Verifying Security Protocol Implementations with Refinement Types

The miTLS case study

Alfredo Pironti – IOActive
alfredo.pironti@ioactive.com

FOSAD 2016 Summer School
Aug 29 – Sep 3 2016

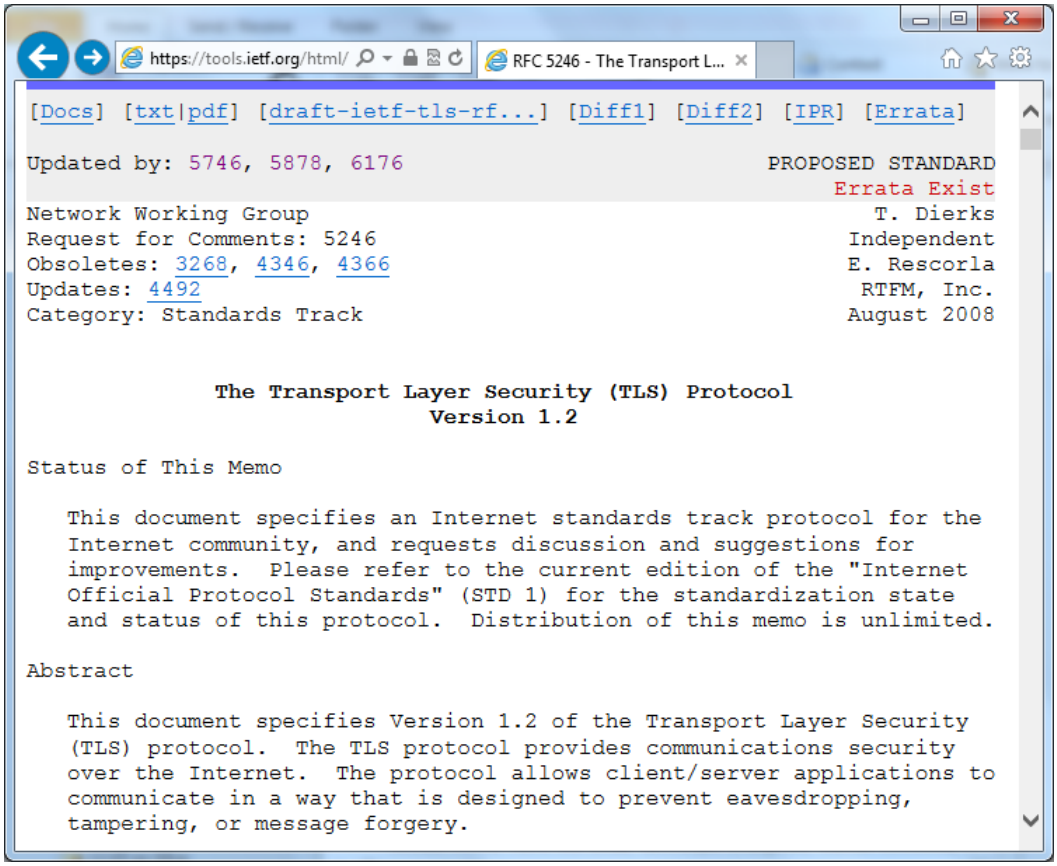
Transport Layer Security (1995—)

The most widely deployed cryptographic protocol?

HTTPS, 802.1x (EAP),
FTPS, VPN, mail, VoIP, …

19 years of attacks, fixes, and extensions

1995 – Netscape's Secure Sockets Layer
1995 – SSL2
1996 – SSL3
1999 – TLS1.0 (RFC2246, ≈SSL3)
2006 – TLS1.1 (RFC4346)
2008 – TLS1.2 (RFC5246)



Many implementations

- SChannel, OpenSSL, NSS, GnuTLS, JSSE, PolarSSL, …
- **Several security patches every year**

Many papers

- **Well-understood, detailed specs**
- **Security theorems… mostly for small simple models of TLS**

Still hot topic in practice and theory

- At least 11 research papers in the last 3 years
  - Of which 5 describing attacks
- Several flaws found in the last 2 years
  - Protocol logic (e.g. Alert attack, Triple Handshake, Logjam)
  - Implementation specific (e.g. goto fail; Heartbleed; CCS injection; ClientHello fragmentation, FREAK)
  - Cryptography (RC4, 3DES)

What can still possibly go wrong?

Application
protocol configuration

Infrastructure
certificate management

Protocol Logic
e.g. ambiguous messages

- cause servers to attribute secrets to wrong clients

TLS DESIGN

Cryptography
e.g. no fresh IV

- write applet to realize adaptive attack (BEAST)

Implementation Errors
many critical bugs

Weak Algorithms
MD5, PKCS1, RC4, …

TLS in F# & F7: miTLS

Develop and verify a **reference implementation** of SSL 3.0—TLS 1.2

1. **Standard compliance**: closely follow the RFCs
   - concrete message formats
   - support for multiple ciphersuites, sessions and connections, re-handshakes and resumptions, alerts, message fragmentation,…
   - interop with other implementations such as web browsers and servers

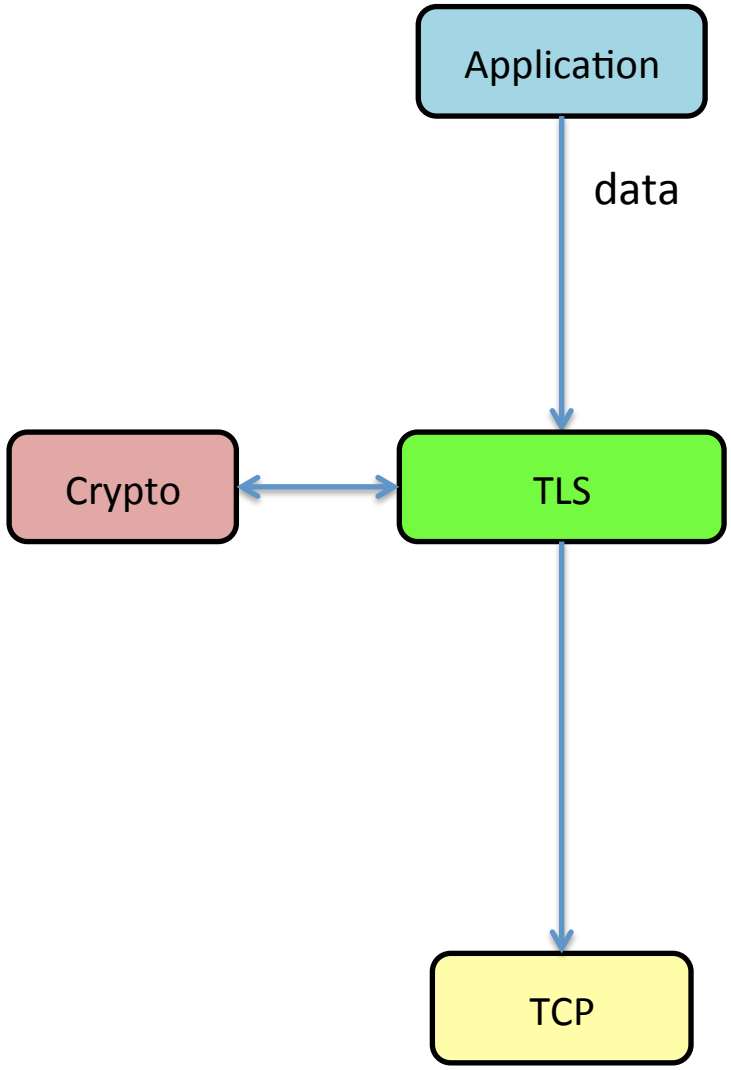
2. **Verified security**: code is structured to enable modular verification, from its main API down to concrete assumptions on its base cryptography (e.g. RSA)
   - formal computational security theorems for a 7000-line functionality (automation required)

3. **Experimental platform:** FlexTLS
   - for testing corner cases, trying out attacks, analysing new extensions and patches, …

TLS Security Goals, Informally

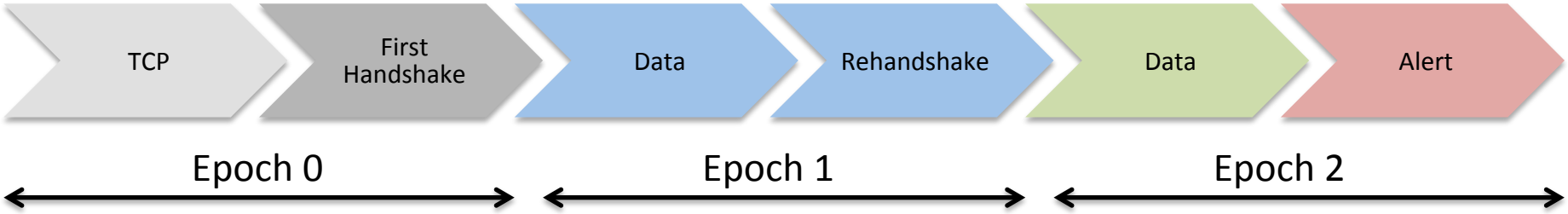
- Goals
  - Plaintext confidentiality
  - Server (and client) authentication
  - Stream integrity

- Given a TLS connection with
  - Honest parties
  - Strong crypto algorithms
  - Recent protocol versions and extensions



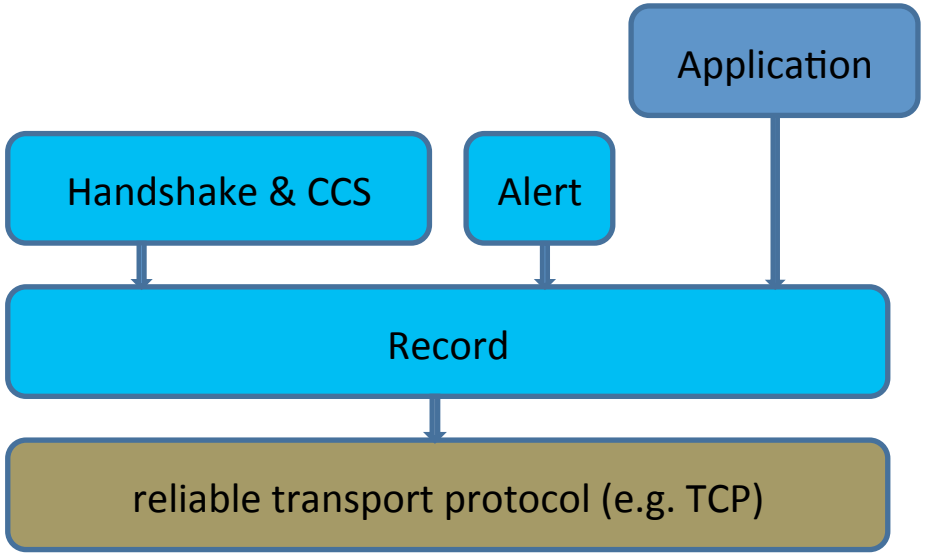
Challenges

- Cryptographic agility
  - Ciphersuites, protocol versions
  - Some are weaker than others
  - Prove security for the negotiated parameters
- Complex state machines
  - Multiple epochs: initial handshake; resumption; renegotiation
  - Fragmentation
  - Specify and prove security invariants



Verification Approach:
Modular Refinement Type-Checking

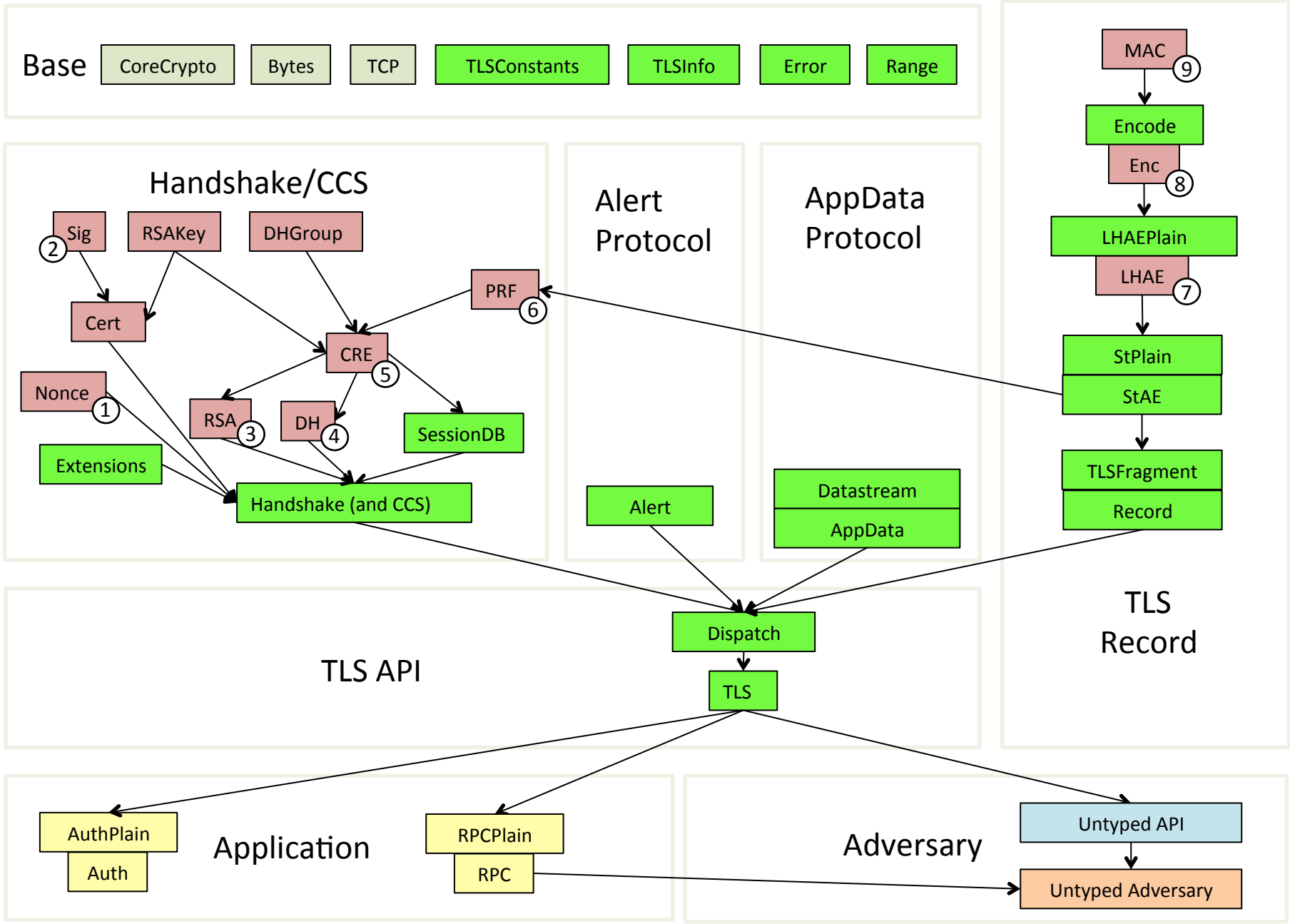
TLS Protocol Structure



Between TCP and Application

- **Record**: private reliable connection
- **Handshake**: ciphersuite negotiation, key exchange
- **Change Cipher Spec**: signaling new keys
- **Alert**: errors and warnings

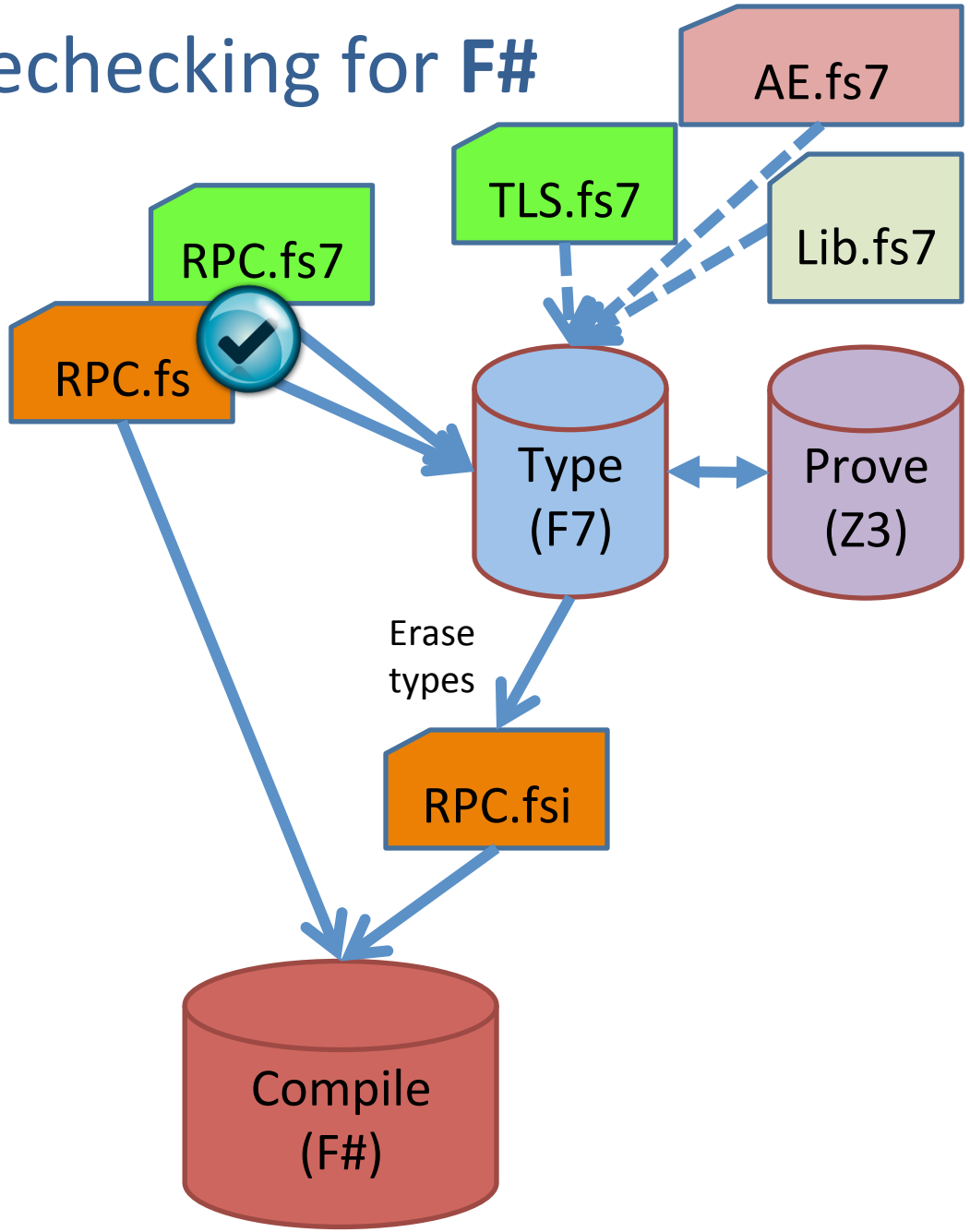
Modular Architecture for miTLS



F7: refinement typechecking for **F#**

[BFG'11]

- We program in F#
- We specify in F7
- We verify modules against interfaces:
  F7 typechecks & calls Z3, an SMT solver, on each logical proof obligation



Refinement Types [BFG'11]

- A value M of type x:T{C(x)} is such that
  - M has type T
  - Formula C(M) holds
- Dependent functions

```
type nat = x:int{x >= 0}

val createBytes = l:nat -> x:bytes{Length(x) = l}
```

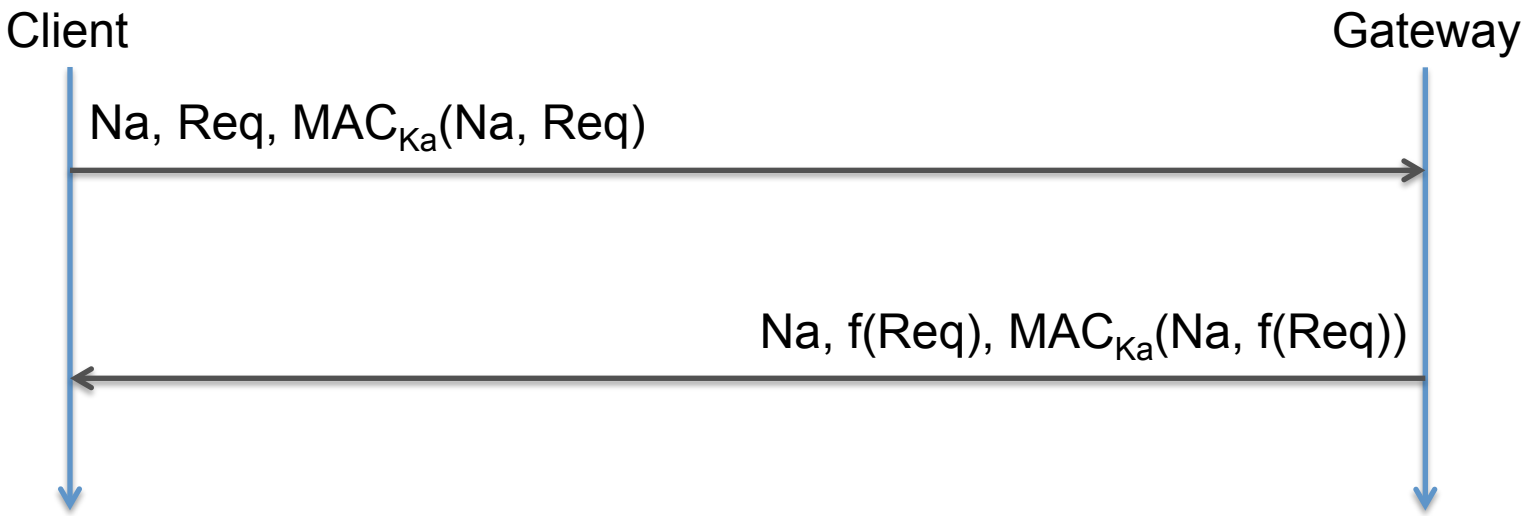
Precondition
l >= 0

Postcondition

- We use these types to
  - Specify cryptographic assumptions
  - Verify protocol code by propagating these assumptions

Type-Based Cryptographic Verification

Example: a Flawed RPC

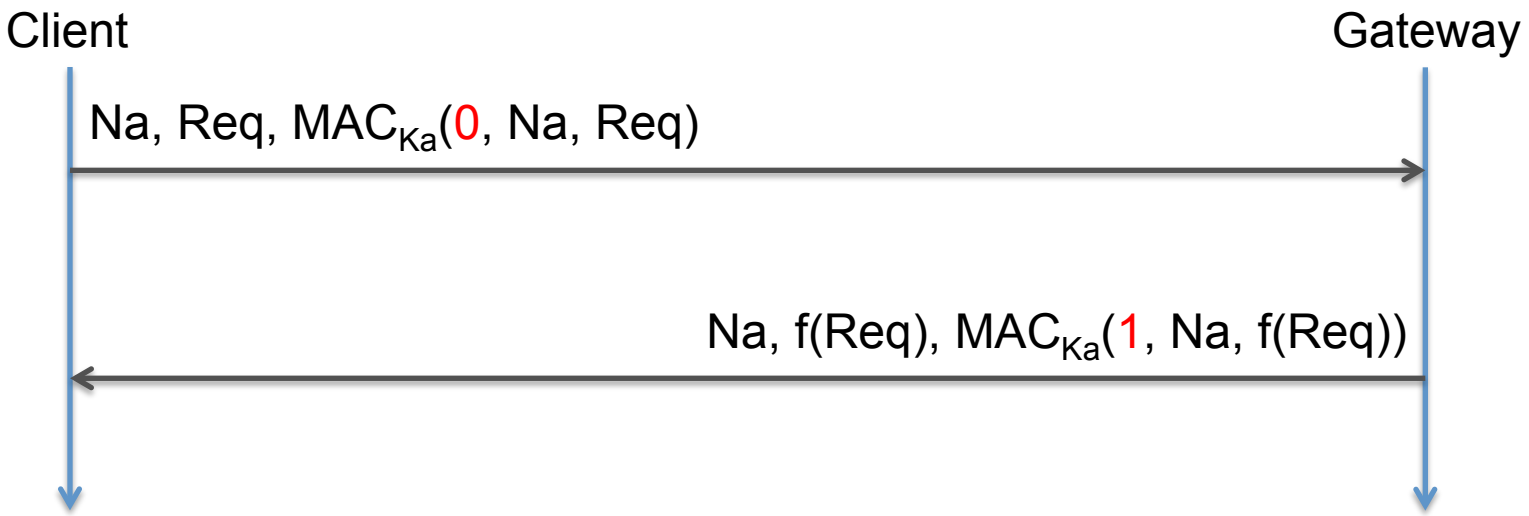



```
module RPC
definition !k,na,msg. Msg(k,na,msg) ⇔
                          Request(na,msg) \/ Response(na,msg)
```

```
let client_send req =
  // precondition:
  // Request(na,req)
 … send MAC k (na,req)
```

```
let client_recv res =
 … if VERIFY k (na,res) mac
then // we have Response(…) or Request(…)
  assert Response(…) //will not typecheck
```

Example: a possible Fix

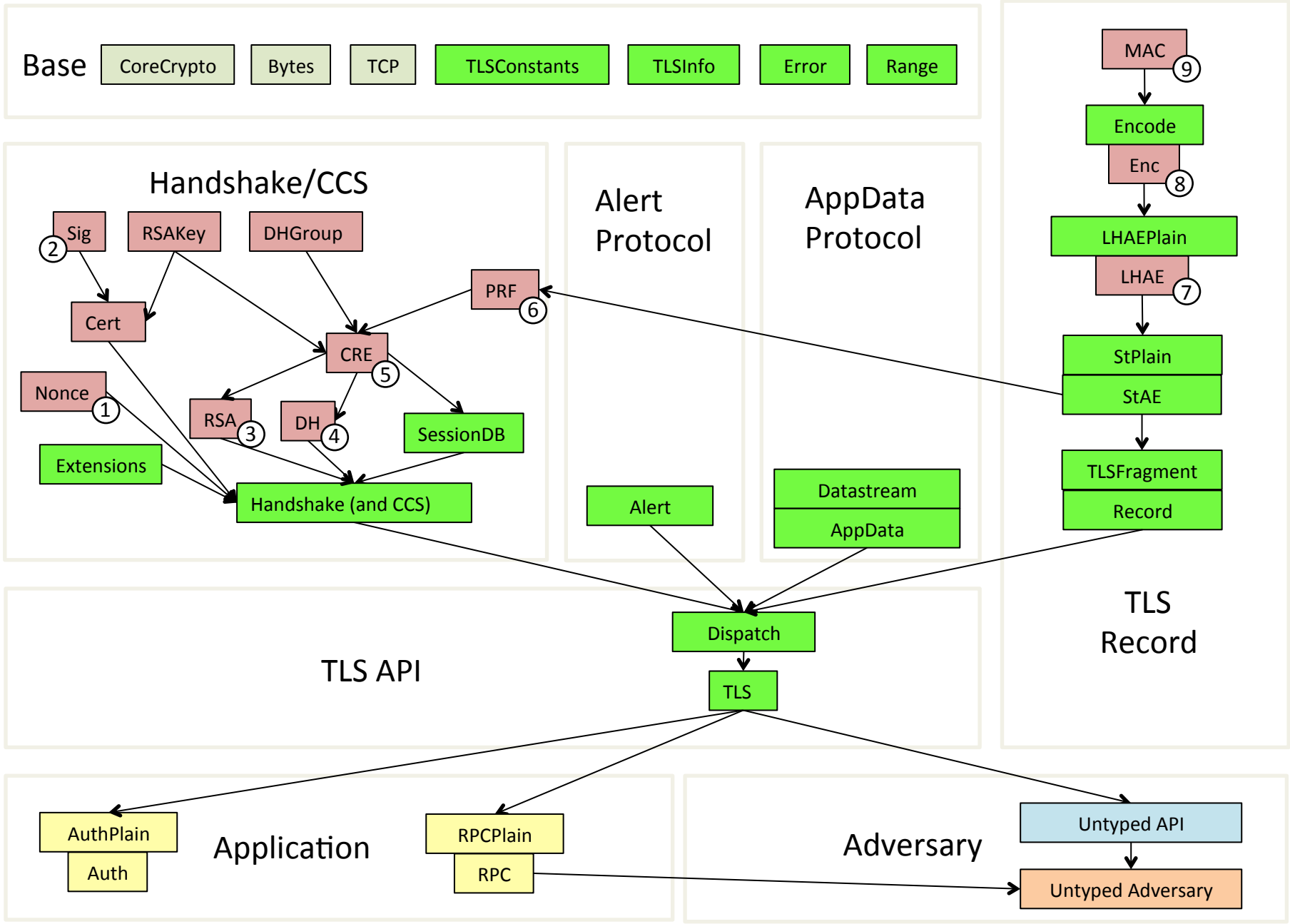



```
module RPC
definition !k,na,msg. Msg(k,tag,na,msg) ⇔
           (tag=0 /\ Request(na,msg)) \/ (tag=1 /\ Response(na,msg))
```

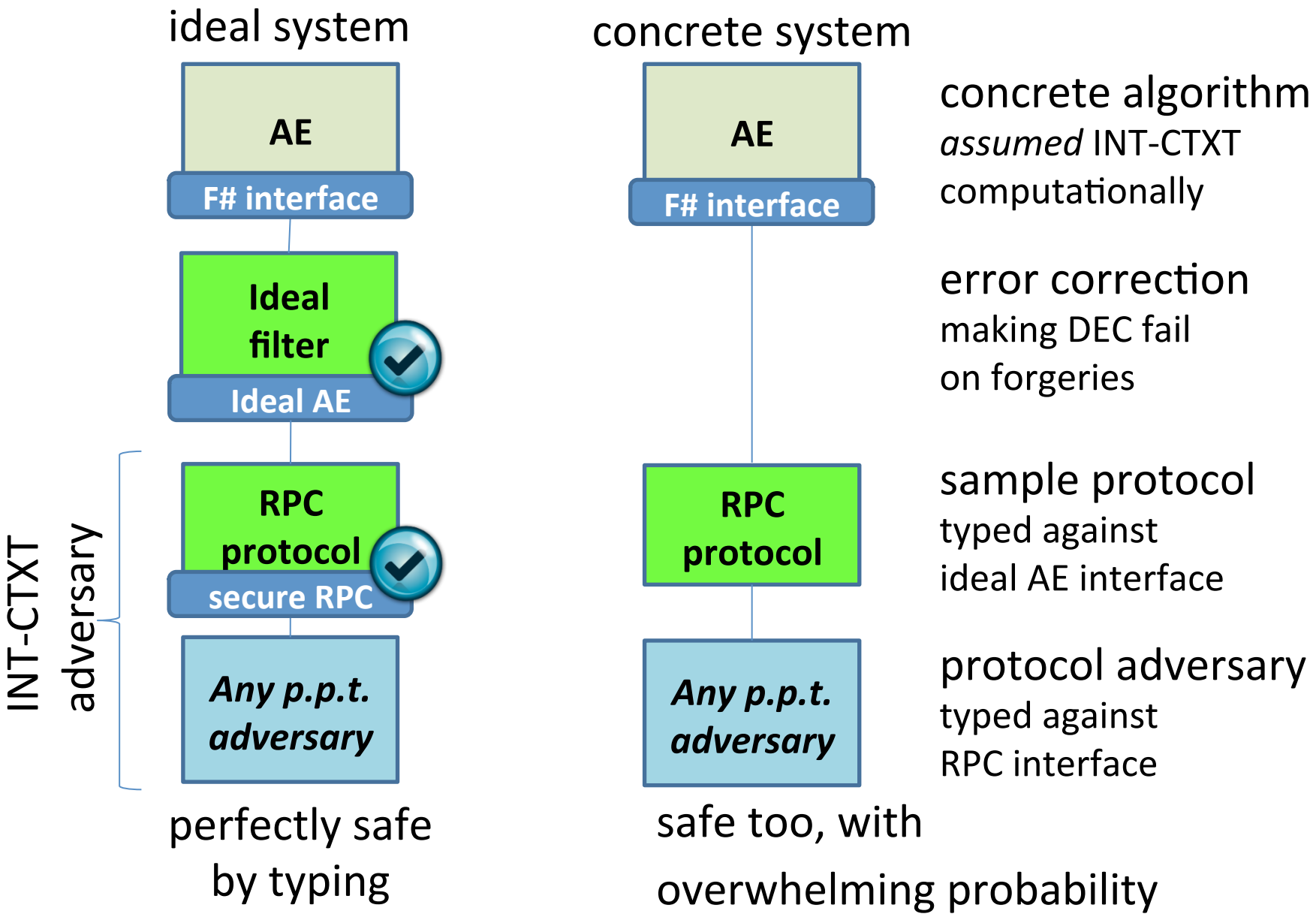
```
let client_send req =
  // precondition:
  // Request(na,req)
 … send MAC k (0,na,req)
```

```
let client_recv res =
 … if VERIFY k (1,na,res) mac
then // we have Response(…)
  assert Response(…) //will typecheck
```

Modular Architecture for miTLS



Computational Safety for AE [FKS'11]



Plaintext Modules

- Encryption is parameterized by a module that abstractly defines plaintexts, with interface

```
module Plaintext

val size: int
type plain
type repr = b:bytes{Length(b)=size}

val coerce : repr  -> plain // turning bytes into secrets
val leak   : plain -> repr  // breaking secrecy!
```

If we remove the **leak** function, we get secrecy by typing

If we remove the **coerce** function, we get integrity by typing

```
val request: unit  -> plain
val respond: plain -> plain // sample protocol code
```

Plain may also implement any protocol function that operates on secrets

Ideal Interface for Authenticated Encryption

```
module AE
open Plaintext

type key
type cipher = b:bytes{Length(b)= size + 16}

val GEN: unit-> key
val ENC: key -> plain  -> cipher
val DEC: key -> cipher -> plain option
```

- Relying on basic cryptographic assumptions (IND-CPA, INT-CTXT) its **ideal implementation** never accesses plaintexts!
  Formally, ideal AE is typed using an abstract **plain** type

ENC k p	encrypts instead zeros to c and logs (k,c,p)
DEC k c	returns Some(p) when (k,c,p) is in the log, or None

Towards TLS: adding **Type Indexes**

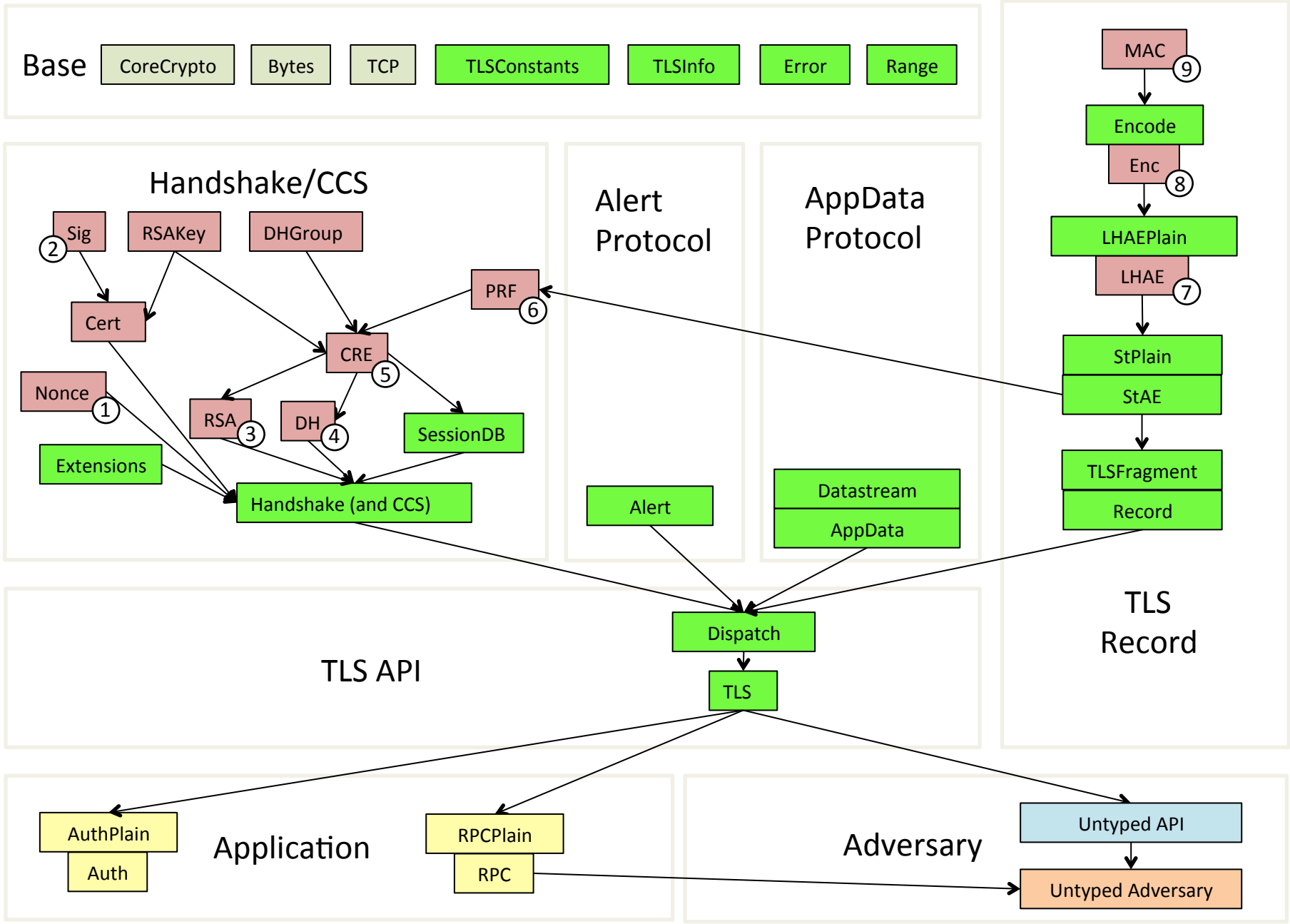
- Within TLS, we keep track of many keys, for different algorithms & sessions
- We use finer ideal functionalities that provide **conditional security** only for "good" keys
  - generated by algorithms assumed **computationally strong**; and
  - for sessions between **honest** participants (not those with the adversary)

```
module AE
open Plain
type (;algorithm, sessionID) key
(…)
val GEN:    a:algorithm -> s:sessionID -> (;a,s) key
val LEAK:   a:algorithm -> s:sessionID
        {Weak(a) or Corrupt(s)} -> (;a,s) key -> bytes
val COERCE: a:algorithm -> s:sessionID
        {Weak(a) or Corrupt(s)} -> bytes -> (;a,s) key
```

The type of the key generated for this algorithm used only for this session

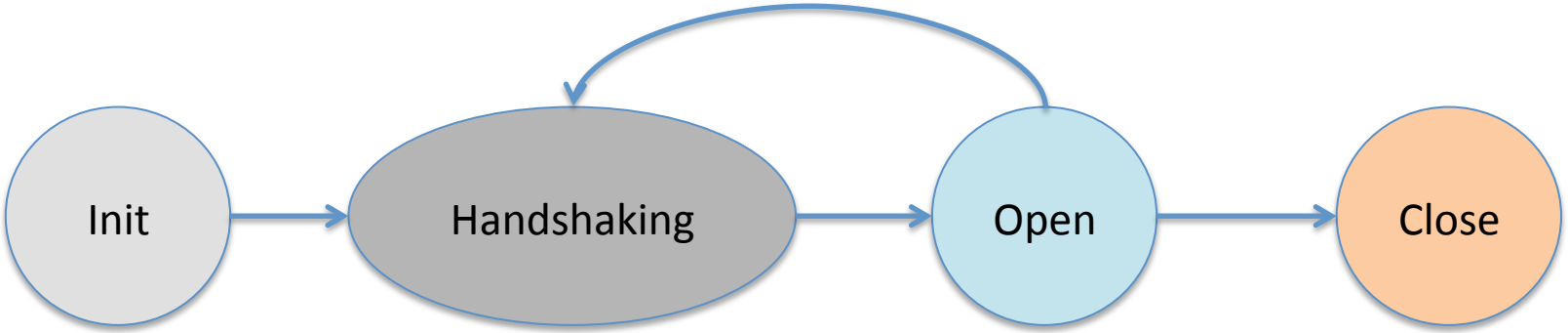
Type-Based
State Machine Verification

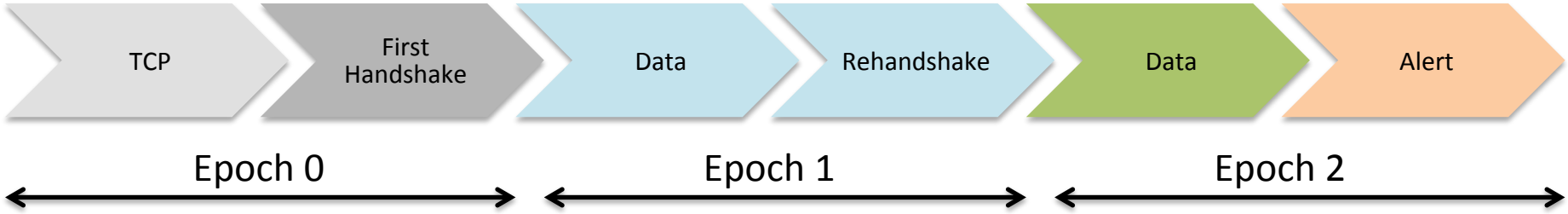
Modular Architecture for miTLS



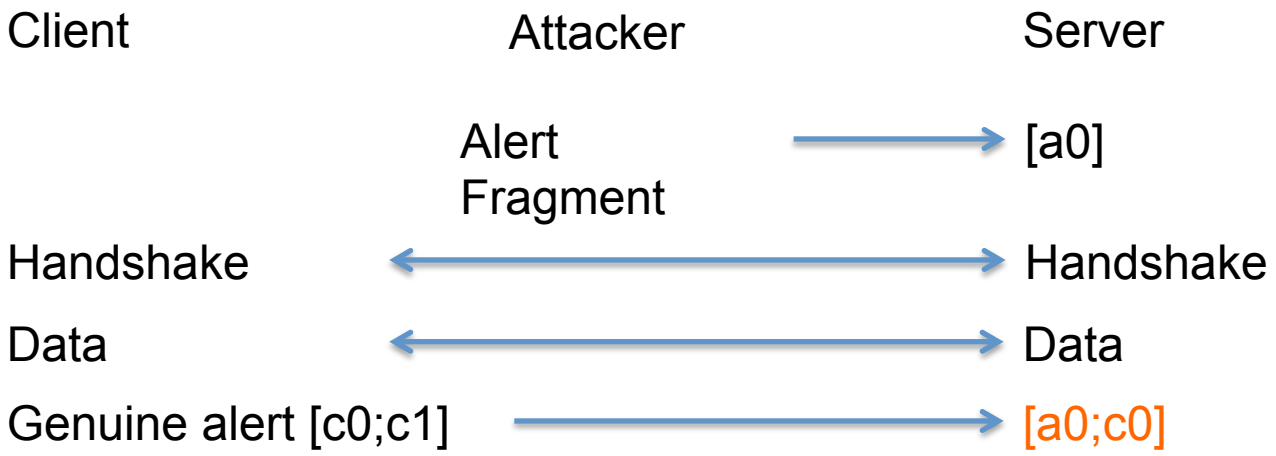
Epochs: Refreshing Keys

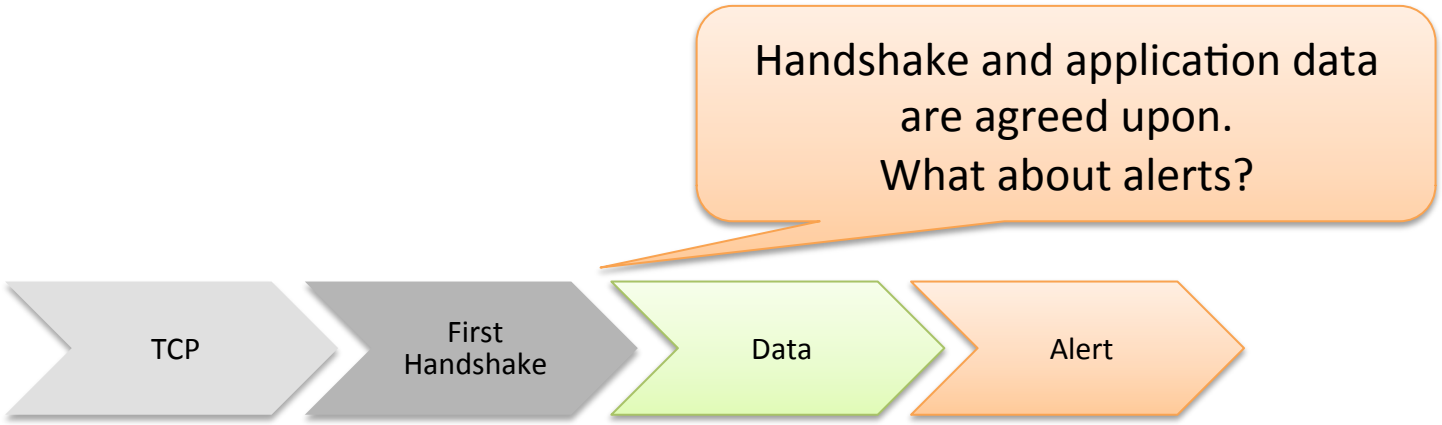
- Example invariants
  - Application data is only sent or accepted in Open state
  - Alerts received in Open state are authentic
- Fragmentation handling
  - Integrity issues (OpenSSL ClientHello bug; Alert attack)





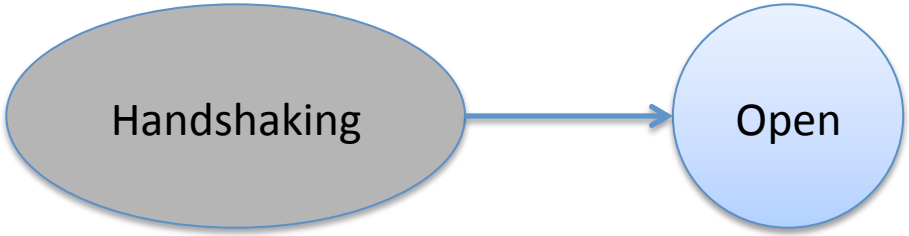
An attack on Alerts





Type checking epoch transitions

```
type (;e:epoch) state = {
        handshake: (;e)Handshake.state
        application: (;e)Application.state
        alert: (;e)Alert.state }
```




```
match recv with
| CCS (e') -> state = {
        handshake = Handshake.moveEpoch e'
        application = Application.moveEpoch e'
        alert = state.alert // Will not typecheck }
```

The TLS API

How applications use TLS

The API Problem

- What applications want: socket replacement
  - connect(), listen(), accept(), read(), write(), close()
- What we can prove:

type (;*id*:*epoch*) *stream*
type (;*id*:*epoch*, *h*:(;*id*)*stream*, *r*:*range*) *data*
val *data*:
  id:*epoch*{*not*(*Auth*(*id*))} → *s*:(;*id*) *stream* → *r*:*range* →
  b:(;*r*) *rbytes* → *c*: (;*id*,*s*,*r*) *data*
val *repr*:
  id:*epoch*{*not*(*Safe*(*id*))} → *s*:(;*id*) *stream* → *r*:*range* →
  c: (;*id*,*s*,*r*) *data* → (;*r*) *rbytes*
val *split*: *id*:*epoch* → *s*:(;*id*) *stream* →
  r0:*range* → *r1*:*range* → *d*:(;*id*,*s*,*Sum*(*r0*,*r1*)) *data* →
  d0:(;*id*,*s*,*r0*) *data* ∗ *d1*:(;*id*,*ExtendStream*(*id*,*s*,*r0*,*d0*),*r1*) *data*

type (;*c*:*CI*) *query*

type *Cn*
type (;*g*:*config*) *Cn0* = *c0*:*Cn*{*InitCn*(*g*,*c0*)}
type (;*c*:*Cn*) *nextCn* = *c'*:*Cn*{*NextCn*(*c*,*c'*)}
type (;*c*:*Cn*) *msg_i* = *r*:*range* ∗ (;*CI*(*c*).*id_in*, *Stream_i*(*c*), *r*) *data*
type (;*c*:*Cn*) *msg_o* = *r*:*range* ∗ (;*CI*(*c*).*id_out*, *Stream_o*(*c*), *r*) *data*

type (;*c*:*Cn*) *ioresult_i* =
| *Read* **of** *c'*:(;*c*) *nextCn* ∗ *d*:(;*c*) *msg_i*
  {*Extend_i*(*c*,*c'*,*d*) ∧ (*Auth*(*CI*(*c*).*id_in*) ⇒ *Write*(*CI*(*c*).*id_in*, *Bytes_i*(*c'*)))}
| *Close* **of** *TCP.Stream*{*Auth*(*CI*(*c*).*id_in*) ⇒ *Close*(*CI*(*c*).*id_in*, *Bytes_i*(*c*))}
| *Fatal* **of** *a*:*alertDescription*
  {*Auth*(*CI*(*c*).*id_in*) ⇒ *Fatal*(*CI*(*c*).*id_in*,*a*,*Bytes_i*(*c*))}
| *CertQuery* **of** *c'*:(;*c*) *nextCn* ∗ (;*c'*) *query* {*Extend*(*c*, *c'*)}
| *Handshaken* **of** *c'*:*Cn* {*Complete*(*CI*(*c'*),*Cfg*(*c'*)) ∧ ...}
| ...
val *read* : *c*:*Cn* → (;*c*) *ioresult_i*

type (;*c*:*Cn*,*d*:(;*c*) *msg_o*) *ioresult_o* =
| *WriteComplete* **of** *c'*:(;*c*) *nextCn* {*Extend_o*(*c*,*c'*,*d*)}
| *WritePartial* **of** *c'*:(;*c*) *nextCn* ∗ *d'*:(;*c'*) *msg_o*
  { ∃*d0*. *Extend_o*(*c*,*c'*,*d0*) ∧ *Split_o*(*c*, *d*, *d0*, *c'*, *d'*) }
| *WriteError* **of** *alertDescription option*
| *MustRead* **of** *c'*:*Cn* {...}
val *write*: *c*:*Cn* → *d*:(;*c*) *msg_o* → (;*c*,*d*) *ioresult_o*

API Example: SSL_read

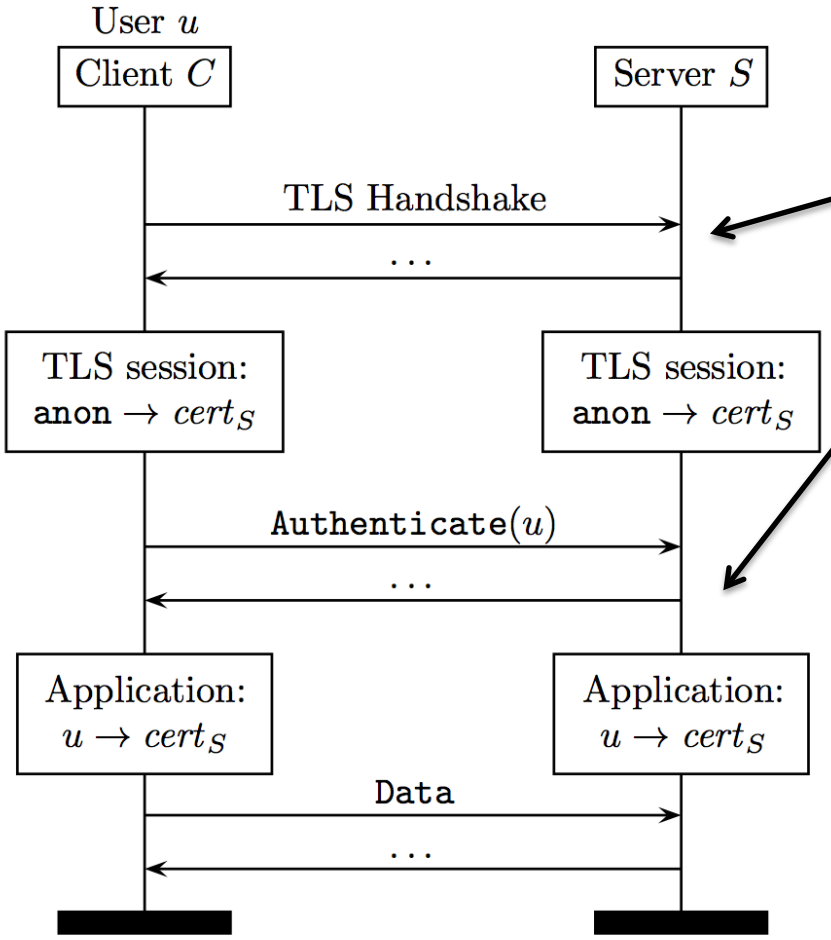
- Return value 0: Read operation was not successful. The reason may either be:
  - **a clean shutdown** due to a *close_notify* alert sent by the peer (in which case the SSL_RECEIVED_SHUTDOWN flag in the SSL shutdown state is set)
  - **or the peer simply shut down the underlying transport**

OpenSSL Manual

- Consequence: Truncation attacks
  - "Send money to Charlie~~'s Angels~~"
  - Logout issues; Cookie integrity

The Triple Handshake attack

User Authentication over TLS



- Common Pattern
  - *Outer*: server-authenticated TLS
  - *Inner*: user authentication protocol
- Many examples
  - SASL, GSSAPI, PEAP, …
  - Renegotiation with client certificate
- Common concerns
  - *How to bind inner authentication with outer channel?*

API Example: Renegotiation

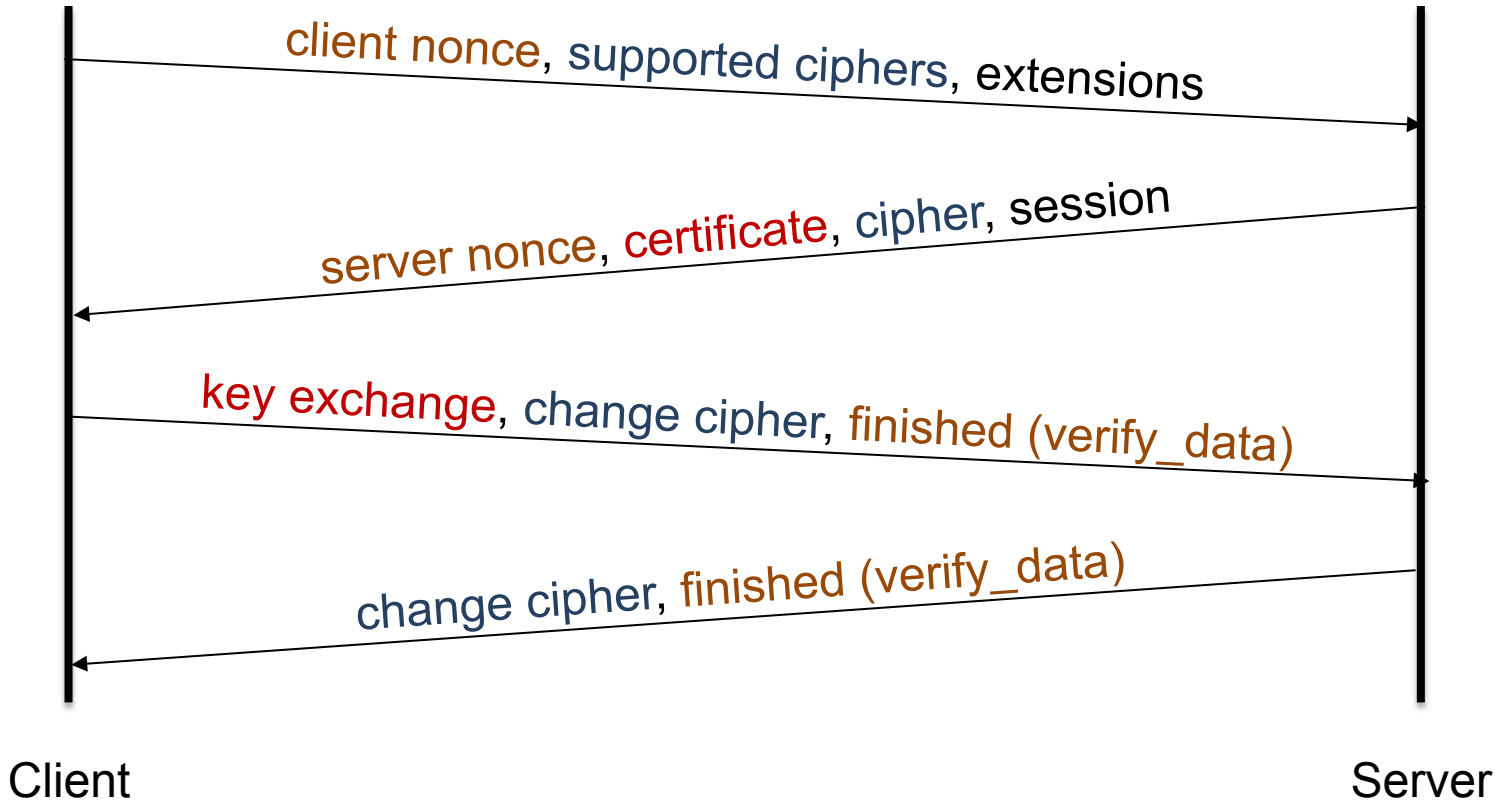
- "If peer requests a renegotiation, **it will be performed transparently** during the SSL_read() operation."
- "As at any time a re-negotiation is possible, a call to SSL_write() **can also cause read operations**!"

OpenSSL Manual

By comparison, in miTLS:

- Application data indexed by epoch
  - No security context confusion
- A write can never read
  - No buffering of application data; have to read often enough

Background: TLS RSA Handshake

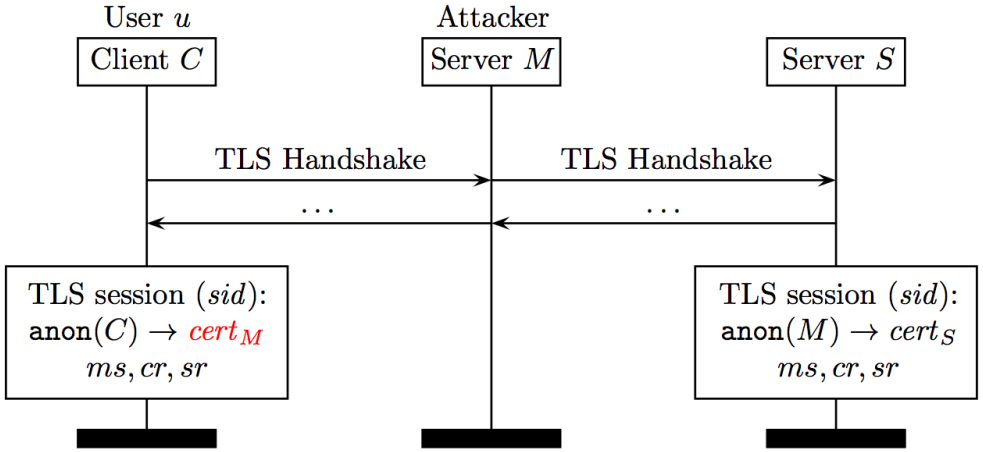


- PMS: randomly chosen by client; RSA-encrypted
- MS = PRF(PMS, Client Nonce, Server Nonce)

Triple Handshake Attack: (Phase 1 of 3)

- In the RSA handshake, *M* can ensure that the master secrets on both connections is the same
  - *M* re-encrypts *C's* premaster secret under *S*'s public key
  - Uses same client and server randoms on both handshakes

- *M* can also do this with DHE handshakes
  - It chooses a "bad" Diffie-Hellman group

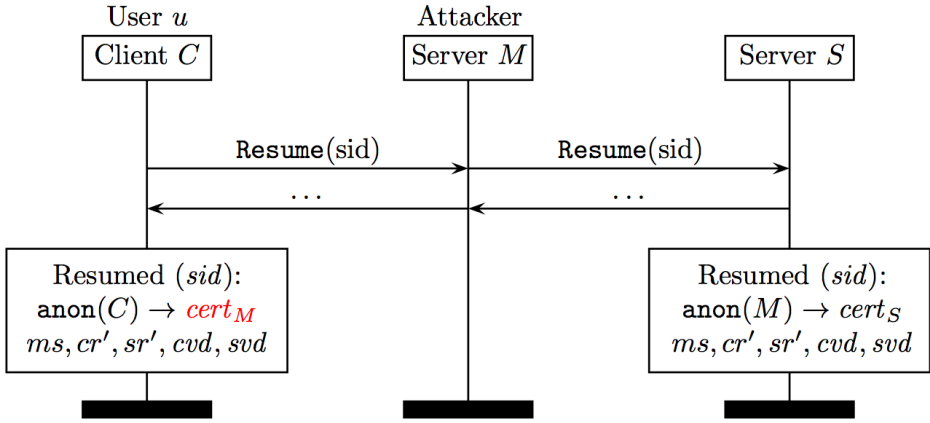


- *Impact*: The master secret is not a good channel identifier
- *Breaks*: Compound authentication (reenables 2002 attack)

Detailed message traces: https://mitls.org/pages/attacks/3SHAKE

Triple Handshake Attack: (Pahse 2 of 3)

- If *C* resumes its session on a new connection, *M* can forward the abbreviated handshake to *S*
  - Works because master secrets, ciphersuites, session ID the same
  - Both new connections will have the ***same handshake log*** and ***same verify_data***
  - On both connections, same tls-unique=server_verify_data

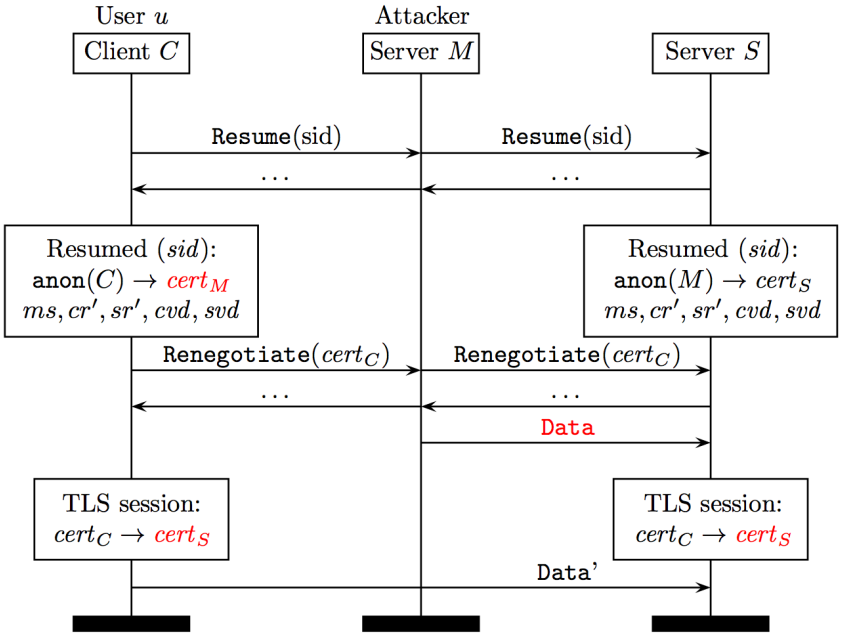


- *Impact*: tls-unique is not a good identifier after resumption
- *Breaks*: Channel Binding in SASL (SCRAM, GS2)

Triple Handshake Attack: (Phase 3 of 3)

- If *C* renegotiates with client certificate;
  M can forward the renegotiation handshake to *S*
  - Works because renegotiation indication is the same

    RI = client_verify_data + server_verify_data



- *Impact*: Renegotiation Indication is not a good channel identifier after session resumption
- *Breaks*: Renegotiation Indication (reenables Rex's attack)

Countermeasures

- For renegotiation, implementation checks will be enough
  - Always validate server certificate during renegotiation
  - Many TLS clients still need to do this
- For tls-unique, compound auth, no easy fixes
  - Don't rely on tls-unique after resumption
  - Ensure that clients only present user credentials to trusted servers

- *Root problem*: Master secrets generated in different TLS security contexts can be the same
  - E.g. master secret does not depend on server certificate

If we make the master secret a good session identifier, tls-unique and renegotiation indication will be fixed for free!

Fix: Extended Master Secret

- Compute a session hash for full handshakes

```
session_hash = Hash(handshake_messages)
```

  - All messages up to and including ClientKeyExchange (since master_secret will be needed in SSL 3.0 CertificateVerify)

- Add session hash to master secret derivation:

```
master_secret = PRF(pre_master_secret,
                    "extended master secret”,
                     session_hash) [0..47];
```

- RFC 7627: TLS Session Hash and Extended Master Secret Extension

Why Triple Handshake Wasn't Discovered Earlier

- Giesen *et al.,* CCS'13
  On the Security of TLS Renegotiation
  - Doesn't model resumption

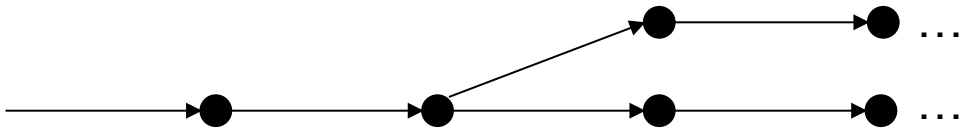
- Krawczyk *et al*., CRYPTO'13. On the Security of the TLS Protocol: A Systematic Analysis
  - Doesn't model resumption or renegotiation

- Bhargavan et al., IEEE S&P'13
  Implementing TLS with Verified Cryptographic Security
  - Models resumption and renegotiation
  - Attack falls outside the scope of the authentication guarantees for resumption

- Bhargavan et al., CRYPTO'14
  Proving the TLS Handshake Secure (as it is)
  - Discussion of the attack; no formal proof of the fix

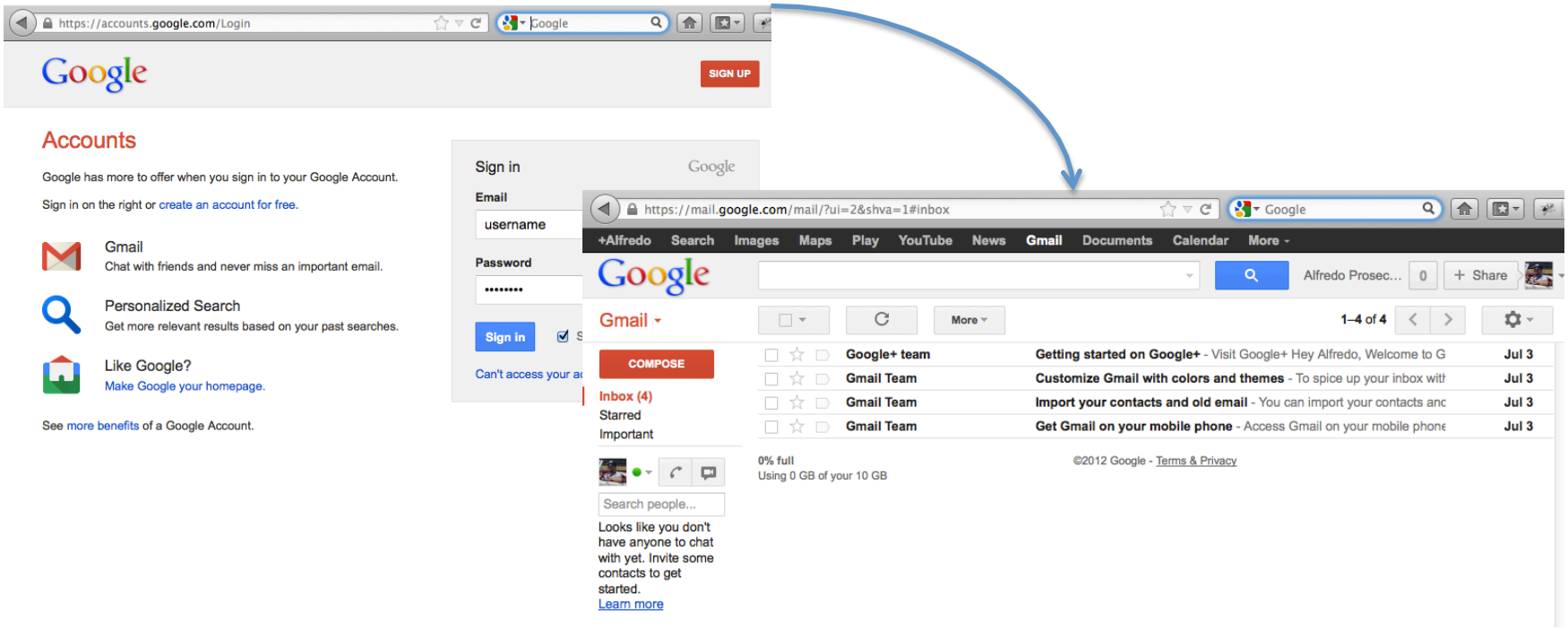
Expected security from tunneled TLS sessions

- In bare TLS, each session is actually independent
- Yet renegotiation is tunneled
  - No forwarding of upcoming attacker data
  - No blessing of previous attacker data (!) [Ray'09]
  - Fixed by renegotiation indication extension
- Resumption over a new TCP connection
  - No binding to the original session context



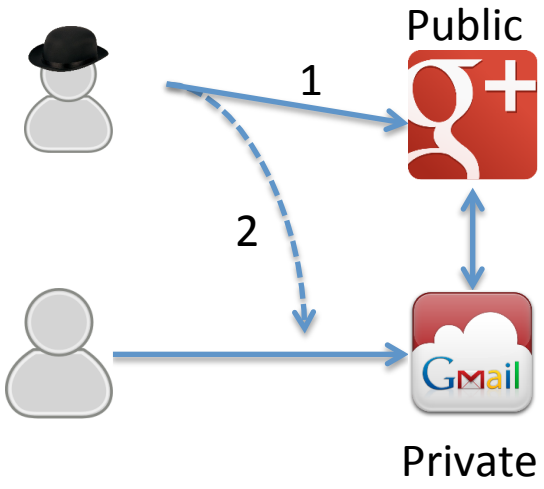
Taking Side Channels into Account

Confidentiality with TLS



- Expected confidentiality
  - Email content
  - Username and password (identity)

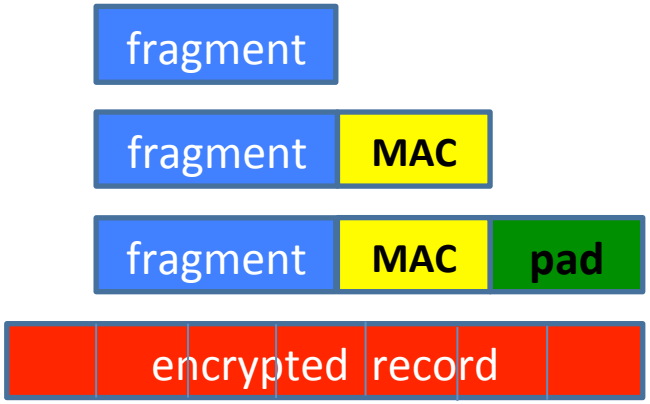
Identifying Web Users



- Each profile picture has a different size
- TLS does not protect message length
- The attacker learns the username

n = 931 Google users	m/n – identifiability	
m	Gmail profile picture	Gmail + XRef
1	292 (31.36%)	789 (84.75%)
2	232 (24.92%)	112 (12.03%)
3	180 (19.33%)	30 (3.22%)
4-6	227 (24.39%)	–

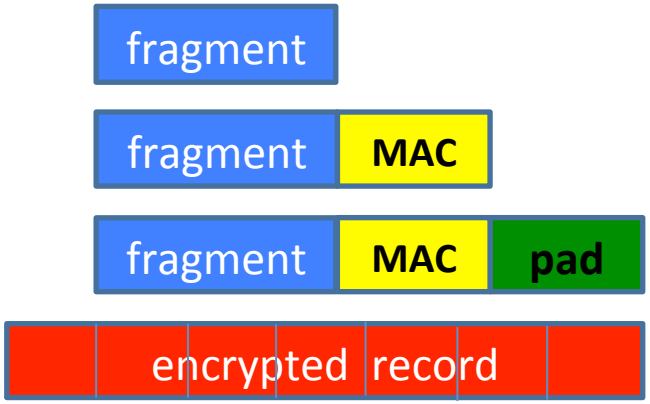
Searching for a countermeasure



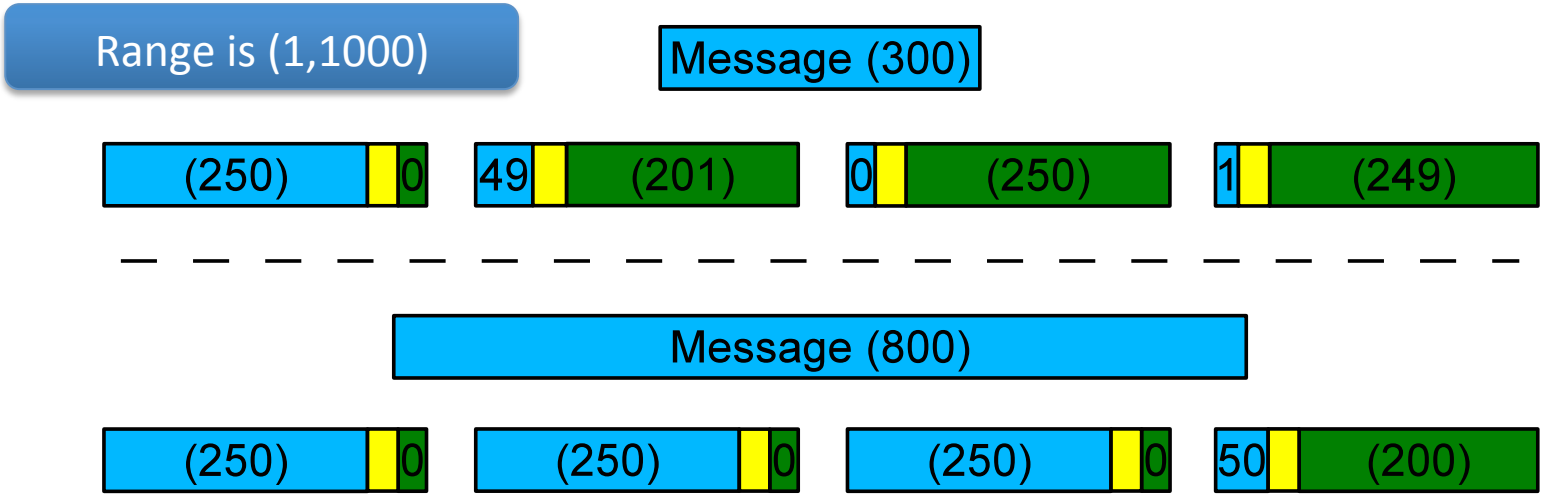
"[Padding] Lengths longer than necessary might be desirable to frustrate attacks on a protocol that are based on analysis of the lengths of exchanged messages"

- Why not random padding?
- No protection against repeated sampling
  - In practice, shortest message leaks length

Searching for a countermeasure



"[Padding] Lengths longer than necessary might be desirable to frustrate attacks on a protocol that are based on analysis of the lengths of exchanged messages"



Length-Hiding: from Crypto to API

LH-AE

```
val ENC: key -> r:range -> p:(;r) plain ->
            c:cipher {InRange(r,c)}
```

Top level API

```
val write: c:cn -> r:range -> p:(;r)plain ->
            (;c,p)ioresult_o
```

Fragmentation and padding

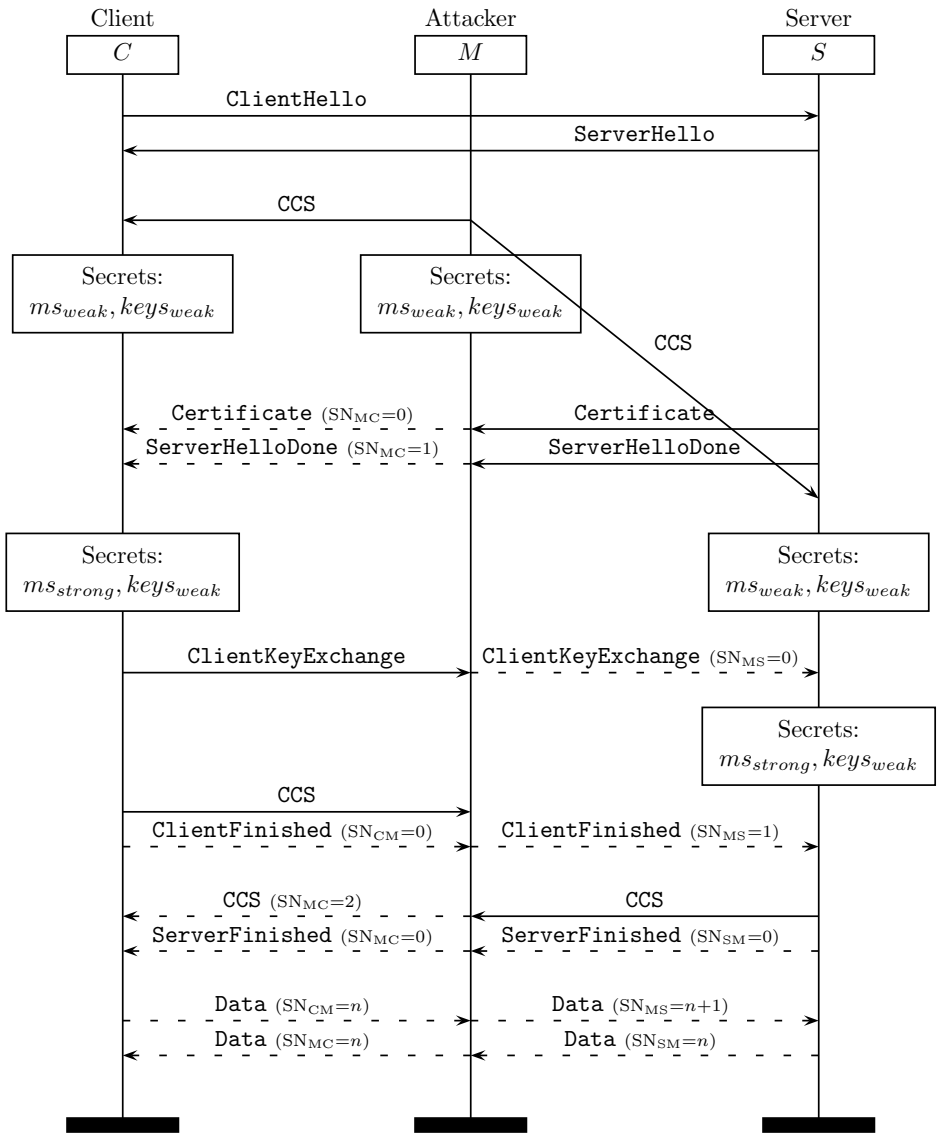
```
val splitRange: r:range ->
            (r0:range * r1:range){r = Sum(r0,r1) /\ Frag(r0)}
val split: r0:range -> r1:range ->
            p:(;Sum(r0,r1)) plain->
            (p0:(;r0) plain * p1:(;r1) plain)
```

FlexTLS
Rapid prototyping of TLS scenarios

Goals and Motivation

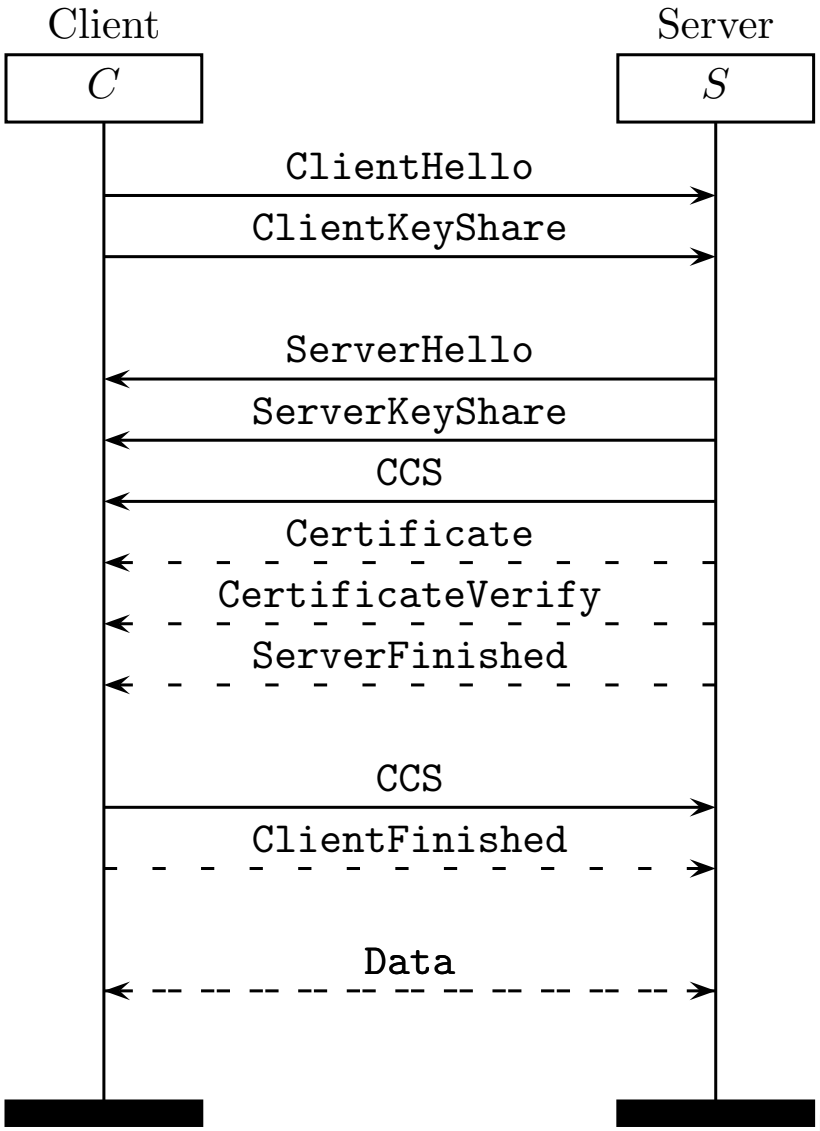
- Rapid prototyping of TLS scenarios
- Arbitrary reordering, fragmentation and tampering of protocol messages
- High-level messaging API, with sensible default values
- Easy management of concurrent sessions and connections
- Quick extensions to miTLS; incremental verification
- Greater tool impact and adoption

Example – Early CCS attack



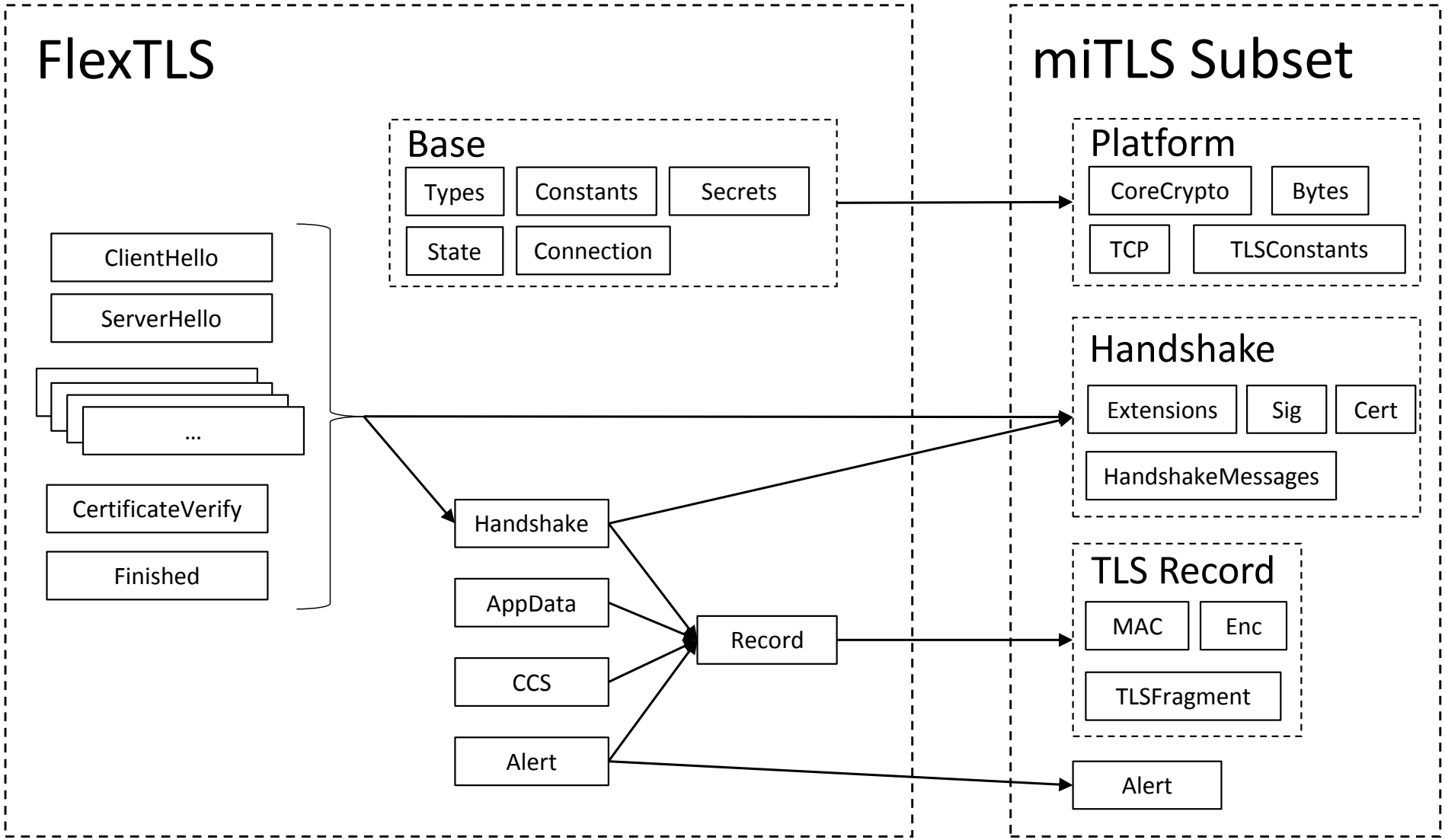

```
1  let earlyCCS (server_name:string, port:int) : state * state =
2
3    (* Start being a Man-In-The-Middle *)
4    let sst,_,cst,_ = FlexConnection.MitmOpenTcpConnections(
5        "0.0.0.0",server_name,listener_port=6666,
6        server_cn=server_name,server_port=port) in
7
8    (* Forward client hello *)
9    let sst,nsc,sch = FlexClientHello.receive(sst) in
10   let cst  = FlexHandshake.send(cst,sch.payload) in
11
12   (* Forward server hello and check the ciphersuite *)
13   let cst,nsc,csh = FlexServerHello.receive(cst,sch,nsc) in
14   if not (isRSACipherSuite (cipherSuite_of_name (getSuite csh))) then
15       failwith "Demo implemented for the RSA key exchange only"
16   else
17   let sst = FlexHandshake.send(sst,csh.payload) in
18
19   (* Inject CCS to both *)
20   let sst,_ = FlexCCS.send(sst) in
21   let cst,_ = FlexCCS.send(cst) in
22
23   (* Compute the weak keys and start encrypting data we send *)
24   let weakKeys = { FlexConstants.nullKeys with
25                    ms = (Bytes.createBytes 48 0)} in
26   let weakNSC  = { nsc with keys = weakKeys} in
27
28   let weakNSCServer = FlexSecrets.fillSecrets(sst,Server,weakNSC) in
29   let sst = FlexState.installWriteKeys sst weakNSCServer in
30
31   let weakNSCClient = FlexSecrets.fillSecrets(cst,Client,weakNSC) in
32   let cst = FlexState.installWriteKeys cst weakNSCClient in
33
34   (* Forward server cert, server hello done, and client key exchange *)
35   let cst,sst,_ = FlexHandshake.forward(cst,sst) in
36   let cst,sst,_ = FlexHandshake.forward(cst,sst) in
37   let sst,cst,_ = FlexHandshake.forward(sst,cst) in
38
39   (* Get the Client CCS, drop it, but install new weak reading keys *)
40   let sst,_,_ = FlexCCS.receive(sst) in
41   let sst   = FlexState.installReadKeys sst weakNSCServer in
42
43   (* Forward the client finished message *)
44   let sst,cst,_ = FlexHandshake.forward(sst,cst) in
45
46   (* Forward the CCS, and install weak reading keys on client side *)
47   let cst,_,_ = FlexCCS.receive(cst) in
48   let cst   = FlexState.installReadKeys cst weakNSCClient in
49   let sst,_ = FlexCCS.send(sst) in
50
51   (* Forward server finished message *)
52   let cst,sst,_ = FlexHandshake.forward(cst,sst) in
53   sst,cst
```

Example – TLS 1.3




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

  (* Enable TLS 1.3 and the mandatory "negotiated DH" extension *)
  let cfg = {defaultConfig with maxVer = TLS_1p3; negotiableDHGroups =
             DHE2432; DHE3072; DHE4096; DHE6144; 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

  (* Get the CCS and switch to the next security context *)
  let st,_,_       = FlexCCS.receive(st) in
  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

  (* Send CCS and switch to the next security context *)
  let st,_       = FlexCCS.send(st) in
  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
```

FlexTLS architecture



FlexTLS ongoing and future work

- Systematic testing of existing implementation state machines
  - Automatic generation of deviating scenarios
- Early contribution to the IETF TLS WG
  - TLS 1.3 prototype uncovered parsing issues and potential security problems
  - In contrast with typical research that focuses efforts on established standards
- Further ideas
  - Unit testing
  - Implementation fingerprinting
  - Automatic attack reconstruction from ProVerif counterexamples
  - Incremental verification of new protocol features and versions

Conclusion

What's next?

- Support more ciphersuites (ECDHE) and versions (TLS 1.3)
- Verify linear usage of states
- Relax crypto assumptions
- Reduce trusted code base
- Account for side channels (e.g. timing)

Discussion

- miTLS: a verified reference implementation of TLS
  - http://www.mitls.org
  - 7K lines of code; 3K lines of typed interfaces
  - Verified under precise cryptographic assumptions

- Code designed and developed for verification
  - How to support existing code, “as programmer would write it”?

- Adoption: drop-in .NET stream replacement
  - Still reports only from enthusiasts
  - Lack of engineering support in artifact and verification tools
  - Verification “green light” slows down new feature development