handshake. The execution of the `State` is done by executing each action of the `WorkflowTrace`. These actions take care of sending and receiving TLS messages. Depending on the `Config`, the executor also initiates or terminates the network connection. The connection, TCP or UDP, is handled by the `TransportHandler` class that provides methods to send and receive messages using a socket.

Executing a `State` involves sending and receiving TLS messages. As outlined above, each TLS message is implemented as a Java class that inherits from the `ProtocolMessage` class. Sending and receiving a message involves (de)serialization of this class. This process is explained below. In addition, the `TlsContext` object constantly needs to be updated, to keep track of negotiated parameters and keys.

2.2.2.1 Sending TLS messages

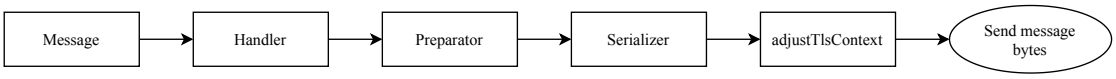
Before a TLS message can be sent, it needs to be prepared and serialized. For every TLS protocol message class, a `handler` class exists inheriting from `ProtocolMessageHandler` that provides a `prepareMessage` function. This function takes an instance of a message object, coordinates the preparation and serialization using the `Preparator` respectively `Serializer` classes and returns a byte array. Finally, the `TlsContext` is updated by the `adjustTlsContext` function of the `handler`. The data flow of the message is shown in Figure 2.5a.

Preparator. The `Preparator` classes preform the preparation of the messages. This means that the `Preparator` sets the fields of a message as specified in the RFC, considering the values specified in the `Config` and `TlsContext` object. A mechanism like this is required to perform TLS handshakes since it is not known before connecting to the peer, which algorithms it supports. Therefore, the exact content of the messages is not known when the `WorkflowTrace` is created. However, modifications that are attached to the `ModifiableVariable` fields of the messages are not overwritten by the `Preparator` and executed during serialization.

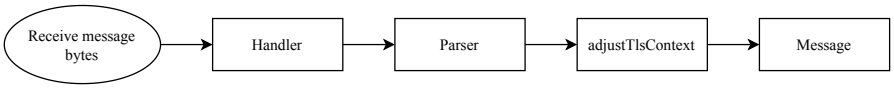
Serializer. The `Serializer` classes take a message as input and serialize the message content according to the specification into a byte array. To access the values of the messages, the *getValue* function of the `ModifiableVariable` fields is used. Therefore the value created by the `Preparator` is used as foundation modified by the modifications attached to the field. The resulting byte array is sent over the network using the TCP- or UDP protocol.

2.2.2.2 Receiving TLS messages

Receiving a TLS message is the transformation from a byte array to the message class (see Figure 2.5b). This operation is also controlled by the `Handler` class, specifically the `parseMessage` function. This function uses the `Parser` class of the corresponding message to deserialize the byte array into an message object. After that, the `adjustTlsContext` function is called that updates the `TlsContext` based on the received information.



(a) Sending a message.



(b) Receiving a message.

Figure 2.5: Processing of a sending/receiving TLS messages with TLS-Attacker classes.

2.3 Docker

Docker is a tool for container virtualization that are lightweight virtual machines. In contrast to virtual machines, Docker containers use the same kernel as the operating system and therefore can omit the hypervisor that emulates the computer. This makes Docker containers more efficient than virtual machines [27].

In 2015, Docker founded the Open Container Initiative (OCI) [28] to standardize and open-source how containers are executed and built. Since then, other tools like Podman [29] were created that are able to run Docker containers as well. Therefore, Docker is nowadays only a command-line tool for the administration of the container ecosystem. Since this tool is used for the evaluation, specific Docker terms are explained in the following.

Container. A Docker container is a running operating system or program that is instantiated from a Docker image. Starting a Docker container is done using the `docker run` command-line tool [30]. All changes an application makes during the runtime inside a container only affect the container and never the underlying image. Such changes are, for example, the creation of new files. Thus, they are vanished when the container is removed. To persist data outside the container, it is possible to mount a local folder inside the container or using a Docker volume. A Docker volume is an abstraction of persistent storage. In its most simple form it is a folder on the hard drive, but also can be a folder shared over network [31].

Image. A Docker image contains a file system containing all the software installed during the build process. Additionally, it contains instructions for the runtime how to instantiate a container from this image. That is, for example, the program that is executed when the container is started. This information is specified inside a Dockerfile. Docker images are usually referenced by tags. That is a name and version number of the image.

Dockerfile. The Dockerfile is a text file with instructions how to build a Docker image with the `docker build` command-line tool [32]. Each Dockerfile first inherits from an existing image, the base image, and extends this image with additional software. This can be additional packages that are installed from a Linux repository or downloaded from the internet, as well as source code that is copied from the local file system to the image. The benefit of having a Dockerfile is that it produces a reproducible docker image independent of the host operating system.

Docker Network. By default, all docker containers run inside a private network that uses network address translation (NAT). Therefore, the containers have access to the internet, but they are not accessible from the host without port forwarding. Inside the network, Docker provides DNS services, so that each container can connect to another container in the same network using the name or identifier of a container as hostname. The `docker run` command-line tool provides the parameter `-network` that specifies inside which network a container should run. Putting multiple containers into multiple networks isolates them on the network layer from each other.

2.4 TLS-Docker-Library

The TLS-Docker-Library [33] is a project created in 2017 by Ebert [18] with the goal to provide a collection of TLS implementations as Docker images. The repository contains Dockerfiles for different implementations in multiple versions, which create Docker images that execute as server or client. Therefore, this project is a valuable resource for the analysis of how TLS implementations evolved across multiple versions.

Besides the Dockerfiles, the project also offers a Java library that provides an abstraction layer to start these Docker containers using a unified API that works for every implementation. This is necessary since each server or client implementation needs different command-line arguments to start successfully. These are, for example, the port on which the server listens or the server to which a client implementation should connect to.

Starting a TLS Docker container using the library is shown in Listing 1. This example starts an OpenSSL server in version 1.1.1g listening on port 4343.

```
1 DockerTlsManagerFactory manager = new DockerTlsManagerFactory();
2 TlsInstance server =
  ↪ manager.getTlsServer(TlsImplementationType.OPENSSL, "1.1.1g",
  ↪ 4343);
3 server.start();
```

Listing 1: Starting an OpenSSL server using the Java library.

The command-line arguments for each implementation are specified in `.profile` files that are XML files and part of the `resources` folder of the project. The content of these files are serialized objects of the `ParameterProfile` class. Each profile specifies for which implementation and role (server/client) it is suited, as well as the parameters that are needed for the launch of the implementation. Using the library to start a TLS Docker container triggers the evaluation of the parameters inside the `.profile` file. The resulting string is used as command-line argument for the Docker container.

The library transforms the code from Listing 1 into the `docker run` command in Listing 2. When both listings are compared it is notable that the Java library also handles the network configuration that makes the server container reachable at `127.0.0.42:4343` that is loop-back address. Additionally, the Docker volume `cert-data` is mounted by the container that contains the server certificates. These are generated during the setup process of the project, that is explained in the readme file of the project repository [33].

```
docker run -v cert-data:/cert/ openssl-server:1.1.1g -port 4343
↪ -cert /cert/rsa2048cert.pem -key /cert/rsa2048key.pem -p
↪ 127.0.0.42:4343:4433
```

Listing 2: `docker run` command equivalent to the code from Listing 1.

2.5 Software testing

Software testing is used to find failures in software. These are, for example, bugs that are the result of implementation failures. The goal is to find these failures by writing more source code in the form of tests. Tests ideally execute the software with every possible input to ensure that the software works as expected.

Writing software tests can be done on multiple levels. It is, for example, possible to only test a single function of a program. This kind of test is known as unit tests. If the software is more complex and consists of multiple modules created by multiple developers, there

might also arise problems when these modules interact with each other. Tests that analyze the functionality on this level are known as integration tests.

In the sense of this thesis, where TLS implementations are tested, the whole system is under test and observed as a black box. In this case implementation failures are, when the software receives an input, that is a (sequence) of TLS messages, and reacts differently than specified in the specification of the TLS protocol. Thus, the developed test cases test even more than integration tests and are therefore known as system test.

2.5.1 JUnit 5

JUnit 5 is used as a testing framework to write the tests included in the testsuite. It is the successor of JUnit 4 which is the most popular testing framework for Java according to the Maven Repository usage numbers [34]. Version 5 was released in 2017 and introduced an extension system that allows extending the functionality of JUnit.

Tests in JUnit can obtain three states after execution. Disabled, failed and succeeded. Disabled tests are not executed by the framework, since specified conditions are not fulfilled. A test is considered as failed if it throws an exception during execution. If neither of them has occurred, the test is succeeded successfully.

The testsuite uses three different extension types to extend JUnit. JUnit provides an interface for each extension type that can be implemented by classes. If the extension class is registered to JUnit, the appropriate interface functions are called at runtime by the JUnit framework.

TestWatcher. This extension interface has functions that are called whenever a test terminates independently of the test result.

ExecutionCondition. Before a test is started, the function provided by the `ExecutionCondition` interface is called that returns an object indicating if the test case should be executed or not.

ParameterResolver. Test class constructors or test methods are allowed to take a parameter. If such a parameter is specified, a class implementing this extension interface is responsible for resolving the parameter. That is creating the object that is passed as the parameter to the function.

3 Implementation

This chapter describes the implementation of the TLS testsuite [35], the test framework [36] and the setup that is used for the evaluation. The testsuite depends on the test framework that provides the business logic for the test derivation and execution, and is based on JUnit 5.

Finally, the report analyzer is introduced. This web application visualizes the reports created by the testsuite and provides filter capabilities to quickly find interesting results.

3.1 Design

The implementation of the testsuite and the framework aims for modeling tight tests, which are easy to understand and do not require much code. Section 3.3.6 describes how test cases are modeled in detail. To make the tests tight, the target has to be forced to negotiate certain TLS parameters, such as a specific cipher suite. This approach allows observing whether the target reacts differently depending on negotiated TLS parameters.

For example, a test case sending a Finished message with invalid content should always succeed in the sense that the handshake is aborted by the receiving peer. However, bugs in TLS implementations may lead to the completion of the handshake for a specific cipher suite, even if the Finished message is invalid. These kinds of implementation flaws can only be detected if multiple handshakes are performed. To perform this test, a `WorkflowTrace` and `Config` object are created first that perform a handshake including sending an invalid Finished message. The framework derives multiple `WorkflowTrace` and `Config` objects from this, each sending only a single cipher suite in the ClientHello or selecting a different cipher suite in the ServerHello message. The framework ensures that only cipher suites are used that the server or client supports.

The general idea is to keep the test case that models a TLS handshake generic. Based on the configuration of the framework, it derives multiple handshake configurations from the one provided, each forcing the target to negotiate different TLS parameters. This concept is illustrated in Figure 3.1. A test case described by a single TLS handshake is therefore expanded to a test case with multiple handshakes. Each derived handshake can be seen as a distinct test case and thus has its own test result.

After the handshakes are created, the collection of handshakes is executed, which results in sending and receiving TLS messages to the target client or server. Finally, the test result needs to be determined. The framework provides APIs to model a validation process and examine the received TLS messages.

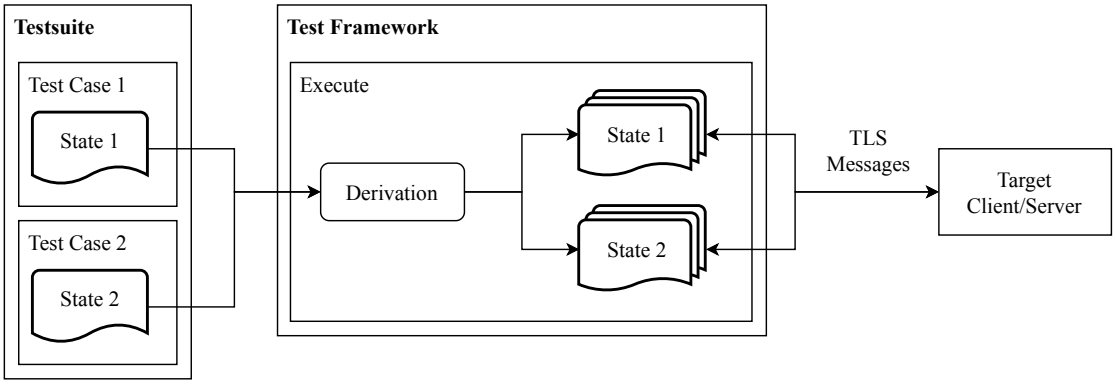


Figure 3.1: Overview of the testsuite architecture.

Another design goal for the framework is to provide more flexibility than the testsuite built by Ebert [18]. Four major points are a limitation of the existing architecture, which the newly developed testsuite and test framework aim to improve.

Target Configurability. The existing testsuite is mainly designed test implementations running in Docker containers and is therefore difficult to use for testing arbitrary targets. In contrast, the test framework introduced in this thesis, provides a command-line interface that allows configuring the test target.

Client Tests. The developed testsuite should be able to perform client tests. This feature is not supported by the existing testsuite. The new test framework implements features that are needed for testing clients. This includes a feature that triggers a client to connect to the testsuite so that automatic testing can be performed.

Message Validation. A testsuite has to check the content of the received messages to determine whether a test completed successfully. The existing testsuite does this by populating the `ModifiableVariable` fields of the messages that are part of the `ReceiveAction`. After the handshake is executed it checks if the received value matches the content assigned to the variable. This approach is not sufficiently flexible, since the expected values like selected algorithms are not always known beforehand. Using this setup it is not possible, for example, to set an allow- or deny list for received values. Test cases using the new framework specify Java lambda functions for the validation. Therefore, complex validation setups are possible.

Debugability. The existing testsuite does not provide the possibility to execute single test cases without changing the source code. This is especially important during the development of the test cases to be able to determine quickly if the test case works as expected. Using JUnit as testing framework for the developed testsuite provides GUI integration for many IDEs with the possibility to execute individual test cases.

3.2 Test Result

Unit tests can obtain two possible test results: Success or failure. These two result values are not sufficient to describe the test result of a TLS handshake, especially when taking into account that a single test case can perform multiple handshakes with different configurations as explained in Section 3.1. This means that each handshake can be seen as a single unit test. The overall test result therefore depends on the results of each performed handshake.

Assuming a test case expecting an alert message sent by the target in response to a message violating the specification. For each handshake, derived from the test case, the following four outcomes can occur. These can be modeled using three different result values as detailed below.

1. The expected alert message is equal to the received alert message. In this case, the target passes the test, so the result for the handshake is set to `SUCCEEDED`.
2. The target does not send an alert at all and does not close the connection. Since this does not conform to the specification, the result is set to `FAILED`.
3. The expected alert is different from the received alert. This behavior does not conform to the specification. But if both alert messages have the same severity level, this behavior is acceptable in most cases. However, to indicate that the target reacts differently than described by the specification, the result is set to `PARTIALLY_SUCCEEDED`.
4. Instead of sending an alert, the target sends a TCP FIN packet and therefore closes the connection. If a test case expects a fatal alert and receives such a packet, the result for the handshake is also set to `PARTIALLY_SUCCEEDED`, since the handshake is terminated.

Depending on the results of the individual handshakes, the result for the complete test case is determined. Table 3.2 shows the possible results for a test case depending on the result of two handshakes it performs. Besides the mentioned possible result values, the test case can obtain a new result value, that is `PARTIALLY_FAILED`. The test case obtains this value when at least one handshake `SUCCEEDED` and one handshake `FAILED`. A single handshake should not obtain this value, since there is no sensible condition when this value could be assigned to the handshake.

Handshake A	✗	✗	✓	✓
Handshake B	✗	✓/✓	✓	✓
Test Result	✗	✗	✓	✓

✗ `FAILED`
✗ `PARTIALLY_FAILED`
✓ `PARTIALLY_SUCCEEDED`
✓ `SUCCEEDED`

Table 3.2: Possible test results depending on the results of two handshakes.

3.3 Test Framework

The test framework uses JUnit as a dependency and includes JUnit extensions, annotations and superclasses which are used by the test cases of the testsuite. Therefore, it contains all components necessary to model tests for the TLS protocol.

Using JUnit results in multiple benefits, for example, aspects like test scheduling and execution are handled by the unit testing framework. In addition, most IDEs support JUnit tests by offering a graphical user interface (GUI) to execute the tests and view the results.

3.3.1 Command-Line Interface

The command-line interface provides the possibility to configure the framework without changing the implementation. Thus, this is the key feature that allows the tool to be used by users with no background knowledge on the implementation details. The classes modeling the command-line interface are part of the `config` package of the framework.

The interface is structured in the following way. The first argument group contains options that apply to both, client and server tests. These include options like the location where the test report should be saved or how many TLS handshakes should be executed in parallel. A more special option is `-timeoutActionScript`. It takes any shell command-line as an argument. The given command is executed when the last TLS handshake was performed more than 20 seconds ago. It is intended to be used to restart a target in case it gets stuck.

The general options are followed by the command argument describing the role of the target (`server` or `client`).At last, command-specific arguments can be appended, which are used to configure the framework explicitly for client or server tests.

Server Tests. For server tests, hostname or IP address and the listening port need to be provided. This is implemented using the TLS-Attacker `ClientDelegate` class. Therefore, passing the `-connect` argument followed by the host and port configures the framework. For example: `-connect localhost:443`.

Client Tests. For client tests, more arguments have to be provided, more precisely the trigger for the client to initiate a new connection to the server running the testsuite. This is the task of the `-triggerScript` option that followed by an executable and its arguments. The specified executable is executed before every handshake, so that the client establishes a connection to the testsuite. In addition to that, the `-port` option is required. It specifies the port, the testsuite is listening for incoming connections.

3.3.2 Conditional Test Execution

Since TLS is a complex protocol with many parameters that are negotiated during the handshake between the server and the client, it is not sensible to execute every test case against every target. For example, test cases testing the implementation of a TLS extension, assume that the target supports this extension. One solution would be to execute the test, check that the extension is supported by the target, and terminate the test successfully if it does not. However, this solution would bias the overall result, as it is not possible to check whether the test terminated successfully or the extension was not supported by the target later on. To be able to differentiate between these two cases, the test should not be executed at all, when the requirement for the test is not fulfilled.

To achieve the desired effect, JUnit extensions are used. The extensions implemented in the test framework check if a test method is marked with a specific annotation modeling the condition. The following annotations for conditional test execution are supported.

- `TestEndpoint`
- `TlsVersion`
- `KeyExchange`
- `MethodCondition`

TestEndpoint

The `TestEndpoint` annotation controls whether a test method is executed for client or server tests. Since this annotation takes an argument of the `TestEndpointType` enum, the two meta-annotations `ServerTest` and `ClientTest` make this annotation more convenient to use. These annotations can be used on classes, methods or both. In

the latter case the method annotation has a higher priority. If the annotation is not specified at all, the test executes always.

Example usage:
```
@TestEndpoint(endpoint = TestEndpointType.CLIENT)
```

TlsVersion

If the `TlsVersion` annotation is present, the test is only executed, when the target supports a specific TLS version. Since each version of TLS is specified in a different RFC, it is only possible to specify one version at a time. Each test method or class is required to be annotated with this annotation. If the annotation is specified at class and method level, the method annotation takes priority. Since the specified version is represented by the `ProtocolVersion` enum provided by TLS-Attacker, every TLS version included in TLS-Attacker works with this extension.

To make the usage more convenient, the test framework provides the two already annotated classes `Tls12Test` and `Tls13Test`. If a class containing test methods inherits from one of those two classes, it is not necessary to specify this annotation again.

Example usage:
```
@TlsVersion(supported = ProtocolVersion.TLS12)
```

KeyExchange

A test method annotated with KeyExchange annotation is only executed, if the cipher suites supported by the target, use a specified key exchange mode. These modes are `RSA`, `DH` and `ECDH`. In addition, two special modes are available `ALL12` and `ALL13`, which result in the execution of the test for all TLS 1.2 or TLS 1.3 key exchange modes and cipher suites, respectively. The following holds true.

$$ALL12 \iff \{RSA, DH, ECDH\}$$
$$ALL12 \cap ALL13 = \emptyset$$

Although TLS 1.3 supports multiple types of groups for a key exchange, the specification of the TLS 1.3 is more generic and independently from the used group. Therefore, the only valid key exchange mode for TLS 1.3 is `ALL13`.

The annotation takes three arguments.

- `KeyExchangeType[] supported`
  An array of the key exchange modes.

- `boolean mergeSupportedWithClassSupported` (optional, default `false`)
  If the annotation is used at class and method level, setting this option to true, merges the `supported` array from both annotations.
- `boolean requiresServerKeyExchMsg` (optional, default `false`)
  Setting this option to `true`, executes the test only with cipher suites that requires a server key exchange message.

The annotation can be specified at class or method level. Not specifying this annotation results in an implicit annotation with supported set to `{ALL12, ALL13}`.

Example usage:
`@KeyExchange(supported = {KeyExchangeType.RSA, KeyExchangeType.DH})`

MethodCondition

This annotation provides a generic way to disable a test case. It provides two parameters that specify a class and a method that is executed before the test. The class parameter is only needed if the method belongs to a different class than the currently executed test method. Since parameters provided to annotations must be constant expressions, it is only possible to specify the condition method as a string. The JUnit extension uses reflection to call the specified method.

The condition method can optionally receive a single `ExtensionContext` object as an argument. The extension context is provided by JUnit and contains information about the current test execution. The return type must be an instance of the `ConditionEvaluationResult` class. This class provides the two static methods `.disabled()` and `.enabled()` as initializers.

```
1   @ExtendWith({MethodCondition.class})
2   class TestClass {
3       public ConditionEvaluationResult condition() {
4           return ConditionEvaluationResult.enabled("");
5       }
6
7       @Test
8       @MethodCondition(method = "condition")
9       public void test() {
10          System.out.println("This test is executed");
11      }
12  }
```

Listing 3: Example usage of the `MethodCondition`.

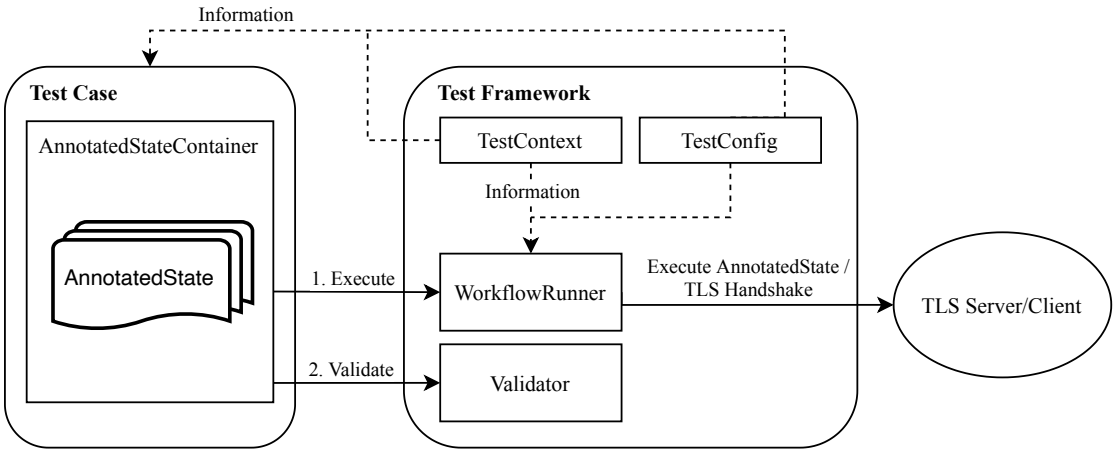


Figure 3.3: Interaction between the most important classes of the test framework.

It is possible to use this annotation on class and method level. If it is specified at both levels, the test is only executed, if both methods return an enabled `ConditionEvaluationResult`. An example usage for this annotation can be found in Listing 3.

3.3.3 Architecture

To understand the test framework and later on the test cases that are part of the testsuite, the most important classes and their tasks are described in this subsection. Figure 3.3 visualizes the interaction between those classes that are part of the framework and help to understand the following sections.

- `TestConfig`
- `TestContext`
- `TestRunner`
- `AnnotatedState`
- `AnnotatedStateContainer`
- `WorkflowRunner`
- `Validator`

TestConfig

The `TestConfig` class extends the TLS-Attacker `TLSDelegateConfig` class and therefore follows the convention established by other projects based on TLS-Attacker. Thus, it stores the configuration that is determined by the command-line arguments passed to the program.

In order to make use of existing code and business logic, the class delegates the parameter resolution to the existing TLS-Attacker classes `ServerDelegate` and `ClientDelegate`. Which one is used, depends on the type of the target that is tested.

TestContext

This class is a singleton class. It contains references to the following objects that have to be made available from the test methods and the JUnit extensions.

- `TestConfig`
- `TestSiteReport`, is a serializable subclass of the `SiteReport` class provided by the TLS-Scanner. It stores TLS properties supported by the target, for example, supported TLS versions, cipher suites, elliptic curves, etc.
- `ParallelExecutor`, that is a `ThreadPoolExecutor` used to perform multiple TLS handshakes in parallel.

In addition, the class manages the test results by storing them inside a map, referencing each result with a unique identifier generated by JUnit.

The class exists to expose the mentioned objects as well as the result management APIs to each test method and the JUnit extensions. This is only possible by utilizing a singleton object since JUnit does not provide an API to inject objects into the test runtime environment, which are created before the tests are started.

Making this class a singleton makes it impossible to execute the testsuite against multiple targets in parallel because only one `TestContext` object can exist during runtime. If it is necessary to test multiple targets in parallel, multiple instances of the testsuite have to be started.

TestRunner

The `TestRunner` class is executed first and is responsible for setting up the framework. First it waits for the target to be available, starts the preparation phase that discovers which TLS properties the target supports, and after that the testsuite execution phase is started (see Section 3.3.5).

AnnotatedState

This class contains a reference to a TLS-Attacker `State` object and annotates it with additional information. Therefore, an `AnnotatedState` object represents a single TLS handshake. The additional information are about the connection, like the source and destination ports, as well as a start and end timestamps of the handshake. It also stores the value of the result described in Section 3.2 that is part of the `TestStatus` enum.

For each `AnnotatedState` a unique identifier is generated by hashing the class- and method name of the test case it belongs to, as well as information describing the test case that is independent of the result. Therefore, each object is associated with a deterministic unique identifier. This results in the possibility to compare the result of each TLS handshake between multiple executions of the testsuite against the same or different targets.

AnnotatedStateContainer

Each test case that performs a handshake is associated with an `AnnotatedStateContainer`. This class contains a list of `AnnotatedState` objects, each of them describing a handshake using the `State` class of TLS-Attacker. It also contains additional information about the test result, for example, the test result value, how many handshakes failed, and information about the test method itself.

Furthermore, a container exposes a `validate` function that takes a lambda function consuming an `AnnotatedState` as an argument. The validation process iterates over the list of `AnnotatedState` objects and executes the provided lambda function for each state. The execution of this function determines the result of the executed handshake and therefore has an impact on the result of the complete test case, respectively the result of the `AnnotatedStateContainer`.

The lambda function provides the result for a single handshake in two possible ways. If it throws an exception or error using, for example, the `assert...` functions provided by JUnit to validate values of received messages, the framework automatically sets the result of the handshake to `FAILED`. If the function completes successfully the result is automatically set to `SUCCEEDED`. As explained in Section 3.2, a handshake can also obtain the test result value `PARTIALLY_SUCCEEDED`. This value must be set manually by the lambda function using the `setStatus` function of the `AnnotatedState` object passed to the lambda function.

The overall test result is set by the framework automatically, depending on the result of each handshake.

WorkflowRunner

An instance of this class is injected into each test method using a `ParameterResolver` JUnit extension interface, implemented by the `WorkflowRunnerResolver` class. This object is responsible for executing TLS handshakes, more specifically TLS-Attacker `State` objects, that are assembled in the test methods. In addition, it provides configuration options and the business logic for the test derivation (Section 3.3.4).

Test methods usually configure the derivation at first using public configuration properties that are explained later on, and finally call the `execute` function of this class. This overloaded function takes a `WorkflowTrace` or `AnnotatedStateContainer` object, performs the derivation and dispatches the resulting list of `State` objects to the shared `ParallelExecutor` of the `TestContext`.

In addition, this class provides a set of `generateWorkflowTrace` functions to generate prepared `WorkflowTrace` objects using the `WorkflowConfigurationFactory` class provided by TLS-Attacker. Although the return value of these functions is a `WorkflowTrace` object, it is important to note that this trace is empty, since the exact configuration of the handshake can only be determined during the derivation phase. To be still able to modify the prepared workflow traces, the `WorkflowRunner` object provides a `stateModifier` function field. A function assigned to this field modifies each `AnnotatedState` object obtained by the derivation. It can either return a new `AnnotatedState` object to replace the existing one or `null`, to use the modified `AnnotatedState`.

Validator

The `Validator` class provides static functions that are intended to be used as lambda functions that are passed to the `validate` function of the `AnnotatedStateContainer`.

Since many test cases require the same checks to validate the performed handshakes, this class reduces the amount of duplicated code across test cases. A common check is, for example, if the target responded with a fatal alert.

3.3.4 Test Derivation

Test derivation is the process the framework performs to automatically generate new `State` objects respectively handshake configurations based on a provided `State`. This aims to force the target to negotiate different TLS parameters and to apply fragmentation to the sent packets.

The framework only provides automated derivations modes for changing the negotiated cipher suite and for packet fragmentation. The derivation is optional and configured using options exposed by the `WorkflowRunner` object that is passed as an argument into each test method and used for the handshake execution. Instead of using the automated

derivation process, it is also possible to create multiple handshakes using for-loops or other control structures inside the test method.

The derivation process is visualized in Figure 3.4. A `WorkflowTrace` specifying the handshake is assembled in the test method and executed using the `execute` function of the `WorkflowRunner` object. Executing a `WorkflowTrace` also runs the `prepare` function that performs the derivation in order to force the target to negotiate a specific cipher suite. The function returns an `AnnotatedStateContainer` containing all `State` objects that are the result of the derivation process. The container is passed to the `execute` function that performs the fragmentation derivation and executes the `State` objects contained in the container.

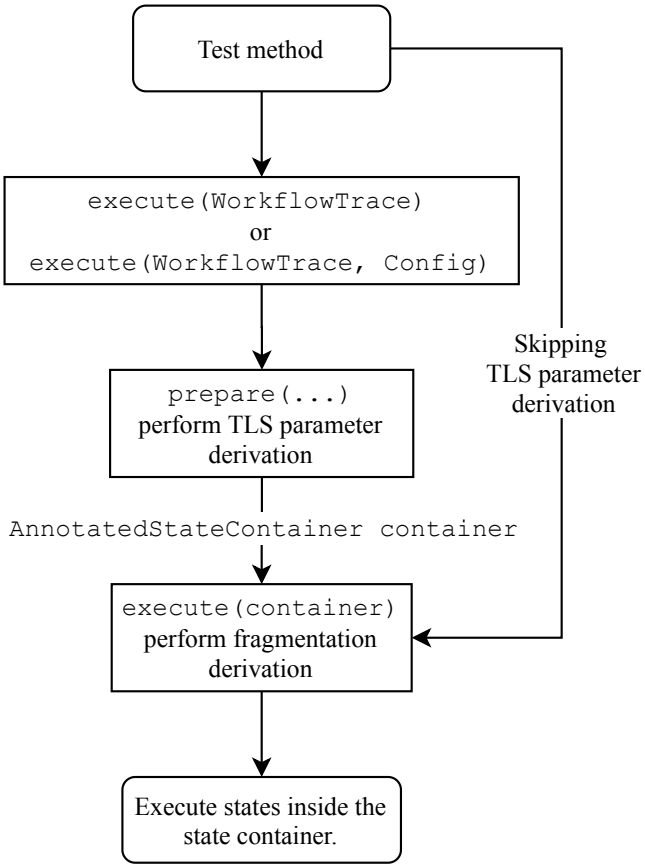


Figure 3.4: Derivation process of `State` objects.

If the test case needs to generate multiple `WorkflowTrace` or `Config` objects with different configurations, the test method triggers the derivation process manually (`prepare` function), merges multiple `AnnotatedStateContainer` objects into a single container and finally calls the `execute` function with the final container as an argument.

The test framework supports three kinds of automated test case derivations.

- Negotiated Cipher Suite
- TLS record fragmentation
- TCP packet fragmentation

3.3.4.1 Negotiated Cipher Suite

As it is determined during the preparation phase which cipher suites a target supports, the test framework can use this information during the derivation process to create new handshake configurations that replace the supported and selected cipher suites that TLS-Attacker uses. This is done by altering the `Config` object.

In addition, the business logic of this derivation considers the key exchange modes that are specified by the appropriate annotation (Section 3.3.2). If the test method is only suited for an RSA key exchange, the derivation only considers RSA cipher suites.

How the cipher suite configuration is changed can be controlled using following four different configuration flags, that are `public boolean` fields of the `WorkflowRunner` object.

replaceSupportedCiphersuites. This option only affects server tests. When it is set to `true`, the framework generates a `State` and sets the `defaultClientSupported-Ciphersuites` list of the TLS-Attacker config to a single cipher suite. Only cipher suites are considered are supported by the server and that are a valid regarding the `KeyExchange` annotation that is assigned to the test method.

appendEachSupportedCiphersuiteToClientSupported. This option only affects server tests as well. Setting this option to `true` causes each cipher suite supported by the server and test method to be appended to the `defaultClientSupportedCiphersuites` list. Only cipher suites that are not already part of the list are appended.

replaceSelectedCiphersuite. This option only affects client tests. Setting this option to `true` results in the generation of a `State` object for every cipher suite the client and the test method supports. This sets the config options `defaultServerSupported-Ciphersuites` and `defaultSelectedCiphersuite` to the same value, that is a single cipher suite.

respectConfigSupportedCiphersuites. This option affects client and server tests and only works in combination with one of the `replace...` configuration options. When this option is set to `true`, the framework sets the supported and selected cipher suite only to a cipher suite that is included in the default client or server supported cipher suite list and is supported by the target.

3.3.4.2 Fragmentation

In addition to the derivation based on changing the cipher suite, the framework also derives new handshake configurations by applying fragmentation. The two available fragmentation modes are based on TLS record fragmentation and TCP packet fragmentation. By default, this kind of derivation is enabled, since it is independent of negotiated or supported TLS parameters.

The derivation is performed right before the handshakes are executed in the `execute(AnnotatedStateContainer)` function. Therefore, if both fragmentation modes are enabled, the amount of states inside the container is tripled. As for derivations based on cipher suites, the configuration is controlled by using `public boolean` fields of the `WorkflowRunner` class.

useRecordFragmentationDerivation. When this option is set to `true`, for every state inside the container passed to the execute function, a new state is generated, where the `DefaultMaxRecordData` of the `Config` object is set to 50. That means that every TLS record larger than 50 bytes is split into multiple records, each with a size of 50 bytes. Setting this to a deterministic static value ensures the comparability between the test cases of multiple testsuite executions. In addition, 50 bytes is a small enough value to ensure that records are actually fragmented.

useTCPFragmentationDerivation. When this option is set to `true`, the transport handler is changed to `TCP_FRAGMENTATION`. This transport handler splits sending messages into three chunks of equal size and flushes the output stream after each chunk.

3.3.5 Testsuite Execution

To start the execution of the testsuite, four lines of code are necessary (Listing 4), excluding exception handling. At first a new `TestContext` object is created (line 3), after that, the testsuite sets the TLS versions it supports (line 4) and triggers the parsing of the command-line arguments (line 8). Finally, the `runTests` method of the `TestRunner` instance is executed (line 9).

Calling the `runTests` function, causes the execution of the preparation and execution phase.

```java
public class Main {
    public static void main(String[] args) {
        TestContext testContext = new TestContext();
        testContext.getConfig().setSupportedVersions(
            ProtocolVersion.TLS12,
            ProtocolVersion.TLS13
        );
        testContext.getConfig().parse(args);
        testContext.getTestRunner().runTests(Main.class);
    }
}
```

Listing 4: Code used by the testsuite to setup the test framework and start the tests.

Preparation Phase

During the preparation phase, the framework executes the TLS-Scanner for server tests. The TLS-Scanner performs multiple handshakes to analyze the server including configuration checks such as such supported cipher suites, extensions, elliptic curves and TLS versions. The results are stored using the `TestSiteReport` class. This class provides a static method `fromSiteReport` that only extracts necessary properties from a `SiteReport` object provided by the TLS-Scanner, with the goal to obtain a serializable object.

The `TestSiteReport` object is part of the `TestContext` singleton object and is used to evaluate the conditional test execution extensions. For the preparation phase, the scanner does not scan for known vulnerabilities to increase the speed, especially for cases where servers can only be scanned single-threaded.

Because the TLS-Scanner only supports scanning servers, it can not be used for the preparation phase for client tests. The test framework implements its own minimalistic Scanner instead. In theory, the client includes all the algorithms, cipher suites, TLS versions and extensions it supports in the ClientHello message. However, this might not be the case for all implementations, since implementation failures can lead to the problem, that a client does not properly advertise every algorithm it supports. The minimal client scanner compensates this problem at least for the cipher suites. It iterates through all cipher suites that TLS-Attacker supports and tries to perform a handshake. If the handshake is successful, the cipher suite is considered to be supported by the client. Other information like supported versions, elliptic curves, signature and hash algorithms are extracted from the ClientHello message under the assumption, that the client only supports the advertised parameters. All of the parameters are also stored inside the `TestSiteReport` class.

To improve subsequent executions of the test suite against the same target, the `TestSiteReport` is saved to a file acting as a cache. Whenever the caching file exists, it is deserialized into a `TestSiteReport` and the preparation phase is skipped. Using the `-ignoreCache` option, the cache can be bypassed.

Testsuite Execution Phase

The execution phase starts right after the end of the preparation phase. In this phase JUnit is instructed to scan for all test methods included in the Java package to which the class object, provided to the `runTests` function, belongs to (Listing 4, line 9). Finally, the discovered test methods are executed.

3.3.6 Modeling a Test Case

A test case is a Java function that should be at least annotated with JUnit's `@Test` annotation or the provided `@TlsTest` annotation to be discoverable by the JUnit test engine. The function can take an optional `WorkflowRunner` object as an argument that provides APIs to perform TLS handshakes. The `@TlsTest` annotation takes up to three arguments that describe the test case.

- `String description`
- `SeverityLevel securitySeverity`
  (optional, default `INFORMATIONAL`)
- `SeverityLevel interoperabilitySeverity`
  (optional, default `INFORMATIONAL`)

Description. The description string describes the test case in order to provide an overview of the functionality that is addressed by this test.

SecuritySeverity. The `securitySeverity` describes the impact on the security of the target in case the test fails.

InteroperabilitySeverity. Similar to the `securitySeverity` this parameter specifies the impact on the interoperability of the target with other TLS servers or clients in case the test fails.

The severity level is modeled by the `SeverityLevel` enum that has five levels in total. The higher the level, the higher the is the possible impact on the security or interoperability. The levels are:

- `SeverityLevel.INFORMATIONAL`

- `SeverityLevel.LOW`
- `SeverityLevel.MEDIUM`
- `SeverityLevel.HIGH`
- `SeverityLevel.CRITICAL`

Each test case performing a TLS handshake, roughly consists of three sections. Listing 5 shows a simple example of a server test that sends an unknown cipher suite in the ClientHello message and checks if the server sent a ServerHelloDone message.

```
1  @RFC(number = 5246, section = "7.4.1.2. Client Hello")
2  @ServerTest
3  public class ClientHello extends Tls12Test {
4
5    @TlsTest(description = "If the list contains cipher suites the server
     ↪  does not recognize, support, or wish to use, the server MUST
     ↪  ignore those cipher suites, and process the remaining ones as
     ↪  usual.", interoperabilitySeverity = SeverityLevel.CRITICAL)
6    public void unknownCipherSuite(WorkflowRunner runner) {
7      runner.replaceSupportedCiphersuites = true;
8
9      Config c = this.getConfig();
10
11     ClientHelloMessage clientHelloMessage = new ClientHelloMessage(c);
12     clientHelloMessage.setCipherSuites(Modifiable.insert(new
       ↪  byte[]{(byte)0xfe, 0x00}, 0));
13
14     WorkflowTrace workflowTrace = new WorkflowTrace();
15     workflowTrace.addTlsActions(
16       new SendAction(clientHelloMessage),
17       new ReceiveTillAction(new ServerHelloDoneMessage())
18     );
19
20     runner.execute(workflowTrace,
       ↪  c).validateFinal(Validator::executedAsPlanned);
21   }
22 }
```

Listing 5: Simple testcase

The first section configures the `WorkflowRunner` object, more precisely the test derivation mode (line 7). This instructs the test case derivation logic to create a TLS handshake

for every cipher suite the server supports. The derived handshakes only send one cipher suite in the ClientHello cipher suite list.

In lines 9 to 19 a `Config` object and a `WorkflowTrace` with two actions are created. The first action sends the ClientHello message to the server. Before the message is sent, the modification that is attached to the cipher suites field of the message is executed. This modification adds an unspecified cipher suite to the beginning of the list in the ClientHello message. The second action receives messages from the server until a ServerHelloDone message is encountered. When such a message is received, it is safe to assume that the server selected the cipher suite it supports.

Finally, the assembled `WorkflowTrace` object is executed (line 21). The lambda function passed to the `validateFinal` function checks for every generated handshake whether the actions of the `WorkflowTrace` were executed successfully. If this is not the case, the function throws an exception. Depending on the result of each handshake, the overall test result is set by the framework as described in Section 3.2.

3.3.7 GUI support in IDEs

Since every test case is a regular JUnit test, it is possible to start test cases right from the IDE. This is especially useful during the development of new test cases. Launching a test case from the GUI results in a different control flow compared to launching the test suite from the console using the command-line arguments. Instead, the IDE launches the test case directly without calling the `runTests` function of the `TestRunner` class, which triggers the preparation phase, configures JUnit and executes the test cases.

3.3.7.1 Preparation Phase

To work around these limitations, the first conditional test execution JUnit extension checks if the test framework is already configured. If this is not the case, it triggers the preparation phase. However, this does not configure JUnit itself, which means that the tests are executed single-threaded and the framework does not create a report after the execution of the tests. To achieve parallel test execution and the creation of a report, two files have to be created inside the `resources` folder of the project containing the tests.

```
resources
├── junit-platform.properties
└── META-INF
    └── services
        └── org.junit.platform.launcher.TestExecutionListener
```

For parallel test execution the `junit-platform.properties` file must have the content of Listing 6. The report is generated if the `org.junit.platform.launcher.TestExecutionListener` file has the content of Listing 7. The latter contains a reference to the class implementing the `TestExecutionListener` interface provided by JUnit, which contains a callback function that is executed after all tests are finished. This function is used to generate the final test report.

```
1 junit.jupiter.execution.parallel.enabled = true
2 junit.jupiter.execution.parallel.mode.default = concurrent
```

Listing 6: Content of the `junit-platform.properties` file to achieve parallel test execution.

```
1 de.rub.nds.tlstest.framework.reporting.ExecutionListener
```

Listing 7: Content of the `org.junit.platform.launcher.TestExecutionListener` file for test report generation.

3.3.7.2 Command-Line Arguments

Another feature that works differently when tests are executed using the GUI features of an IDE, is the way command-line arguments work. The framework provides the features to develop tests for clients and servers, but it is only possible to specify a single set of command-line arguments. Instead of using these, the framework looks up the environment variables `COMMAND_SERVER` and `COMMAND_CLIENT`. These variables should contain the command-line arguments for the server respectively client tests. Dependent on the kind of test being executed, the framework looks up the corresponding variable and configures itself by parsing the value of the variables as command-line arguments. If a test is executed that works for client and server tests and both environment variables are defined, the client arguments take precedence.

3.3.8 Test Report

After the test execution, a test report is generated. The test report includes information on the amount of successful and failed test cases, which tests succeeded and failed, as well as the test result of every handshake.

The test report is generated by serializing the `AnnotatedStateContainer` objects created for each test case and stored inside the shared `TestContext` object during runtime. The output format and output file location are determined by the given command-line arguments. The format of the output file can be either a JSON or XML file, containing the serialized objects.

The information attached to each `AnnotatedState` object provide the possibility to find each performed handshake inside recorded network traffic. This allows the examination of every handshake with an interesting result on byte level.

3.4 Testsuite

The developed testsuite targets server and client implementations for TLS versions 1.2 and 1.3 and contains 175 test cases in total that are distributed as shown in Table 3.5. The implemented tests are based on the specifications from 11 different RFCs, addressing both TLS 1.2 and TLS 1.3. The repository contains an annotated PDF file for each RFC with test coverage, highlighting lines in green that are tested in a test case, lines that are not tested in orange and lines where a server or client test is missing in blue.

	TLS 1.2	**TLS 1.3**	**Total**
Server	49	33	82
Client	31	38	69
Both	17	7	24
Total	97	78	175

Table 3.5: Distribution of the developed test cases.

3.4.1 Implemented Tests

The focus of the implemented tests is to provide as much coverage of the underlying specifications as possible for features most clients and servers support. As a foundation, the test cases developed by Ebert [18] are used. These tests are migrated to the new more flexible test framework.

The existing testsuite containing 42 TLS 1.2 server tests, is extended by tests targeting TLS 1.3 based on RFC 8446 [25]. In addition, client tests targeting TLS 1.2 and TLS 1.3 and further tests for already covered RFCs are added as well.

The tests included in the testsuite currently do not address every feature of TLS. For TLS 1.2, features like session resumption and renegotiation are not covered. In case of TLS 1.3, there are no tests available covering the 0-RTT handshake or early data. In addition, there is no test implemented that covers client authentication or certificate validation. These topics are skipped due to time constraints and because these features require special configurations of the server and clients under test. However, the test framework is prepared to handle tests covering these features.

There are two categories of tests implemented.

1. RFC Compliance Tests

2. Length Field Tests

3.4.1.1 RFC Compliance Tests

Compliance tests check if the behavior of the target conforms to the specification. To be able to implement these tests, it is required to read the RFCs and pay attention to the signal word *MUST*. There are basically two kinds of tests. In some cases, it makes sense to develop a test for clients and servers and in other cases, it only makes sense to write a test for either the server- or the client implementation, as the following examples show.

The TLS 1.2 RFC contains the following sentence: "Recipients of Finished messages MUST verify that the contents are correct." [37, Section 7.4.9]. Since both parties send and receive a Finished message during the handshake, it is sensible to implement the test case for both, client and server.

On the other hand, the TLS 1.3 specification contains the following sentence: "If this [the supported versions] extension is not present, servers which are compliant with this specification and which also support TLS 1.2 MUST negotiate TLS 1.2 or prior as specified in [RFC5246]." [25, Section 4.2.1]. For this example, only a server test makes sense because it is only defined how the server has to react. In this case, the test is only executed if the server supports TLS 1.2 and TLS 1.3. This test sends a ClientHello without the supported versions extension. If the server negotiates TLS 1.2, the test is successful.

Besides categorizing the test case on the targets, which is useful for developing, it is also possible to categorize them based on the results. This system is used for the evaluation in Section 4.1.

Expected Alert $\neq$ Received Alert Many of the tests violate a condition of the specification on purpose and check if they receive the alert message that is specified. Many implementations send different alert messages than specified or no alert message at all. Violating such a specification can result in the fact that an attacker can use the server as an oracle. A widely known attack based on such an oracle is, for example, the padding oracle attack [19].

Expected Handshake Termination The test cases that belong to this category violate a specification described in the RFC. In contrast to the previous category, the RFC does not specify the response of the receiving peer. It only specifies that a message MUST or MUST NOT fulfill certain criteria. The test cases violate the criteria on purpose and expect a handshake termination. Not implementing the specification correctly and parsing the messages according to the specification, can lead to side channels that can be used as an oracle and might result in attacks.

Expected different Behavior Test cases under this category check if the sever behaves as expected. The expected behavior excludes sending alert messages, since this is already covered by the first category.

The following RFCs are covered by tests.

- RFC 4492 [24] – Elliptic Curve Cryptography (ECC) Cipher Suites for Transport Layer Security (TLS)
- RFC 5246 [37] – The Transport Layer Security (TLS) Protocol Version 1.2
- RFC 6066 [20] – Transport Layer Security (TLS) Extensions: Extension Definitions
- RFC 6176 [38] – Prohibiting Secure Sockets Layer (SSL) Version 2.0
- RFC 7366 [21] – Encrypt-then-MAC for Transport Layer Security (TLS) and Datagram Transport Layer Security (DTLS)
- RFC 7465 [39] – Prohibiting RC4 Cipher Suites
- RFC 7507 [40] TLS Fallback Signaling Cipher Suite Value (SCSV) for Preventing Protocol Downgrade Attacks
- RFC 7568 [41] – Deprecating Secure Sockets Layer Version 3.0
- RFC 7685 [22] – A Transport Layer Security (TLS) ClientHello Padding Extension
- RFC 8446 [25] – The Transport Layer Security (TLS) Protocol Version 1.3
- RFC 8701 [42] – Applying Generate Random Extensions And Sustain Extensibility (GREASE) to TLS Extensibility

3.4.1.2 Length Field Tests

These tests are implemented in the `LengthFieldTest` class. This includes four tests, addressing TLS 1.2 and TLS 1.3 clients and servers. As described in Section 2.1, most sections of TLS messages are preceded by a length value that specifies how many bytes of the transmitted data belong to the specific section. If this value is wrong, the implementation can not parse the message, thus the handshake will fail.

In TLS-Attacker, every length value is a modifiable variable and annotated with the `ModifiableVariableProperty` annotation. The test cases create multiple workflows, each modifying a single length value by iterating through every accordingly annotated field. The test is successful if the handshake could not be completed successfully.

3.5 Report Analyzer

As mentioned in Section 3.3.8 the output of the execution of the testsuite is a JSON file, which contains the results for each test case as well as details and the result for every performed TLS handshake.

The report analyzer provides a way to quickly analyze which test failed or what happened in a specific handshake. It is implemented as a web application, based on VueJS [43] in the frontend and a Node.js [44] Express [45] server with a MongoDB [46] database in the backend.

The application provides an interface to upload the test report JSON file, a PCAP file containing the network communication between the testsuite and the target, as well as the keylog file, which contains the premaster secrets of all handshakes. The PCAP and keylog files are stored in the database as is, while the JSON file containing the results is processed further. Figure 3.6 shows how the uploaded files are stored inside MongoDB collections and the relations between these.

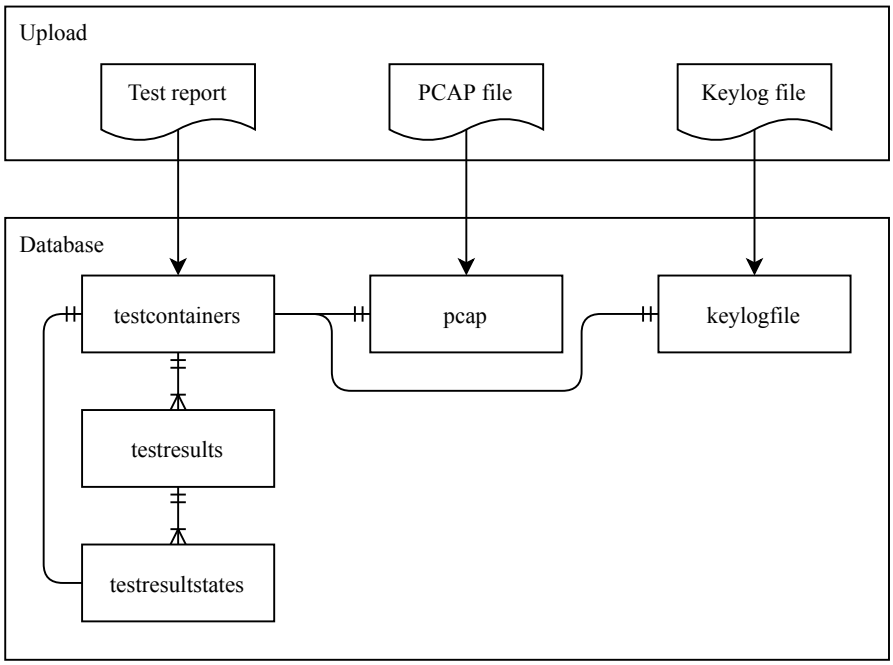


Figure 3.6: Upload process and processing of the documents into MongoDB database collections.

The information from the test report JSON file are stored across three different collections, as shown in Figure 3.6.

1. `testcontainers`
   Contains information about the execution of the testsuite. That means for each

upload, one document inside this collection exists. Each document contains a list of references to documents inside the `testresults` collection.

2. `testresults`
   Contains a document for each test method. The data include but are not limited to the severity level, method name, the result, a stacktrace in case of a failed test as well as a list of references to documents inside the `testresultstates` collection.
3. `testresultstates`
   Contains a document for every performed handshake and its properties, such as the result, source and destination port and timestamps.

Furthermore, the application provides two main views for the visualization of the test results.

- Analyzer View
- State View

Analyzer View

This view provides the option to load and visualize test reports from the database (Figure A.2). The test methods and their results are displayed in a table. The leftmost column shows the test methods grouped by their Java class whereas the remaining columns show the result of each test method. The icons in front of the method name visualize the security and interoperability severity level of the test method.

There are some hidden interactions available that are explained in the following.

- Hovering over the result icon reveals how many handshakes were performed by this test method.
- Clicking on the method name shows a popup with additional information about the test case.
- Clicking on a table row of a test that performed TLS handshakes, navigates to the state view.

State View

The State view provides an overview over the executed TLS-Attacker `State` objects, each representing a TLS handshake, that belongs to a single test method (Figure A.3). The view is also presented as a table, where the leftmost column shows the deterministic unique identifier of the handshake, that identifies the same handshake configuration across different executions of the testsuite.

To retrieve specific information about a single handshake, it is possible to click on the result

icon, or to click on the unique identifier to get information for every handshake in the row or the column header for every handshake in the selected column.

The information popup provides the JSON excerpt from the test report as well as two buttons. One button extracts the sent messages for the given handshake from the PCAP dump and shows a view similar to Wireshark [47]. The other button can be used to download a PCAP file that only contains the messages belonging to the selected handshake. This file can be opened with Wireshark for further packet inspection.

The PCAP download/visualization features are implemented as follows. The backend fetches the uploaded PCAP file belonging to the handshake from the database and uses tcpdump [48] first to extract only packets that are sent from the source or destination ports that are part of the report data. Next, tshark, a program that is part of Wireshark, is used to filter the remaining packets based on the time when the handshake started and ended. The output of tshark is similar to the representation in Wireshark and is displayed in the frontend.

3.5.1 Score Calculation

Another feature that is currently implemented by the report analyzer is the calculation of a test score, that makes it possible to compare test results. The calculation depends on the severity levels that are assigned to each test method and their result. Therefore, two scores are calculated. A security score and an interoperability score. Only tests that are not disabled during the testsuite execution are considered in the calculation process.

Depending on the severity level, a test case earns up to a specified number of points. The higher the severity the more points are available. Depending on the test result, the amount of points is lowered. For a succeeded test, 100% of the available points are added to the reached score counter. A failed test does not add any points to the score. A `PARTIALLY_SUCCEEDED` test earns 80% of the points whereas a `PARTIALLY_FAILED` test earns only 20% of the points. This results in the matrix shown in Table 3.7. The purpose of this matrix is that tests with a higher severity contribute more to the score than lower severity tests. If an implementation fails at a low number of high severity tests, it might have a lower score than an implementation that fails at more tests of the lowest severity level.

To achieve comparability between two testsuite executions against different implementations, the percentage of the reached score is used.

SeverityLevel	TestResult ✓	✓	✗	✗
CRITICAL	100	80	20	0
HIGH	80	64	16	0
MEDIUM	60	48	12	0
LOW	40	32	8	0
INFORMATIONAL	20	16	4	0

✗ `FAILED`
✗ `PARTIALLY_FAILED`
✓ `PARTIALLY_SUCCEEDED`
✓ `SUCCEEDED`

Table 3.7: Scoring system. Shows the score added to the scoring counter depending on the severity level and the test result.

3.6 TLS-Docker-Library

The TLS-Docker-Library [33] is an existing project, as described in Section 2.4. To be able to analyze many TLS implementations, the library is improved as part of this thesis.

The central aspect of this project is the collection of Dockerfiles that include build instructions for 23 different TLS implementations, configured as server or client.

3.6.1 Docker Images – Build System

To build the Docker images using these Dockerfiles, a collection of shell scripts exists, that start the build process. The current build system is limited in its capabilities. For example, it is not possible to selectively build a specific version of a certain TLS implementation. Moreover, it is not optimized for a multi-processor system. Therefore, building multiple images in parallel is not possible. Thus, building over 1000 images without parallelization takes more than a day. To work around those limitations, the build system is improved by the following features.

The optimized build system provides a new Python script with a command-line interface and useful logging. This allows to debug which builds succeeded and why a build failed. The script only logs the name of the image and the status of the build process to the console, thereby providing a quick overview of how many builds succeeded or failed. In addition, it creates a log file containing detailed information about failed builds, so that occurred problems can be debugged. Using this script also allows to selectively build an image of specific versions of a specific implementation and to build multiple Docker images in parallel using Docker BuildKit [49]. Docker BuildKit parallelizes multiple

independent stages of a Dockerfile. In addition, the script executes multiple Docker builds in parallel. This results in higher utilization of multi-processor systems and therefore reduces the build time for the Docker images significantly. On a system with 24 cores and execution 16 Docker build tasks in parallel, 1430 Docker images could be built in 6.75 hours.

The main reason for failed builds was, that the used Docker base image was not pinned to a specific version. The implementations are compiled from source code and depend on other packages, such as compilers and other libraries. Since the project was created in 2017 by Ebert [18], the Docker base image, alpine-linux, has been updated several times. These updates of the image also resulted in updated dependencies. Thus, the build process failed for many Docker images using the latest version of the alpine-linux image. Setting the version of the base image to 3.6, which was the most recent version in 2017, fixed most of the builds.

In addition, new versions for the implementations are added that are evaluated in Section 4.1. All in all, these improvements result in 1430 working Docker images for 23 TLS implementations that can be built using the library.

3.6.2 Docker Images – Entrypoints

Additional improvements are implemented regarding the way TLS client and server executables are started. Especially client tests depend on the possibility to trigger the client to initiate a connection to the server executing the tests. Some server implementations, for example, BoringSSL below version 2987 terminate after a single handshake is performed. Restarting the Docker container for every single handshake is inefficient and slow when many handshakes need to be performed.

To address this problem, two small wrapper applications are implemented written in Go for client and server applications, respectively. The benefit of using Go for this task is that the language provides many features and compiles to native executables. Executing these do not require an installed runtime environment or interpreter. This keeps the size of the built Docker images small.

The wrapper applications take an infinite number of arguments, treating the first argument as a program and the remaining as arguments for the program. During runtime, the specified program is executed with its arguments. The execution strategy depends on the type of the image (server or client). In addition to that, an HTTP server providing a REST API is created by the wrapper application listening on port 8090. When the HTTP server receives a GET request on the `/shutdown` endpoint, the application is terminated.

Client Images. The specified program, in this case a TLS client implementation, is executed when the wrapper application is started and every time a GET request is sent

to the `/trigger` endpoint.

Server Images. The specified program, in this case a TLS server implementation, is executed in an infinite loop which starts when the wrapper application is started. This is useful for server applications terminating after one handshake. In case of an unexpected crash of the server, the TCP port used by the server might still be used by the operating system. Therefore, the server terminates immediately again until the port is released. This behavior is undesired as it is time-consuming. To accelerate this process, the wrapper application only tries to restart the server 5 times. If the server always terminates in less than 100ms, the wrapper application terminates as well. In this case, restarting the Docker container is faster than waiting for the operating system to release the port.

3.6.3 Java Library

The Java library of the TLS-Docker-Library provides an API to instantiate Docker containers that run a TLS implementation. The API is improved as part of this thesis, in order to be able to customize the configuration, like network settings or resource limits. The `getTlsClient` or `getTlsServer` functions return a `DockerTlsInstance` object. This object provides a `getContainerConfig` function returning a `ContainerConfig` that is part of the Docker library provided by Spotify [50] to control the Docker daemon. The `ContainterConfig` object can be modified before the container is started. This change makes the usage of the library more flexible for other projects that depend on it.

Furthermore, the profiles are updated containing the command-line arguments that are needed for the execution of the TLS implementations. This includes adding a new parameter type `INSECURE` for client profiles. Using this parameter disables the server certificate validation of the client. Setting the `insecureConnection` boolean value of the `DockerTlsInstance` object to `true` launches the client application of an implementation with the `INSECURE` and without the `CA_CERTIFICATE` parameters.

4 Evaluation

The improvements described in Section 3.6 result in more than 1000 TLS implementation Docker containers of 23 different implementations that are analyzed in this chapter with the developed testsuite.

4.1 Setup

Although there are Docker images for 23 implementations available in the Docker library, it is not possible to analyze every implementation, because some of them are less tolerant regarding TLS hello messages. Thus, the TLS-Scanner that runs before the testsuite for server tests can not scan every implementation. This means that it is not possible to detect the supported cipher suites and other algorithms. However, this is required by the testsuite. The same applies to clients. Although the client initiates the connection with a ClientHello message and the server only needs to select the correct algorithms, the same problems appear for clients.

Solving these problems should be possible by changing the values of the used TLS-Attacker `Config`. This results in different values for the hello messages that work for the implementation, so that and the handshakes can be performed. This is out of scope for this thesis, because it would require a lot of time using a trial and error approach that changes a different value every time to find a working setup for every implementation. Since the majority of implementations work with the default configuration, the benefit of fixing those problems is small.

Table 4.1 lists the different implementations where at least the server or client application could be tested. The Docker images available for these implementations are up-to-date. The table is sorted by the popularity of their GitHub repositories. Implementations that do not provide an executable server or client application are not considered for the evaluation, to exclude implementation failures by developers who are not familiar with the library.

The main repository of BoringSSL and NSS is not available on GitHub, therefore, the numbers are not available. Since these implementations power the TLS stack of Google Chrome and Firefox, respectively, they are also considered as very popular.

All implementations are tested with their default configuration using the sample applications that are the result of the compilation of the implementation. For tests against

Repository	Stars	Forks	Open Issues	Language
OpenSSL	13192	5860	1329	C
s2n	3731	495	355	C
mbed TLS	2460	1410	638	C
Rustls •	1918	174	56	Rust
Botan	1423	348	109	C++
LibreSSL	925	203	52	C
wolfSSL ◦	883	376	83	C
GnuTLS	248	22	201	C
MatrixSSL ◦	135	33	10	C
tlslite-ng •	98	41	61	Python
BearSSL •	16	6	0	C
BoringSSL	n.a.	n.a.	n.a.	C/C++
NSS	n.a.	n.a.	n.a.	C/C++

• Only servers could be tested.
◦ Only clients could be tested.

Table 4.1: Evaluated TLS implementations sorted by their GitHub repository popularity, as of 19th June 2020.

implementations running in client mode, the certificate validation for server certificates is disabled. TLS server implementations are configured using a 2048-bit RSA certificate. Therefore, cipher suites that depend on a non-RSA server key can not be negotiated.

The tests are performed using the TLS-Testsuite-Large-Scale-Evaluator [51]. This small project manages the execution of the testsuite and the Docker containers running the TLS implementation. It provides a command-line interface that allows specifying which TLS implementation and version is tested. For the evaluation, the testsuite is also running inside a Docker container. The startup script for the testsuite container starts an instance of tcpdump to capture the network traffic between the testsuite and the target as well as the testsuite itself.

The evaluator is capable of testing multiple targets in parallel. Each testrun tests one specific version of an implementation, that is one TLS implementation Docker container. Therefore, the testsuite Docker container and the TLS implementation container have to be started. The evaluator coordinates the start of the Docker containers and monitors the execution. A testrun is considered as finished when the testsuite container terminates. Until this event occurs, the evaluator monitors the status of the TLS implementation container in a separate thread and restarts the implementation container if it terminates before the testsuite container. The restart mechanism could also be achieved using a Docker feature, but during the tests this was not reliable. Therefore, polling the container status and restarting it when necessary is faster and more reliable.

To isolate the network traffic for each testrun, a Docker network is created before the Docker containers are started. The testsuite and TLS implementation containers are configured to be attached to the network. Therefore, both containers communicate with each other directly without any routers in between that could interfere with the network traffic. Figure 4.2 shows the evaluator managing the execution of three testruns in parallel, testing the TLS implementations A, B and C, which can be TLS clients or servers. The evaluator is implemented in a generic manner, therefore this setup can be extended for other purposes, for example, efficient validation of the functionality of the TLS implementation Docker containers.

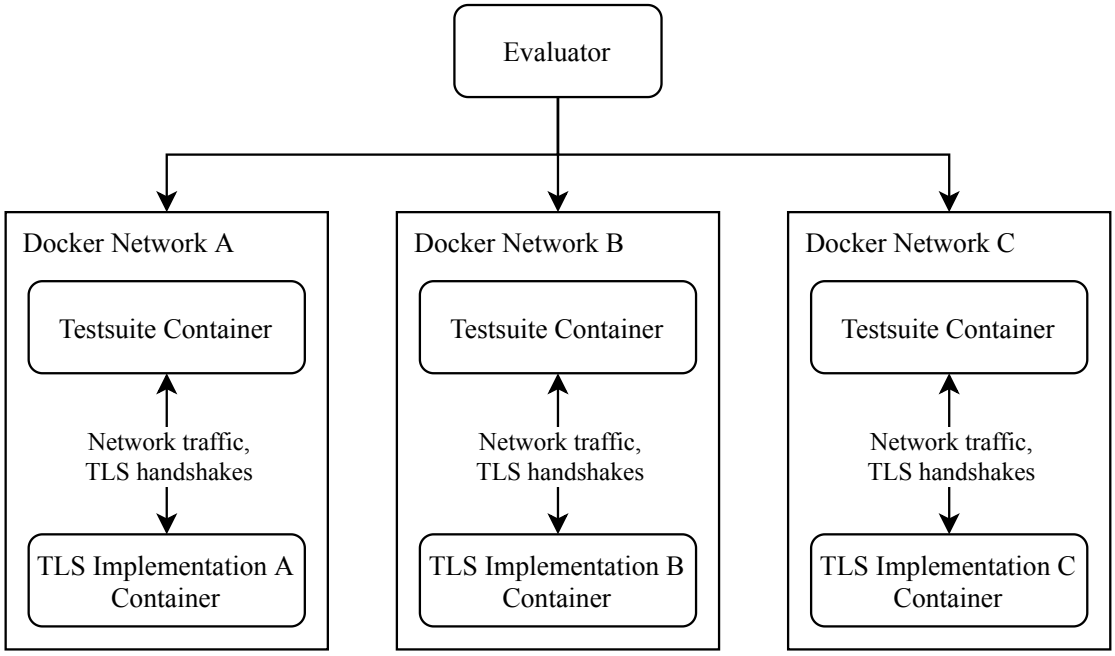


Figure 4.2: Parallel evaluation of three testruns.

Before the final testrun was started on which the results below are based on, the testsuite was executed two times against the newest available versions of all implementations. After it was ensured that the test results of both testruns were equal, the final testrun was started.

4.2 Results

This section discusses the results of the evaluation and is split into two subsections. In the first subsection, the newest versions of all implementations are compared to each other regarding their reached score, their default configurations and individual test results. The second subsection evaluates the development of the score for different versions of the same implementation. This provides insights on what was changed during the lifecycle of

an implementation.

4.2.1 Multiple Implementations - Newest Versions

The versions of the analyzed implementations are listed in Table 4.3. The listed versions were up-to-date in June 2020. The client version of Botan was released in March 2019. Since no handshake could be performed with later versions, this one is used instead. The situation for LibreSSL is similar, although version 3.1.3 was released after version 3.2.0 as a maintenance release. However, the features between those two major releases might be different.

Repository	Version
BearSSL •	0.6
BoringSSL	3945
Botan	2.14.0 (S), 2.10.0 (C)
GnuTLS	3.6.14
LibreSSL	3.1.3 (S), 3.2.0 (C)
MatrixSSL ○	4.2.2
mbed TLS	2.24.0
NSS	3.54
OpenSSL	1.1.1g
Rustls •	0.17.0
s2n	0.10.7
tlslite-ng •	0.8.0-alpha38
wolfSSL ○	4.4.0-stable

• Only servers could be tested.
○ Only clients could be tested.

Table 4.3: Versions of the analyzed TLS implementations.

4.2.1.1 Overall Results

First, the overall results are discussed to get an overview how the implementations reacted to the tests in general. The comparison is based on the scoring system, that assigns two scores to each implementation, the security and interoperability score.

Clients

The results of the client tests are shown in Figure 4.4. The implementations are sorted by their popularity from top to bottom. The chart shows that the most popular implementations are among the implementations that reach the highest score.

s2n and mbed TLS reach the lowest score. For both implementations, there are reasons for this result. mbed TLS does not support record fragmentation. Therefore, tests that perform TLS handshakes terminate most of the times with the result value `PARTIONALLY_FAILED` due to the test derivation. Only those handshakes can succeed that are not generated by the test derivation feature and do not contain fragmented TLS record layers.

In the case of s2n, the problem is different. If s2n receives a message that does not comply to the specification, the implementation often neither responds with an alert message nor closes the socket. This makes it impossible for the testsuite to determine the reaction of the implementation. When a message does not conform to the specification, in most cases an alert or other termination action is expected. If this does not happen, the test fails.

Another implementation that reaches a noticeable score is Botan, since it reaches the highest interoperability and third highest security score, although it is only at position 6 sorted in the popularity ranking. The reason for this is that Botan only supports secure cipher suites by default and does not support TLS 1.3. This results in 50 test cases less that are executed compared with most popular implementations. However, the performance throughout the test cases is also very good.

MatrixSSL has a special role among the implementations. The results show that it was only possible to negotiate a TLS 1.3 cipher suites with the client, although it proposes TLS 1.0 to TLS 1.2 including the DTLS versions in its supported versions extension. Therefore, only the TLS 1.3 implementation of the analyzed version is used for the evaluation.

Servers

The results of the server tests are shown in Figure 4.5. The range from the lowest to the highest score is higher compared to the client tests. The reason for this is that server tests require that the client initiates the handshake with the ClientHello message. In contrast to that, client tests can already operate on the received ClientHello message without sending any messages to the client. In addition, the server performs more work that is prone to failure, such as selecting the algorithms for the connection.

Because of this, the scores of the s2n and mbed TLS servers are even lower compared to the client implementations. The TLS servers of these implementations have the same problem as the clients. Most of the tests of mbed TLS `PARTIALLY_FAIL` because it does not support fragmented TLS record layers.

Regarding s2n there is a new observation that is the difference of 24% between the interoperability and security score. The testsuite also contains server tests that check if the server chooses parameters correctly regarding the preference of the client. These tests have usually a higher interoperability severity than security severity level. Since these tests

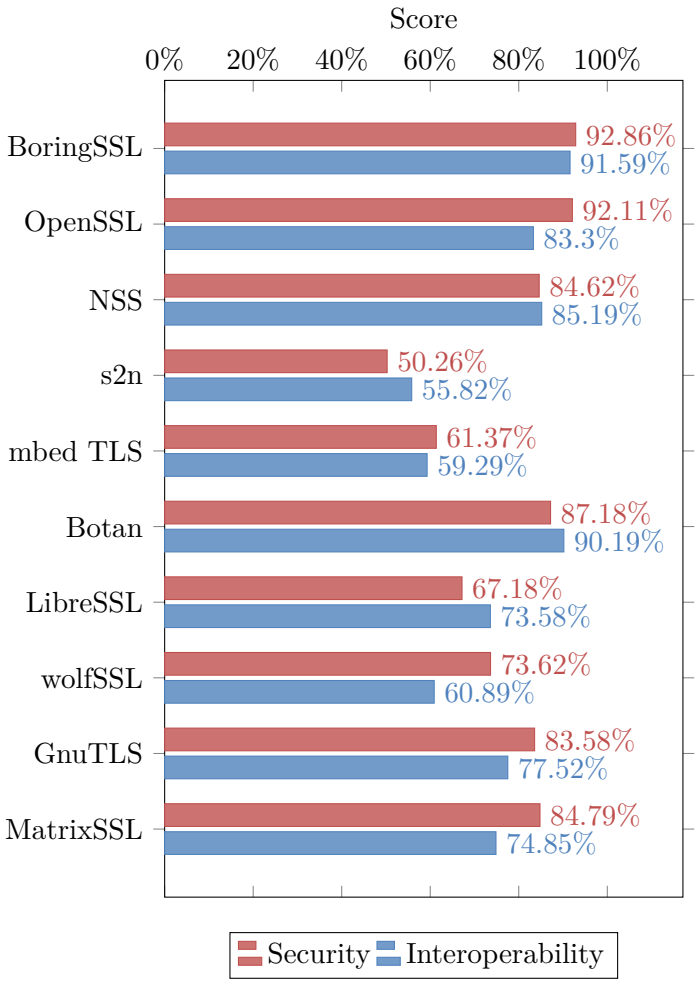


Figure 4.4: Scores of the newest tested client implementations sorted by their popularity.

do not send invalid TLS messages, they succeed. Tests that send invalid TLS messages and try to uncover implementation failures usually have a higher security severity than interoperability severity level. In these cases, s2n often neither sends an alert message nor closes the TCP socket, which results in a failed test.

Another implementation where the score probably does not reflect the real-world situation, is GnuTLS. The test results for TLS 1.3 are part of the score, although they contain false-negative results. The server of the implementation is extremely unreliable and does often not respond to TLS 1.3 ClientHello messages. Despite the safeguard mechanisms implemented by the framework to ensure a reliable connection, this problem occurred. Because of this, nearly all TLS 1.3 tests against this implementation fail and therefore lower the score. This issue only affects the server of this implementation.

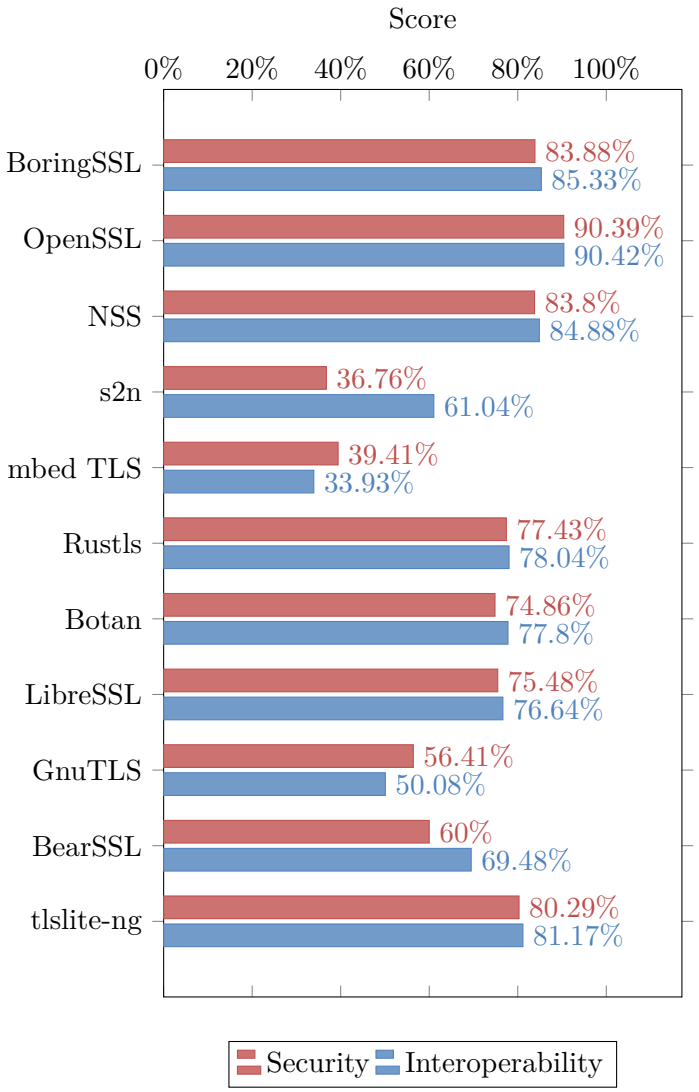


Figure 4.5: Scores of the newest tested server implementations sorted by their popularity.

4.2.1.2 Default Configuration

Despite the results of individual test cases it is also worth looking at the default configuration regarding the supported cipher suites of the implementations. Since every implementation is compiled without any extra flags that disable certain cipher suites or protocol versions, analyzing this configuration gives an overview of the security by default principles of each implementation.

Table B.1 lists which TLS versions the implementations support as well as properties of the supported cipher suites. Looking at the table reveals that BoringSSL, NSS, tlslite-ng, LibreSSL and BearSSL support cipher suites using either 3DES or RC4 that

both are insecure. These four implementations provide the worst default configuration.

NSS and LibreSSL are the only implementations that support the RC4 stream cipher and MD5 as hash algorithm by default. NSS shows a strange behavior. If the server has access to an RSA- and EC certificate instead of only an RSA certificate, only cipher suites that use the EC certificate and provide PFS are supported by the server. The old legacy algorithms (RC4, 3DES, MD5) are not supported anymore, although cipher suites using an EC certificate and these algorithms are specified in RFC 4492 [24, Section 6], such as `TLS_ECDHE_ECDSA_WITH_RC4_128_SHA`.

A better configuration is provided by the implementations that do not offer cipher suites using the insecure algorithms, but still support non-PFS cipher suites. These are OpenSSL, s2n, mbed TLS and GnuTLS.

The best default configuration is provided by Rustls, Botan and wolfSSL. Those implementations only support the most recent cipher suites that provide PFS and do not use the CBC mode of operation for encryption algorithms.

4.2.1.3 Server (TLS 1.2)

This and the following sections discuss the results of individual server and client test cases, grouped by the test categories explained in Section 3.4.1. Only the tests resulting in the most curious behaviors are discussed. This is specifically the case when multiple implementations show different behaviors. Table B.2 shows a summary of how many test cases of the discussed categories failed for a specific implementation.

Expected Alert ≠ Received Alert

Invalid Record Layer Content-Type. During the TLS handshake 9 implementations fail at a class of test cases that send an invalid record layer content-type. One test sends a ClientHello message inside a record layer with the content-type `0xff`. The second test sends the CCS message in such a record layer. In both cases an `unexpected_message` fatal alert must be sent [37, Section 6]. The results of the test with the ClientHello message show that BearSSL, Botan, LibreSSL, mbed TLS and Rustls close the TCP socket. GnuTLS and tlslite-ng send an `record_overflow` alert, NSS sends an `illegal_parameter` alert and s2n neither closes the socket nor sends an alert message. If the invalid content-type is sent with the CCS message, Botan, GnuTLS and LibreSSL, which failed before, complete the second test successfully by sending the correct alert message. BearSSL, mbed TLS, tlslite-ng and s2n show the same behavior as before. NSS on the other hand now sends a `decode_error` alert.

SCSV Fallback. TLS servers can detect a fallback of the TLS version if the client sends a special cipher suite as part of the cipher suite list and if the highest supported TLS version sent by the client is lower than the highest supported version by the server. In this case, the server must respond with an `inappropriate_fallback` fatal alert [40, Section 3]. s2n does not fulfill the specification. The implementation does not continue the handshake but also neither closes the TCP socket nor sends an alert.

Lower TLS Version. If the ClientHello proposes a lower TLS version than the server supports, the server must send a `protocol_version` alert according to RFC 5246 [37]. NSS and OpenSSL respond with a `handshake_failure` alert, s2n does neither send an alert nor closes the TCP socket.

Max Fragment Length Extension. One test case covers RFC 6066 [20] that specifies the maximum fragment length extension. The RFC specifies certain values that are valid length values. If the client sends an invalid length the implementation must abort the handshake with an `illegal_parameter` alert [20, Section 4]. From the tested implementations, BearSSL, GnuTLS, mbed TLS and OpenSSL support this extension. BearSSL does not fulfill the specification. Instead of sending the alert message, it only closes the TCP socket.

GREASE – ALPN Extension. Although RFC 8701 [42] only refers explicitly to the RFC of TLS 1.3, tests for TLS 1.2 are implemented as well checking if the implementation complies to the RFC. This provides insights how implementations react to unknown values. The first test covering this RFC, addresses the Application-Layer Protocol Negotiation Extension specified in RFC 7301 [52]. The test sends the special GREASE value as protocol name in the extension. Only Botan and tlslite-ng support the ALPN extension. Botan negotiates the `echo/0.1` ALPN protocol, that was not offered by the client in the ClientHello message and thus fails the test. tlslite-ng responds with a `no_application_protocol` alert and conforms to the specification.

Expected Handshake Termination

Zero-Length Fragment – Handshake Record. According to the RFC, implementations "MUST NOT send zero-length fragments of Handshake, Alert, or ChangeCipherSpec content types" [37, Section 6.2.2]. The implemented test cases check how the implementations react if they receive zero-length record layer fragments of Handshake and Alert content types. The expected result is a termination of the handshake since it violates the specification. Receiving a zero-length handshake fragment followed by a record layer containing the ClientHello message shows three different behaviors. BearSSL, BoringSSL, LibreSSL, NSS, OpenSSL and Rustls continue with the handshake normally, thus fail the

test. Botan, mbed TLS and s2n close the TCP socket, whereas GnuTLS and tlslite-ng respond with an `unexpected_message` fatal alert.

Zero-Length Fragment – Alert Record. Some implementations change their behavior shown above if they receive a zero-length fragment of an alert message. The test case sends a `close_notify` warning level alert. Without a zero-length fragment, the other peer should respond with the same alert before closing the socket [37, Section 7.2.1]. Since the test case sends a zero-length fragment, the expected response is either a fatal alert or the termination of the socket. s2n, BearSSL and Rustls send a `close_notify` warning alert. This indicates that they ignored the zero-length fragment and continued normally. LibreSSL and mbed TLS close the socket, whereas the remaining implementations send a fatal alert. In comparison to the zero-length ClientHello test, s2n, Botan, OpensSSL, NSS and BoringSSL react differently. s2n now accepts the zero-length fragment, whereas the other implementations now reject it. The behavior of Rustls, mbed TLS and tlslite-ng are consistent.

Select RC4 Cipher Suite. As already shown in Table B.1, two server implementations support cipher suites using RC4. The test cases covering RFC 7465 [39] that prohibits RC4 cipher suites, test if the server aborts the handshake when the client offers only RC4 cipher suites. NSS and LibreSSL both continue the handshake and thus select an RC4 cipher suite. This test case is rated with a critical security severity since the RC4 cipher is insecure [53].

ECC Extensions. RFC 4492 and 8422 specify two extensions addressing the handling of elliptic curves. The RFCs specify that the client must not send the `elliptic_curves` and `ec_point_formats` extension "if it does not propose any ECC cipher suites" [24, Section 4]. The test case sends a ClientHello message that violates this specification. The expected result is a termination of the handshake. All implementations show the same behavior. They ignore the extensions and continue with the handshake. This is a sensible choice to reach higher interoperability. However, this still violates the specification and thus results in a failed test. To compensate the impact of this test case to the overall score, both severities are set to the lowest level.

Expected different Behavior

Select RC4 Cipher Suites. A second test that addresses RC4 cipher suites works differently from the test explained above. This time the client sends RC4 cipher suites and one non-RC4 cipher suite. The non-RC4 cipher suite is replaced with every non-RC4 cipher suite that the server supports. The non-RC4 cipher suite is always at the bottom of the cipher suite list. According to the RFC, the implementation must not select a cipher suite using RC4 [39, Section 2]. Although NSS and LibreSSL support RC4 cipher suites,

only LibreSSL fails this test and selects an RC4 cipher suite. NSS always selects the non-RC4 cipher suite at the bottom of the cipher suite list.

Signature Algorithms Extension. RFC 5246 specifies that the server must use the `RSA_PKCS1_SHA1` signature and hash algorithm if the client does not send the extension as part of the ClientHello message. The test case sends a ClientHello message without the extension. Botan and Rustls completely abort the handshake responding with a `handshake_failure` fatal alert and thus fail the test. The other implementations select `RSA_PKCS1_SHA256` as algorithm. This is a sensible choice since SHA-1 is vulnerable to collisions [54]. A RFC deprecating SHA-1 is currently in draft [55].

Encrypt-then-MAC Extension. The Encrypt-then-MAC extension makes the implementation resistant against attacks mentioned in Section 2.1.1.1. Because of this, a test case exists that checks if an implementation supports this extension. The test sends the extension in the ClientHello to the server and expects the extension to be present in the receiving ServerHello message. Only GnuTLS, mbed TLS, OpenSSL and tlslite-ng support this extension. The test fails for the other implementations.

GREASE – Signature and Hash Algorithms Extension. A second GREASE test sends GREASE values as part of the signature and hash algorithms extension in the ClientHello, together with valid values. LibreSSL and s2n terminate the handshake although the list of algorithms contains values supported by the implementation. LibreSSL responds with a fatal `decode_error` alert. s2n does neither close the connection nor sends an alert or any other message.

4.2.1.4 Server (TLS 1.3)

Since the TLS 1.3 server implementation of GnuTLS is very unreliable, the results generated by this implementation are ignored in this section.

Expected Alert ≠ Received Alert

ClientHello Message – Compression. The TLS 1.3 RFC is stricter than the TLS 1.2 RFC. For the ClientHello message it requires specific values for fields that were flexible in TLS 1.2. If the value does not meet the requirement, the handshake must be aborted with a fatal `illegal_parameter` alert [25, Section 4.1.2]. This applies to the compression and version fields. Sending an illegal value for the compression field causes OpenSSL to respond with a `decode_error` alert instead of an `illegal_parameter` alert. The other implementations respond with the correct alert message.

ClientHello Message – Version. Other test cases modify the version field of the ClientHello message and instead of using the TLS 1.2 version (`0x03 0x03`), the test sets set the version to `0x03 0x04` and `0x03 0x03`, respectively [25, Section 4.1.2]. In this case no implementation responds with an alert, every implementation selects TLS version 1.3 in the ServerHello message. In contrast to that, setting the version field to `0x03 0x00` makes NSS and tlslite-ng fail with a `protocol_version` fatal alert, OpenSSL responds with `handshake_failure` alert and `0x03 0x00` as version for the record layer. The remaining implementations, BoringSSL and Rustls, still negotiate TLS 1.3. One possible explanation could be that `0x03 0x00` represents SSL 3.0 and trying to parse the TLS 1.3 ClientHello message could cause parsing errors. However, none of the implementations supports SSL 3.0 anymore.

Signature and Hash Algorithms Extension. The TLS 1.3 RFC requires that the signature and hash algorithms extension must be sent by the client if the server uses a certificate for authentication [25, Section 4.2.3]. If the client does not send this extension, the server must respond with a `missing_extension` alert. The test case does not send the extension. BoringSSL and Rustls respond with a `handshake_failure` fatal alert. NSS, OpenSSL and tlslite-ng respond with the correct alert message.

Expected Handshake Termination

Zero-Length Fragment – Handshake Record (ClientHello). As for TLS 1.2, the same test case also exists for TLS 1.3. The results are very similar. From the tested implementations that support TLS 1.3, only tlslite-ng rejects a zero-length handshake record layer in front of the ClientHello message with an `unexpected_message` fatal alert. The other implementations ignore the zero-length fragment and continue the handshake.

Zero-Length Fragment – Handshake Record (Finished). By sending a zero-length handshake record layer preceding the Finished message, the behavior of OpenSSL changes. OpenSSL and tlslite-ng respond with an `unexpected_message` fatal alert. The other implementations ignore the fragment and continue.

Keyshare Extension. The client sends the keyshare extension as part of the ClientHello message. According to the RFC the keyshare entries that are part of the extension and containing a public key "MUST correspond to a group offered in the 'supported_groups' extension and MUST appear in the same order" [25, Section 4.2.8]. The test case sends the keyshare entries in a different order than the groups listed in the `supported_groups` extension. Only tlslite-ng terminates the handshake with an `illegal_parameter` fatal alert. The other implementations continue the handshake normally. The test case

executes a complete handshake that shows that the other implementations select the correct combination of group and keyshare entry.

Expected different Behavior

The test cases that try to trigger a reaction of the server that is different from the expected behavior all complete successfully. These tests cover the `supported_versions` extension and test the version selection algorithm. Tests covering the GREASE RFC [42] are also implemented. These test cases do not show any interesting behavior.

4.2.1.5 Client (TLS 1.2)

Expected Alert $\neq$ Received Alert

Select Unsupported Version. A test case selects a TLS version that the client does not support and sends it as part of the ServerHello message. According to the RFC, the client must respond with a fatal `protocol_version` alert [37, Section E.1]. The version that is selected by the server is not specified in any RFC, more precisely `0x03 0x0F`. This ensures that the client does not support the version and must send the alert message. NSS and Botan send an `illegal_parameter` respectively `handshake_failure` fatal alert. s2n and wolfSSL close the TCP socket. Thus, these 4 implementations fail the test.

Invalid Record Layer Content-Type. This test case is already explained above as part of the server tests. The client test sends the invalid record layer content type with the ServerHello and CCS messages. It is expected from the client to respond with an `unexpected_message` fatal alert. NSS responds with a `decode_error` alert, when the invalid content type is sent with the CCS message record layer. In case of an invalid content type for the ServerHello, it responds with an `illegal_parameter` alert. In contrast to this, s2n, wolfSSL and mbed TLS show a more consistent behavior in those two tests. These implementations close the TCP socket and thus also terminate the handshake.

Additional Extension. If the client receives an extension that it does not request with the ClientHello message, "it MUST abort the handshake with an `unsupported_extension` fatal alert" [37, Section 7.4.1.4]. The test case checks the extension sent by the client in the ClientHello message and adds an extension to the ServerHello message that is not part of the ClientHello message. LibreSSL, s2n and wolfSSL continue with the handshake normally. mbed TLS responds with a `handshake_failure` alert. Every other implementation passes the test and sends the expected alert. It is not covered by any test if the clients respect the additionally sent extension.

GREASE – Extension. A second test case sends a GREASE extension as part of the ServerHello message. In this case only BoringSSL, Botan and NSS terminate the handshake. Botan responds with a `handshake_failure` fatal alert instead of an `unsupported_extension` alert message. OpenSSL, GnuTLS and mbed TLS accept the additional GREASE extension whereas they rejected the additional extension in the test case before. Botan on the other hand accepted the additional extension before and rejects the GREASE extension. This shows that the implementations check the extension type and react differently depending on whether they know the extension type or not.

Expected Handshake Termination

Zero-Length Fragment – Handshake Record (ServerHello). Part of the client tests is also a test case that sends a zero-length record layer fragment to the client. This is sent before the ServerHello message. Since the specification does not allow this, the termination of the handshake is expected. BoringSSL, NSS and OpenSSL continue the handshake and ignore the zero-length fragment. mbed TLS, s2n and wolfSSL terminate the handshake by closing the TCP socket. GnuTLS and LibreSSL respond with an `unexpected_message` fatal alert whereas Botan responds with a `decode_error` alert message.

Sending Alert Messages. The testsuite contains a client test that sends fatal alerts with every available alert description to the client. According to the RFC any connection must not be resumed that received a fatal alert [37, Section 7.2.2]. The alert is sent in two different tests, each using a different location for the injected alert message. The first test sends it in front of the ServerHello message and the other test sends it in front of the ServerHelloDone message. Both tests result in different behaviors of the implementations. If the alert is sent before the ServerHello message, LibreSSL ignores the `user_cancelled` fatal alert and continues the handshake normally. Every other alert description results in closing the TCP socket. In contrast to that, Botan and s2n ignore the `close_notify` fatal alert. If the alert is sent before the ServerHelloDone message, LibreSSL also terminates the handshake when the `user_cancelled` alert is sent. The behavior of Botan and s2n is consistent, so that they still only ignore the `close_notify` alert. These results show that the handling of the alert messages does not evaluate the alert level before the description.

GREASE – Server Initiated Extension Points. As for the TLS 1.2 server tests, the testsuite also includes client tests covering the GREASE RFC [42]. In this case, the server selects GREASE values as version, cipher suite, signature algorithm and elliptic curve and evaluates the reaction. Since the client does not know the selected values, a handshake termination is expected. Selecting a GREASE value as cipher suite shows that s2n and wolfSSL close the TCP socket, whereas the remaining implementations

respond with an `illegal_parameter` or `handshake_failure` alert. mbed TLS responds with an `internal_error` fatal alert. Setting the elliptic curve or signature algorithm to a GREASE value, the behavior of s2n changes. s2n now neither sends an alert message nor closes the TCP socket. The other implementations still send fatal alert messages.

Encrypt-then-MAC Extension. According to the RFC specifying the extension, a server "MUST NOT send an encrypt-then-MAC response extension back to the client" [21, Section 3], if it selects a cipher suite using a stream or AEAD cipher. If the client sends the extension as part of the ClientHello message, the test case selects a cipher suite using an AEAD cipher and includes this extension. Every implementation that requested this extension accepts the ServerHello message and continues the handshake normally. Since this extension has no effect on AEAD cipher suites, this test case shows that the clients do not perform a sanity check on the ServerHello message.

Expected different Behavior

Encrypt-then-MAC Extension. As for the server tests, a test case for the clients exists as well that checks if a client supports the extension and therefore offers a possibility to mitigate certain attacks. Botan, GnuTLS, mbed TLS, OpenSSL and wolfSSL support the extension.

RC4 Cipher Suites. As already mentioned in Section 4.2.1.2, the client implementations of LibreSSL and NSS offer RC4 cipher suites. This is a behavior that is not expected from any client since the RC4 cipher is insecure [53].

4.2.1.6 Client (TLS 1.3)

Expected Alert $\neq$ Received Alert

Select unsupported Cipher Suite. If the server selects a cipher suite that was not offered by the client, the client must terminate the handshake by sending an `illegal_parameter` fatal alert [25, Section 4.1.3]. The test case sends a GREASE value as a cipher suite. GnuTLS is the only implementation that responds with the wrong alert description, that is a `handshake_failure` alert.

ServerHello – Random. If a TLS 1.3 server chooses to select TLS 1.2 it must set the last 8 bytes of the server random value to a specific value. A TLS 1.3 client must check these bytes and terminate the handshake with an `illegal_parameter` alert [25, Section 4.1.3]. The test case is only executed when the tested client supports TLS 1.3 and

TLS 1.2, selects TLS 1.2, sets the correct value for the server random and expects an alert message. The NSS client implementation does not check the 8 bytes of the server random and continues the handshake normally. This seems to be an issue of the default configuration. The downgrade check is enabled since Firefox 72. It was disabled before because of problems with middleboxes that do not support TLS 1.3, but forward the ServerHello random value from the server [56].

ServerHello – Session ID. TLS 1.3 requires that the session id in the ServerHello is equal to the session id in the ClientHello message [25, Section 4.1.3]. If the session id in the ServerHello is different the client must abort the handshake with an `illegal_parameter` alert. The implementations show two different behaviors. GnuTLS and MatrixSSL do not check the session id and continue normally. BoringSSL responds with a different alert, a `decode_error` fatal alert.

ServerHello – Extensions. Since TLS 1.3 supports extensions to be present in multiple places like the ServerHello and the EncryptedExtension message, the TLS 1.3 client must check the extensions in both messages and send an `illegal_parameter` alert if an extension is sent in the wrong message [25, Sections 4.2, 4.3.1]. One test case adds the heartbeat extension to the ServerHello message, that is usually part of the EncryptedExtensions message. LibreSSL, MatrixSSL and OpenSSL continue with the handshake normally.

Encrypted Extensions. The EncryptedExtensions message is covered by two tests. One test adds the padding extension to the message, another test adds the `supported_versions` extensions to this message. The `padding` extension is usually part of the ClientHello and only sent by the client. The `supported_versions` extension is only allowed in the ClientHello and ServerHello messages. If the `supported_versions` extension is added by the test case to the Encrypted Extensions message, BoringSSL, GnuTLS and NSS respond with a wrong `unsupported_extension` instead of an `illegal_parameter` fatal alert. If the `padding` extension is sent, GnuTLS, LibreSSL and MatrixSSL ignore the extension and continue with the handshake. This indicates that these implementations do not check the extensions of the EncryptedExtensions message for extensions that are only sent by the client. In comparison to the previous test case, the behavior of GnuTLS and OpenSSL is not consistent.

Expected Handshake Termination

Zero-Length Fragment – Handshake Record (ServerHello, Finished). Zero-length record layer fragments are also explicitly excluded for TLS 1.3 by the RFC [25, Section 5.1]. The test cases send zero-length fragments in front of the ServerHello message and the Finished message and expect a termination of the handshake. BoringSSL and NSS

ignore the fragments in both cases. OpenSSL only continues with the handshake when the fragment is in front of the ServerHello message.

Expected different Behavior

Supported Versions Extension. According to the RFC, clients must ignore the version field of the ServerHello message if the supported version extension is present [25, Section 4.2.1]. The test case sets the version field to an invalid value of `0x05 0x05` but sends the `supported_versions` extension as required to perform a TLS 1.3 handshake. Only MatrixSSL and OpenSSL continue the handshake as expected. The other implementations respond with fatal `protocol_version` or `illegal_parameter` fatal alert. The RFC is not consistent here for client implementations, since it also defines for the ServerHello message that the version field must be set to `0x03 0x03` [25, Section 4.1.3].

4.2.1.7 Client and Server (TLS 1.2)

This subsection describes results of test cases that are implemented once but executed for client and server tests. If the result refers to an implementation where the client and server are part of the evaluation it is written in brackets behind the implementation which network peer is affected. For example, GnuTLS (S) refers to the server and GnuTLS (C) to the client. If only the server or the client could be tested, the brackets are omitted.

Expected Alert $\neq$ Received Alert

Application Data – AEAD Cipher. After the handshake is completed, test cases are implemented that send application data and modify the ciphertext and authentication tag to observe the reaction of the other peer that receives data that it can not decrypt. The RFC requires that a `bad_record_mac` must be sent if such an event happens. Although the implementations react consistently in both test cases, there are implementations where the client reacts differently than the server. BearSSL, GnuTLS (C), Rustls and s2n (S) close the TCP socket instead of sending a `bad_record_mac` fatal alert. The s2n client also fails at the test cases but it does not close the TCP socket or respond with an alert message. The other implementations including the GnuTLS server send the correct alert message.

Application Data – CBC Cipher. The approach described above is also tested for CBC cipher suites. Instead of modifying the authentication tag and the ciphertext, the padding and MAC are modified. If the result between both test cases would be different, the implementation would be vulnerable to the padding oracle attack. Although these test

cases check a property that lead to attacks in the past, not all implementations respond with the correct `bad_record_mac` fatal alert message. BearSSL, GnuTLS (C), s2n (S) and WolfSSL close the TCP socket. s2n (C) does not respond at all, neither with a TCP FIN packet to close the socket nor with an alert message.

Skip CCS. The RFC requires that the ChangeCipherSpec message is sent before the Finished message. If this is not the case, a fatal alert must be sent [37, Section 7.4.9]. s2n (C), WolfSSL and BearSSL close the TCP socket without sending an alert message. This time the server of s2n does not send an alert or closes the socket.

Expected Handshake Termination

CCS Message. The ChangeCipherSpec message contains only a single byte set to 1. The test case checks if the implementations check this condition. This is covered by two test cases. One test case sends a different value for the single-byte and the other test case sends two bytes instead of one. GnuTLS (S), WolfSSL and Rustls does not verify the value of the single byte. If multiple bytes are sent Rustls is the only implementation that accepts the longer CCS message and continues with the handshake. In both cases the other implementations react either with closing the TCP socket or they respond with a fatal alert message.

Zero-Length Fragment – CCS Record. Sending a zero-length CCS record is not allowed by the RFC. The test case sends a zero-length CCS record followed by the CSS message in a separate record. In contrast to the previous test cases only LibreSSL (S/C), OpenSSL (S/C) and BearSSL accept such a message and continue with the handshake.

Expected different Behavior

Zero-Length Fragment – Application Record. According to the RFC it is allowed to send zero-length record layer packets that use the application content type [37, Section 6.2.1]. Although this is allowed, WolfSSL closes the connection. This behavior can have a negative impact on the interoperability between the WolfSSL client and another server that sends such packets.

4.2.1.8 Client and Server (TLS 1.3)

Expected Alert $\neq$ Received Alert

Decryption failure. Similarly to TLS 1.2, the TLS 1.3 specification requires that a `bad_record_mac` fatal alert is sent if the decryption fails. This is covered by two test cases

that modify the authentication tag and the cipher text. In both cases all the implementations react consistently. Rustls and GnutTLS close the TCP socket. LibreSSL responds with a `close_notify` warning alert before closing the socket. The other implementations that support TLS 1.3 respond with the correct alert.

Invalid Record Layer Content-Type. The implemented test case changes the content-type of the record layer for the first sent message to `0xFF`. That is either the ClientHello for client tests or the ServerHello for server tests. If an implementation receives a record layer with an unknown content-type "it MUST terminate the connection with an 'unexpected_message' alert" [25, Section 5]. 3 implementations respond differently. NSS sends an `illegal_parameter` fatal alert, tlslite-ng a `record_overflow` alert and Rustls closes the TCP socket.

Too large Record Layer. In contrast to TLS 1.2, TLS 1.3 requires that a `record_overflow` fatal alert message must be sent if the record layer is larger than $2^{14} + 256$ bytes. The test case sends an application message after the handshake with a size of $2^{14} + 266$ bytes, so that the message is larger than the limit. GnuTLS is the only implementation that does not conform to the specification and closes the socket instead of sending the alert.

Expected Handshake Termination

Record Layer Version. The RFC specifies that "implementations MUST NOT send any records with a version less then 0x0300. Implementations SHOULD NOT accept any records with a version less than 0x0300 (but may inadvertently do so if the record version number is ignored completely)" [25, Section D.5.]. The test case sends a ClientHello/ServerHello message inside a record layer with the version `0x02 0x03`. NSS, MatrixSSL and tlslite-ng ignore the version number and continue as a regular TLS 1.3 handshake. Thus, all implementations pass the test but the behavior is inconsistent.

Expected different Behavior

There is no test implemented in the testsuite targeting server and client implementations that fits into this category. TLS 1.3 tests that belong to this category are part of Section 4.2.1.6 and Section 4.2.1.4.

4.2.1.9 Other Observations

TLS 1.3 – Encrypted Alert Messages. The client tests have shown that the implementations behave differently regarding the encryption of alert messages. If the server testing

the client injects a failure after the ServerHello message, the client can already calculate the encryption keys and encrypt the alert message reporting the failure. MatrixSSL and OpenSSL do not encrypt alert messages at this point in time. The other implementations send encrypted messages although. BoringSSL and GnuTLS send a CCS message before the encrypted alert. In contrast to that, LibreSSL and NSS send the alert message without CCS before.

LibreSSL – Fatal alerts. If the client implementation of LibreSSL sends a fatal alert it is often followed by a `close_notify` warning alert. This behavior does not conform to the specification. According to the RFC fatal alert messages "result in the immediate termination of the connection" [37, Section 7.2].

4.2.2 Same Implementation - Multiple Versions

In this section multiple versions of the same implementation are evaluated. The development of the score and the analysis of the tests where the results changed, show which parts of an implementation were updated. The following sections limit the analysis of version histories to the most interesting ones.

The analyzed versions of the NSS and tlslite-ng implementations only include the latest patch version of every minor release. If, for example, the versions 3.28, 3.28.1 up to 3.28.5 are available, the data set only includes version 3.28.5. This reduces the amount of data that is needed to be analyzed. Since the patch releases do not include any new major features, the impact of the scores between patch versions is small and still visible in comparison with a following minor release that includes theses patches as well.

4.2.2.1 tlslite-ng Server

The development of the score of the tlslite-ng server shows in Figure 4.6 two points where it changes. Although there is not a lot of dynamic in this graph, the results are still interesting.

The first increase is caused by the update from version 0.5.2 to 0.6.0 that changes the result of four test cases. The versions up to 0.6.0 do not pass the three tests completely that invalidate the MAC, padding and ciphertext of CBC encrypted data. These three test cases succeed in general except for the case when the server selects the `TLS_DHE_RSA_WITH_AES_128_CBC_SHA` cipher suite. The fourth test sends a TLS record that exceeds the size limitation. The result changes here from `PARTIALLY_FAILED` to `SUCCEEDED`. Up to version 0.6.0, the test case fails for every cipher suite except the one mentioned before, where the implementation closes the socket. From version 0.6.0 on all of the mentioned test cases succeed.

The second change of the graph occurs from version 0.7.5 to the latest available version 0.8.0-alpha38. The newest version is the first version supporting TLS 1.3. Since multiple TLS 1.3 test cases fail, the score drops by around 1%. It is interesting to note that version 0.8.0-alpha38 adds support for 3DES cipher suites, although the versions before did not support those. This also causes a test checking for supported deprecated cipher suites to fail and therefore lowers the score.

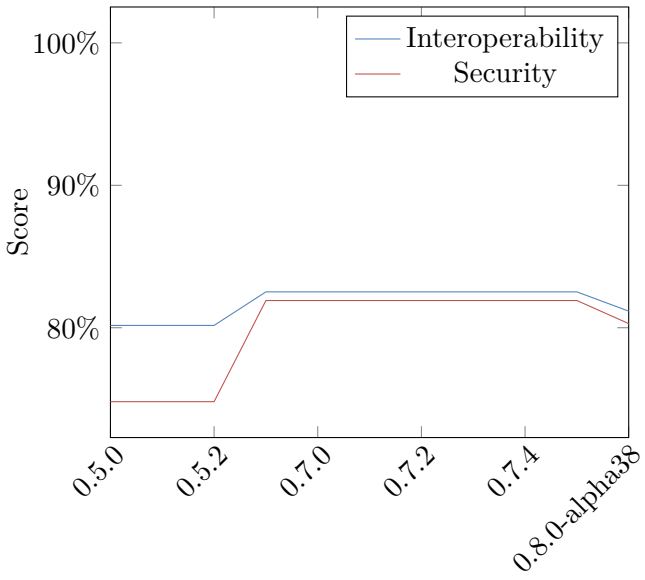


Figure 4.6: Development of the testsuite scores for the tlslite-ng server.

4.2.2.2 Botan Client

The development of the score for the client implementation of Botan is visible in Figure 4.7. The test results of this implementation show an interesting behavior. From version 1.11.9 to 1.11.21 the test case sending a zero-length application record fails for cipher suites that use an AEAD cipher. From version 1.11.22 this high severity interoperability test fails completely which results in a loss of around 1% of the interoperability score. This change is visible in Figure 4.7.

The update from version 1.11.29 to 2.0.0 lets this test case succeed and adds around 3% on top of the interoperability score. The security scores rises as well, since Botan adds support for the encrypt-then-MAC extension, which results in a successful medium severity security test.

The next bigger change can be observed with the update from version 2.4.0 to 2.5.0. This update causes a high severity interoperability test to succeed that sends a GREASE extension with the ServerHello message to the client. Before version 2.5.0 Botan accepts this extension which causes the test to fail. After the update, the extension is rejected

as expected since the client did not request it. This test also adds a few points to the security score that causes it to rise as well.

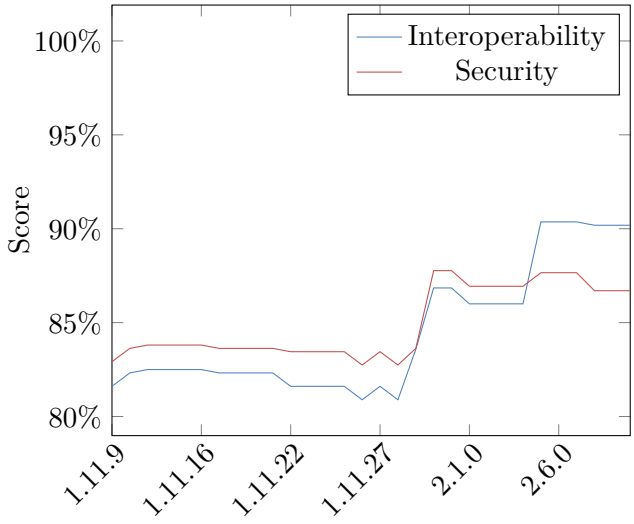


Figure 4.7: Development of the testsuite scores for the Botan client.

4.2.2.3 NSS Client

The scores of the NSS client also see an improvement over time (Figure 4.8). Noticeable is the drop in the beginning from version 3.28.1 to 3.29.5. The older version passes the test case that sends a zero-length ServerHello message because it terminates the handshake by closing the socket. All of the newer versions fail at this test since they ignore the zero-length fragment and continue with the handshake.

The update from version 3.33 to 3.34.1 results in a higher interoperability score because the implementation changed the handling of records that are too large. Instead of ignoring those records and failing the test, they are rejected since version 3.34.1 with the correct `record_overflow` fatal alert.

The same update also changes the behavior when the server selects an unsupported TLS version. Instead of sending a `protocol_version` alert, the client responds with a `decode_error` after the update. This change was reverted in the next update to version 3.35. Therefore, the score reaches a plateau. Two versions later it drops again because the behavior of the same test case changed again. In the versions after 3.37.3, the client responds with a `illegal_parameter` fatal alert, instead of the specified `protocol_version` alert.

With the release of version 3.39 both scores increase again. This version introduces support for TLS 1.3. Since most of the TLS 1.3 test cases complete successfully, the score increases a lot. The same update also changes the behavior of the client when

the server selects TLS 1.2 and sends an invalid signature algorithm identifier in the ServerKeyExchange message. Instead of just closing the TCP socket, the client responds with an `illegal_parameter` fatal alert. This behavior is also already visible since version 3.37.3. In versions 3.37.3 and 3.38 the client only responds with the alert if the server selected a non-`TLS_DHE` cipher suite.

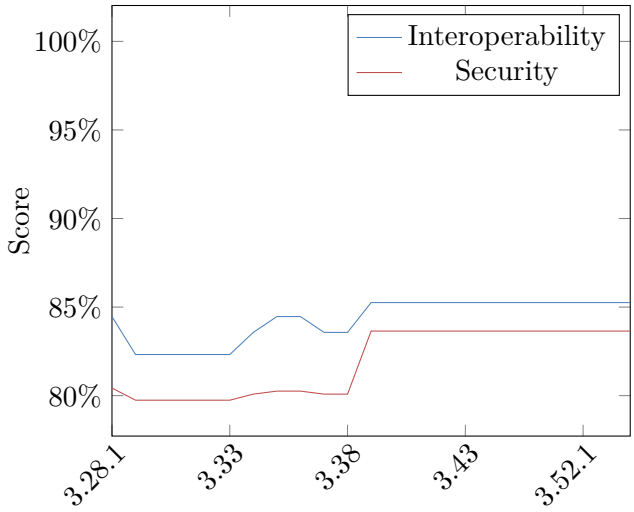


Figure 4.8: Development of the testsuite scores for the NSS client.

5 Conclusion

The result of this thesis is a developed testsuite and framework that allows to model test cases for the TLS handshake based on the specification. Due to the usage of JUnit, the framework can be extended with new JUnit extensions to be able to model more specific TLS tests. The workflow for the development of further test cases also highly benefits from JUnit because of the GUI support for many IDEs. This allows executing a single test during development without an overhead. The flexibility of the framework together with the features of TLS-Attacker allow to create and execute TLS handshake sequences in a convenient way. Even more important is the validation of the received messages from the other peer. The framework offers the necessary APIs to be able to model even complex validation procedures.

The evaluation shows that the developed testsuite can find violations of the specification. In most cases, the implementations respond with different alert messages than specified. While this behavior is in most cases only a violation of the specification, sending different alerts could lead in specific scenarios to oracles that might result in side-channels that can be used for attacks. Not even the sending of a `bad_record_mac` alert message when the padding or MAC of CBC encrypted data is invalid is implemented correctly across all implementations, although this is important to prevent the basic padding oracle attack [19]. However, such a padding oracle could not be found during the evaluation, since the behavior of the implementations between those two test cases was always consistent. Other implementations terminate the handshake while they should not. This is the case for LibreSSL and s2n when they receive unknown signature and hash algorithms in the ClientHello message.

The implemented tests can also reveal the inner workings of an implementation. There is, for example, the LibreSSL client that ignores an `user_cancelled` fatal alert that is sent before the ServerHello message, but terminates the handshake when it receives every other fatal alert. In contrast to that, if LibreSSL receives a `user_cancelled` fatal alert in front of the ServerHelloDone message it terminates the handshake as expected. Other implementations, like OpenSSL, GnuTLS and mbed TLS, terminate the handshake when they receive a known extension in the ServerHello that they did not request, but continue the handshake if they receive an unknown GREASE extension.

The evaluation of multiple historic versions of the same implementation shows that the score of the testsuite changes over time. Further, it can be determined when certain features of the implementation are changed, like the support of new extensions or the handling of unknown algorithms. Although the newest versions of the implementations do

not show different behaviors in the same test case depending on the negotiated cipher suite, this is, for example, observed in older versions of Botan. This proves that the testsuite can detect such behaviors using the test derivation feature, but only in very few occasions. If the needed time to perform a testrun is important, having this feature enabled costs more than the benefit of the derivation feature is.

The development of the testsuite and the results also showed that implementing a TLS stack is a very complex task. Especially for TLS 1.2, since the specification is spread over multiple RFCs. Often the information in the RFC about a specific element of the protocol is not available in a single place, so that a developer must combine multiple sections to be able to implement the correct behavior for every case. Another problem is that the RFC specifies that something MUST happen, but it does not elaborate why this is a requirement. This makes it extremely difficult to estimate the consequences that can occur if the specification is not correctly implemented. Elaborating more about the consequences would probably result in a higher awareness for the problems at the developer's side and a higher motivation to develop an implementation that conforms to the specification in its entirety.

5.1 Future Work

The tests contained in the testsuite are not covering every aspect of the specifications. Features like session resumption, 0-RTT handshakes, early data, client authentication or certificate validation are currently not covered. To get an even better understanding of the implementation's behavior, tests covering those features should be implemented. For the development of more consistent tests, it might be better to put more effort into the process finding a formal way to be able to model test cases based on the RFC.

Another goal should be to make the tests even tighter. This could be achieved, for example, by validating every value in every received message automatically. That would reveal implementations that use invalid values together with valid values in their messages, for example, a cipher suite that is part of the ClientHello cipher suite list but not specified at all.

Furthermore, the edge cases that prevent the testsuite and the TLS-Scanner from performing a TLS handshake with implementations should be eliminated. These can occur because the parameters of a message are not accepted by the implementation, although the message itself conforms to the specification. Another kind of edge cases are those that lead to false-negative test results. If an implementation sends an alert message, closes the socket and TLS-Attacker tries to send another message, it receives an RST TCP packet. The following receive action of the workflow is not able to receive the alert message anymore. The problem is that this message might be needed to flag the test case as successfully completed. This issue only appeared in a few test cases and was only detected during the final evaluation of the results.

Considering the test results, looking into the extension handling of clients seems to be promising. As shown there are client implementations available that do not terminate the handshake when they receive an extension that they did not request in their ClientHello message. It should be validated if the clients just ignore the extension or react to the injected extension as if the client had requested it.

The length field tests still need to be evaluated. None of those test cases succeeded for any implementation. The tests modify every available length field in every sent message. This means that there are also length fields modified that are part of the TLS-Attacker message class but only serialized when a specific protocol version is negotiated. The `cookieLength` field of the `ClientHelloMessage` class, for example, is only sent if DTLS is used. When the test case modifies such a field and performs the handshake, the handshake will always succeed for non-DTLS protocol versions, but the test case fails. This makes the analysis of these tests very time-consuming. However, analyzing the performed handshakes for multiple versions of the same implementation could give insights how the message parsers have changed over time.

This thesis only evaluated the example server and client applications provided by the implementations. Using the testsuite against real-world TLS implementations, for example, middleboxes or web-servers may result in interesting and different outcomes than described in this thesis.

List of Figures

List of Tables

Bibliography

[1] Hanno Böck, Juraj Somorovsky, and Craig Young. "Return Of Bleichenbacher's Oracle Threat (ROBOT)". In: *27th USENIX Security Symposium, USENIX Security 2018, Baltimore, MD, USA, August 15-17, 2018*. Ed. by William Enck and Adrienne Porter Felt. USENIX Association, 2018, pp. 817–849. URL: https://www.usenix.org/conference/usenixsecurity18/presentation/bock.

[2] N. J. Al Fardan and K. G. Paterson. "Lucky Thirteen: Breaking the TLS and DTLS Record Protocols". In: *2013 IEEE Symposium on Security and Privacy*. May 2013, pp. 526–540.

[3] Karthikeyan Bhargavan and Gaëtan Leurent. "On the Practical (In-)Security of 64-bit Block Ciphers: Collision Attacks on HTTP over TLS and OpenVPN". In: *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, October 24-28, 2016*. Ed. by Edgar R. Weippl et al. ACM, 2016, pp. 456–467. DOI: 10.1145/2976749.2978423. URL: https://doi.org/10.1145/2976749.2978423.

[4] *This POODLE Bites: Exploiting TheSSL 3.0 Fallback*. URL: https://www.openssl.org/~bodo/ssl-poodle.pdf.

[5] Thai Duong and Juliano Rizzo. *Here come the XOR ninjas*. 2011. URL: http://www.hpcc.ecs.soton.ac.uk/dan/talks/bullrun/Beast.pdf.

[6] Juliano Rizzo and Thai Duong. "The CRIME attack". In: *ekoparty security conference*. Vol. 2012. 2012.

[7] Daniel Bleichenbacher. "Chosen Ciphertext Attacks Against Protocols Based on the RSA Encryption Standard PKCS #1". In: *Advances in Cryptology - CRYPTO '98, 18th Annual International Cryptology Conference, Santa Barbara, California, USA, August 23-27, 1998, Proceedings*. Ed. by Hugo Krawczyk. Vol. 1462. Lecture Notes in Computer Science. Springer, 1998, pp. 1–12. DOI: 10.1007/BFb0055716. URL: https://doi.org/10.1007/BFb0055716.

[8] *The Heartbleed Bug*. URL: https://heartbleed.com/.

[9] *Technical Advisory – wolfSSL TLS 1.3 Client Man-in-the-Middle Attack (CVE-2020-24613)*. URL: https://research.nccgroup.com/2020/08/24/technical-advisory-wolfssl-tls-1-3-client-man-in-the-middle-attack/.

[10] *tlsfuzzer*. URL: https://github.com/tomato42/tlsfuzzer.

[11] *Achelos TLS Inspector*. URL: https://www.achelos.de/de/tls-inspector.html.

[12] *TLS-Scanner*. URL: https://github.com/RUB-NDS/TLS-Scanner.

[13] *testssl.sh*. URL: https://testssl.sh/.

[14] *Qualys SSl Labs*. URL: https://www.ssllabs.com/.

[15] *How's My SSL*. URL: https://www.howsmyssl.com/.

[16] Nimrod Aviram et al. "DROWN: Breaking TLS Using SSLv2". In: *25th USENIX Security Symposium (USENIX Security 16)*. Austin, TX: USENIX Association, Aug. 2016, pp. 689–706. ISBN: 978-1-931971-32-4. URL: https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/aviram.

[17] *TLS Attacker*. URL: https://github.com/RUB-NDS/TLS-Attacker.

[18] Malena Ebert. "TLS-Compliance: Erstellen einer Testsuite für TLS-Bibliotheken mit TLS-Attacker". Ruhr-Universität Bochum, 2018.

[19] Serge Vaudenay. "Security Flaws Induced by CBC Padding - Applications to SSL, IPSEC, WTLS ..." In: *Advances in Cryptology - EUROCRYPT 2002, International Conference on the Theory and Applications of Cryptographic Techniques, Amsterdam, The Netherlands, April 28 - May 2, 2002, Proceedings*. Ed. by Lars R. Knudsen. Vol. 2332. Lecture Notes in Computer Science. Springer, 2002, pp. 534–546. DOI: 10.1007/3-540-46035-7_35. URL: https://doi.org/10.1007/3-540-46035-7%5C_35.

[20] *RFC 6066 - Transport Layer Security (TLS) Extensions: Extension Definitions*. URL: https://tools.ietf.org/html/rfc6066.

[21] *RFC 7366 - Encrypt-then-MAC for Transport Layer Security (TLS) and Datagram Transport Layer Security (DTLS)*. URL: https://tools.ietf.org/html/rfc6066.

[22] *RFC 7685 - A Transport Layer Security (TLS) ClientHello Padding Extension*. URL: https://tools.ietf.org/html/rfc7685.

[23] *RFC 8422 - Elliptic Curve Cryptography (ECC) Cipher Suites for Transport Layer Security (TLS) Versions 1.2 and Earlier*. URL: https://tools.ietf.org/html/rfc8422.

[24] *RFC 4492 - Elliptic Curve Cryptography (ECC) Cipher Suites for Transport Layer Security (TLS)*. URL: https://tools.ietf.org/html/rfc4492.

[25] *RFC 5246 - TLS Protocol Version 1.2*. URL: https://tools.ietf.org/html/rfc8446.

[26] *ModifiableVariable*. URL: https://github.com/RUB-NDS/ModifiableVariable.

[27] *What is a Container?* URL: https://www.docker.com/resources/what-container.

[28] *Open Container Initiative*. URL: https://opencontainers.org/.

[29] *Podman*. URL: https://podman.io/.

[30] ***docker run** Reference*. URL: https://docs.docker.com/engine/reference/commandline/run/.

[31] *Docker Volumes*. URL: https://docs.docker.com/storage/volumes/.

[32] *`docker build` Reference*. URL: https://docs.docker.com/engine/reference/commandline/build/.

[33] *TLS-Docker-Library*. URL: https://github.com/RUB-NDS/TLS-Docker-Library.

[34] *Maven Repository: Testing Frameworks*. URL: https://mvnrepository.com/open-source/testing-frameworks.

[35] *TLS-Testsuite GitHub Repository*. URL: https://github.com/RUB-NDS/TLS-Testsuite.

[36] *TLS-Test-Framework GitHub Repository*. URL: https://github.com/RUB-NDS/TLS-Test-Framework.

[37] *RFC 5246 - TLS Protocol Version 1.2*. URL: https://tools.ietf.org/html/rfc5246.

[38] *RFC 6167 - Prohibiting Secure Sockets Layer (SSL) Version 2.0*. URL: https://tools.ietf.org/html/rfc6167.

[39] *RFC 7465 - Prohibiting RC4 Cipher Suites*. URL: https://tools.ietf.org/html/rfc7465.

[40] *RFC 7507 - TLS Fallback Signaling Cipher Suite Value (SCSV) for Preventing Protocol Downgrade Attacks*. URL: https://tools.ietf.org/html/rfc7507.

[41] *RFC 7568 - Deprecating Secure Sockets Layer Version 3.0*. URL: https://tools.ietf.org/html/rfc7568.

[42] *Applying GREASE to TLS Extensibility*. URL: https://tools.ietf.org/html/draft-ietf-tls-grease-04.

[43] *Vue.js*. URL: https://vuejs.org/.

[44] *Node.js*. URL: https://nodejs.org/en/.

[45] *Express*. URL: https://expressjs.com/de/.

[46] *mongoDB*. URL: https://www.mongodb.com/de.

[47] *Wireshark*. URL: https://www.wireshark.org/.

[48] *tcpdump*. URL: https://www.tcpdump.org/.

[49] *Docker BuildKit*. URL: https://www.docker.com/blog/advanced-dockerfiles-faster-builds-and-smaller-images-using-buildkit-and-multistage-builds/.

[50] *Spotify Docker-Client on GitHub*. URL: https://github.com/spotify/docker-client.

[51] *TLS-Testsuite-Large-Scale-Evaluator*. URL: https://github.com/RUB-NDS/TLS-Testsuite-Large-Scale-Evaluator.

[52] *RFC 7301 - Application-Layer Protocol Negotiation Extension*. URL: https://tools.ietf.org/html/rfc7301.

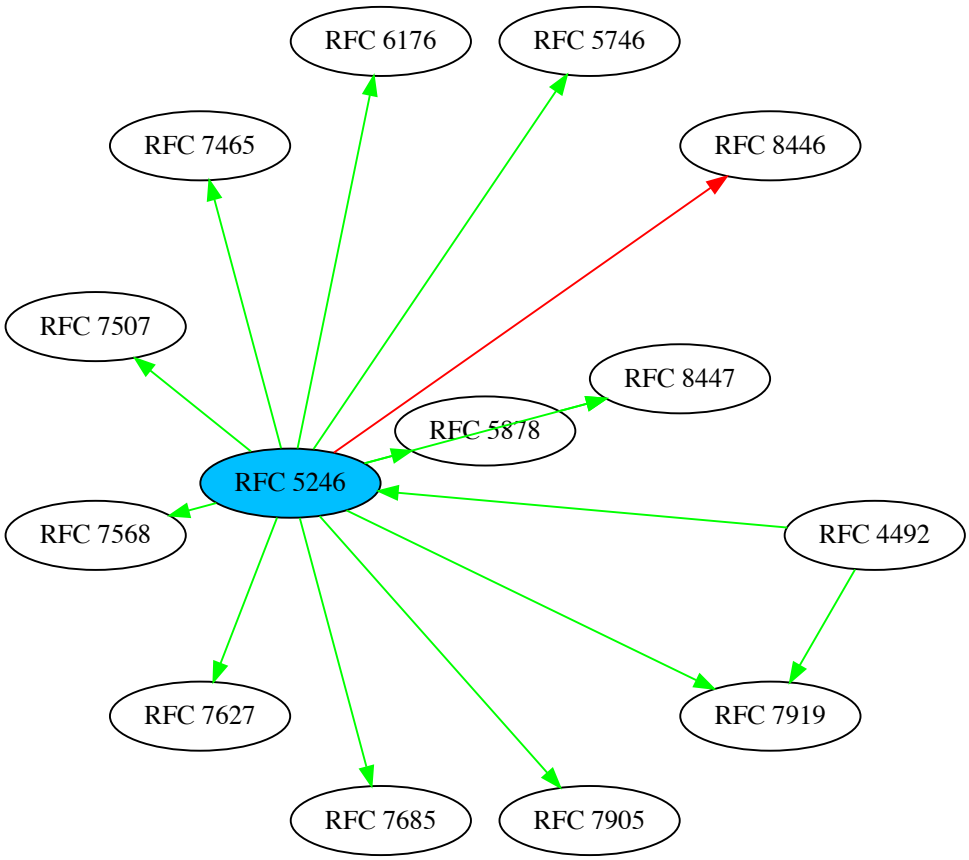
[53] Nadhem J. AlFardan et al. “On the Security of RC4 in TLS”. In: *Proceedings of the 22th USENIX Security Symposium, Washington, DC, USA, August 14-16, 2013*. Ed. by Samuel T. King. USENIX Association, 2013, pp. 305–320. URL: https://www.usenix.org/conference/usenixsecurity13/technical-sessions/paper/alFardan.

[54] Marc Stevens et al. “The First Collision for Full SHA-1”. In: *Advances in Cryptology - CRYPTO 2017 - 37th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 20-24, 2017, Proceedings, Part I*. Ed. by Jonathan Katz and Hovav Shacham. Vol. 10401. Lecture Notes in Computer Science. Springer, 2017, pp. 570–596. DOI: 10.1007/978-3-319-63688-7_19. URL: https://doi.org/10.1007/978-3-319-63688-7%5C_19.

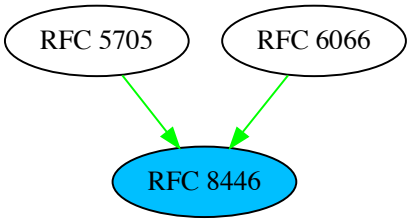
[55] *Deprecating MD5 and SHA-1 signature hashes in TLS 1.2*. URL: https://tools.ietf.org/html/draft-ietf-tls-md5-sha1-deprecate-03.

[56] *Enable TLS downgrade sentinel detection*. URL: https://bugzilla.mozilla.org/show_bug.cgi?id=1576790.

A Figures



(a) TLS 1.2



(b) TLS 1.3

Figure A.1: Network of RFCs. $x \longrightarrow y$ denotes that x is updated (green) or obsoleted (red) by y.

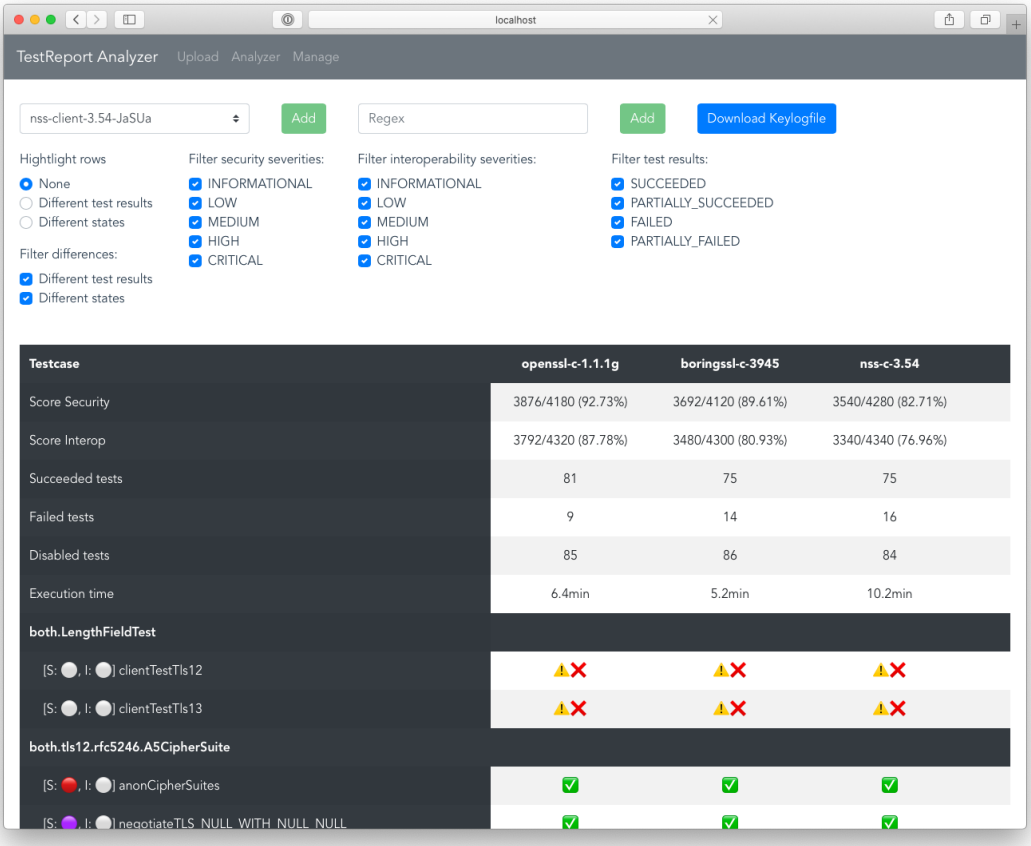


Figure A.2: Analyzer view of the test report analyzer, showing the results of three implementations.

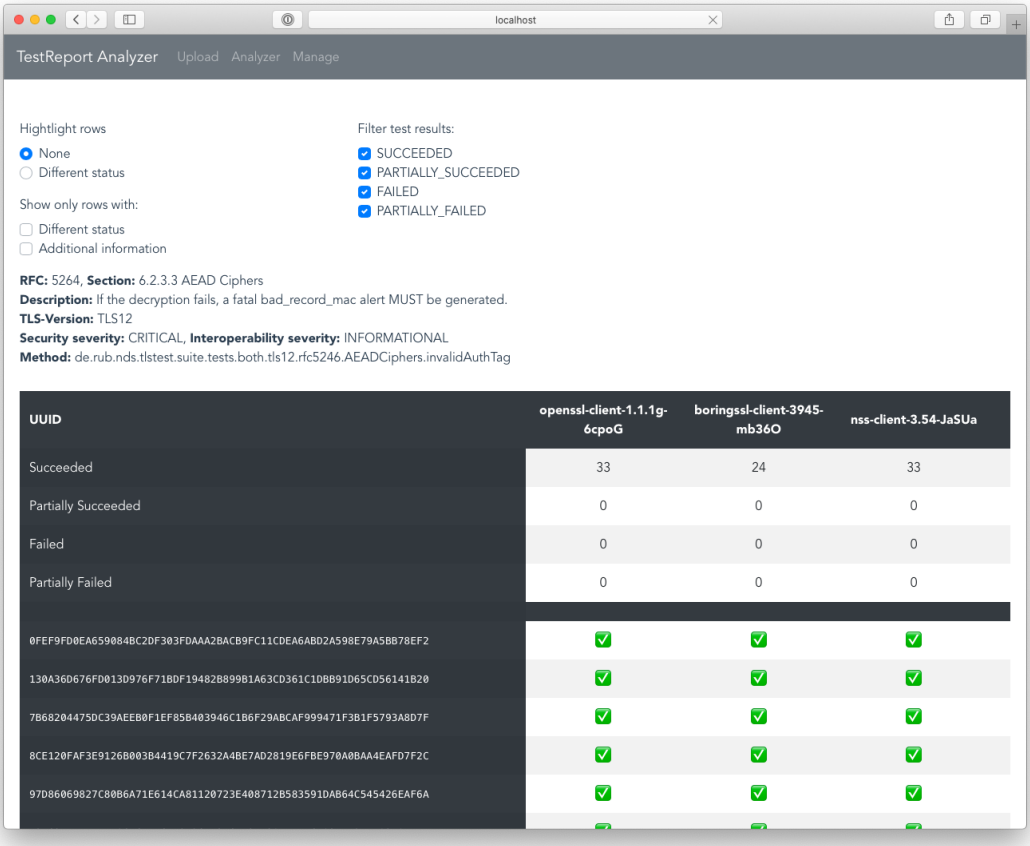


Figure A.3: State view of the test report analyzer, showing the handshakes of a single test case for three implementations.

B Tables

		Supported Cipher Suites							Supported Versions			
		RC4	3DES	MD5	CBC	AEAD	Non-PFS	PFS	TLS 1.0	TLS 1.1	TLS 1.2	TLS 1.3
BoringSSL	S		✓		✓	✓	✓	✓	✓	✓	✓	✓
	C		✓		✓	✓	✓	✓	✓	✓	✓	✓
OpenSSL	S				✓	✓	✓	✓	✓	✓	✓	✓
	C				✓	✓	✓	✓	✓	✓	✓	✓
NSS	S	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
	C	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
s2n	S				✓	✓	✓	✓	✓	✓	✓	
	C				✓	✓	✓	✓	✓	✓	✓	
mbed TLS	S				✓	✓	✓	✓	✓	✓	✓	
	C				✓	✓	✓	✓	✓	✓	✓	
Rustls	S					✓		✓			✓	✓
Botan	S					✓		✓			✓	
	C					✓		✓	✓	✓	✓	
LibreSSL	S	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	
	C	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
wolfSSL	C					✓		✓			✓	
GnuTLS	S				✓	✓	✓	✓	✓	✓	✓	✓
	C				✓	✓	✓	✓	✓	✓	✓	✓
MatrixSSL	C					✓		✓	(✓)	(✓)	(✓)	✓
tlslite-ng	S		✓		✓	✓	✓	✓	✓	✓	✓	✓
BearSSL	S		✓		✓	✓	✓	✓	✓	✓	✓	

Table B.1: Default configurations of the analyzed server (S) and client (C) implementations. Empty fields indicate that a implementation does not support the property.

		TLS 1.2			TLS 1.3		
Categories		α	β	γ	α	β	γ
# Failed tests		10 S, 8 C	12 S, 9 C	6 S, 3 C	8 S, 13 C	3 S, 2 C	0 S, 2 C
BoringSSL	S		✗ (3)		✗ (3)	✗ (3)	
	C		✗ (1)	✗ (1)	✗ (4)	✗ (2)	
OpenSSL	S	✗ (2)	✗ (4)	✗ (1)	✗ (2)	✗ (2)	
	C	✗ (1)	✗ (3)		✗ (1)	✗ (1)	
NSS	S	✗ (4)	✗ (4)		✗ (2)	✗ (3)	
	C	✗ (3)	✗ (1)	✗ (2)	✗ (5)	✗ (2)	
s2n	S	✗ (8)	✗ (6)	✗ (1)	–	–	–
	C	✗ (8)	✗ (3)	✗ (1)	–	–	–
mbed TLS	S	✗ (2)	✗ (2)	✗ (1)	–	–	–
	C	✗ (3)	✗ (1)		–	–	–
Rustls	S	✗ (2)	✗ (6)	✗ (2)	✗ (6)	✗ (3)	
Botan	S	✗ (2)	✗ (2)	✗ (1)	–	–	–
	C	✗ (2)	✗ (2)		–	–	–
LibreSSL	S	✗ (1)	✗ (5)	✗ (1)	–	–	–
	C	✗ (1)	✗ (2)	✗ (2)	✗ (5)		
wolfSSL	C	✗ (7)	✗ (2)	✗ (1)	–	–	–
GnuTLS	S	✗ (1)	✗ (4)	✗ (1)	–	–	–
	C	✗ (3)	✗ (1)		✗ (7)		
MatrixSSL	C				✗ (6)		
tlslite-ng	S	✗ (2)	✗ (3)	✗ (2)	✗ (2)		
BearSSL	S	✗ (6)	✗ (4)	✗ (1)	–	–	–

Category α: Expected Alert $\neq$ Received Alert
Category β: Expected Handshake Termination
Category γ: Expected different Behavior

Table B.2: Overview of the test result evaluation. It shows how many tests belonging to a certain test category (A, B, C) and TLS version failed. Empty fields indicate that all tests passed.