Modular Cryptographic Verification by Typing in F*

Verifying communications protocols: from Needham–Schroeder to TLS
Specifications, models, and implementations
Security, refinements, and type safety

1. Sample security programming: access control
2. Sample protocol: authenticated RPC
3. Computational safety
   for authentication primitives: MACs, signatures
4. Computational secrecy
   for various encryptions: CPA, CCA2, EtM, Hybrid
5. Application:
   authenticated encryption for the TLS transport layer

Review:

Type-Based Verification in F*

Event-Based Specifications: **Assume** and **Assert**

- Suppose there is a global set of events, the **log**
- To evaluate **assume** *C*, add *C* to the log, and return ().
- To evaluate **assert** *C*, return ().
  - If *C* logically follows from the logged formulas,
    we say the assertion **succeeds**; otherwise, the assertion **fails**.
  - The log is only for specification purposes; it does not affect execution
  - Refinement types carry logical properties,
    from assumptions to assertions
  - Type safety guarantees that **all assertions will succeed**.

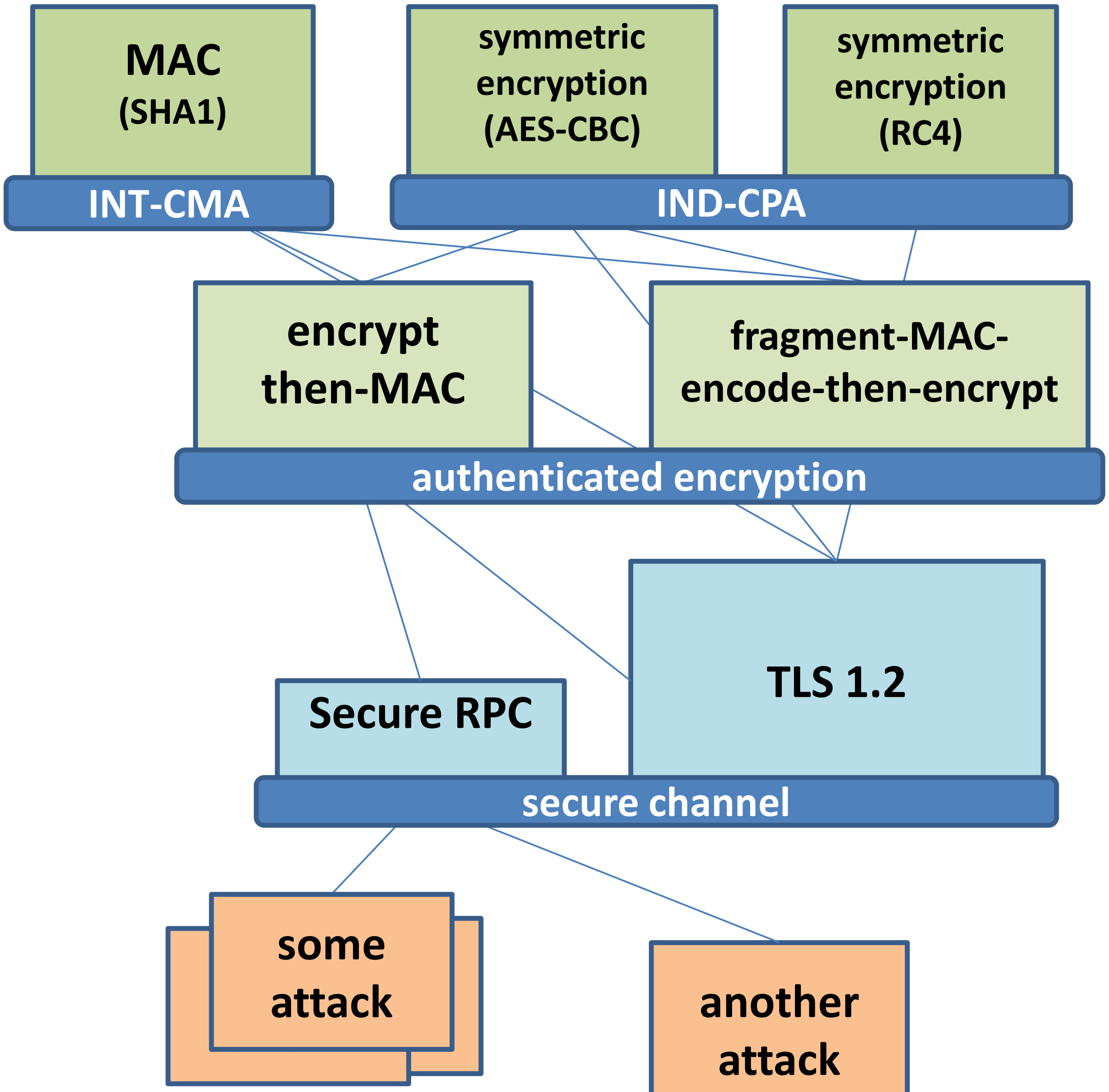
Programming example: access control for files

- Trusted code expresses security policy with assumes and asserts (privileged operations)
- **Untrusted** but **well-typed** code may call trusted libraries
- Typechecking ensures **static compliance** with the policy

Method:

Type-Based Cryptographic Verification

Modular Type-Based Cryptographic Verification



cryptographic algorithms

typed interfaces: **cryptographic assumptions**

cryptographic constructions

typed interfaces: **security guarantees**

security protocols

typed interfaces: **attacker models**

active adversaries

Cryptographic primitives are partially specified

- Symbolic models reason about fully-specified crypto primitives
  - Same rewrite rules apply for the attacker as for the protocol
  - Each crypto primitive yields distinct symbolic terms
- Computational models reason about *partially-specified primitives* (the less specific, the better)
  - *Positive assumptions*: what the protocol needs to run as intended
    e.g. successful decryption when using matching keys
  - *Negative assumptions*: what the adversary cannot do
    e.g. cannot distinguish between encryptions of two different plaintexts
- Security proofs apply parametrically,
  for any concrete primitives that meet these assumptions
- **Typed interfaces** naturally capture partial specifications

Probabilistic F* ?

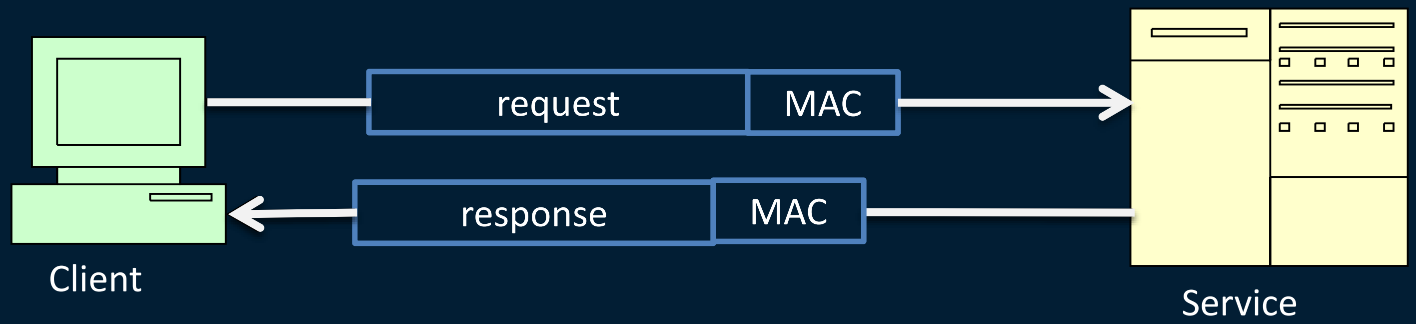
- We equip F* with a probabilistic semantics (Markov chains)

$$A \rightarrow_p A'$$

  - We add a new "fair coin-tossing" primitive
  - The rest of the semantics is unchanged (reductions, structural rules, type safety)

Sample Communications Protocol in F*

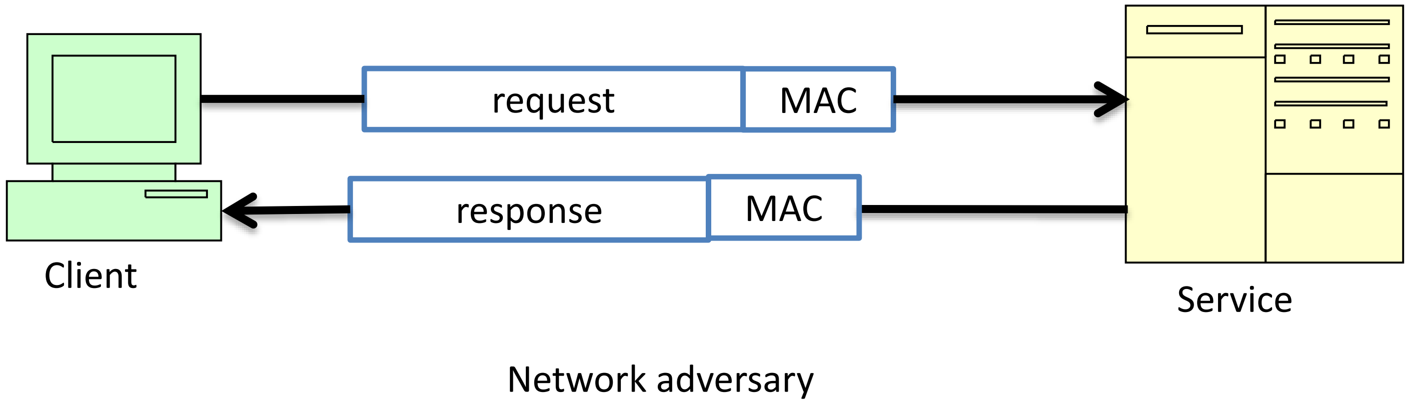
Authenticated RPC



Network adversary

Authenticated RPC

1. $a \rightarrow b : \; utf8\; s \mid (hmacsha1\; k_{ab}\; (request\; s))$
2. $b \rightarrow a : \; utf8\; t \mid (hmacsha1\; k_{ab}\; (response\; s\; t))$



Informal description

1. $a \rightarrow b:\ \mathit{utf8}\ s \mid (\mathit{hmacsha1}\ k_{ab}\ (\mathit{request}\ s))$
2. $b \rightarrow a:\ \mathit{utf8}\ t \mid (\mathit{hmacsha1}\ k_{ab}\ (\mathit{response}\ s\ t))$

We design and implement authenticated RPCs over a TCP connection.
We have two roles, client and server, and a population of principals, $a\ b\ c\ \ldots$

Our security goals:

- if b accepts a request s from a,
  then a has indeed sent this request to b;
- if a accepts a response t from b,
  then b has indeed sent t in response to a's request.

We use message authentication codes (MACs) computed as keyed hashes,
such that each symmetric key k_{ab} is associated with
(and known to) the pair of principals a and b.

There are multiple concurrent RPCs between any number of principals.
The adversary controls the network. Keys and principals may get compromised.

Is this protocol secure?

1. $a \to b$: $\mathit{utf8}\ s \mid (\mathit{hmacsha1}\ k_{ab}\ (\mathit{request}\ s))$
2. $b \to a$: $\mathit{utf8}\ t \mid (\mathit{hmacsha1}\ k_{ab}\ (\mathit{response}\ s\ t))$

Security depends on the following:

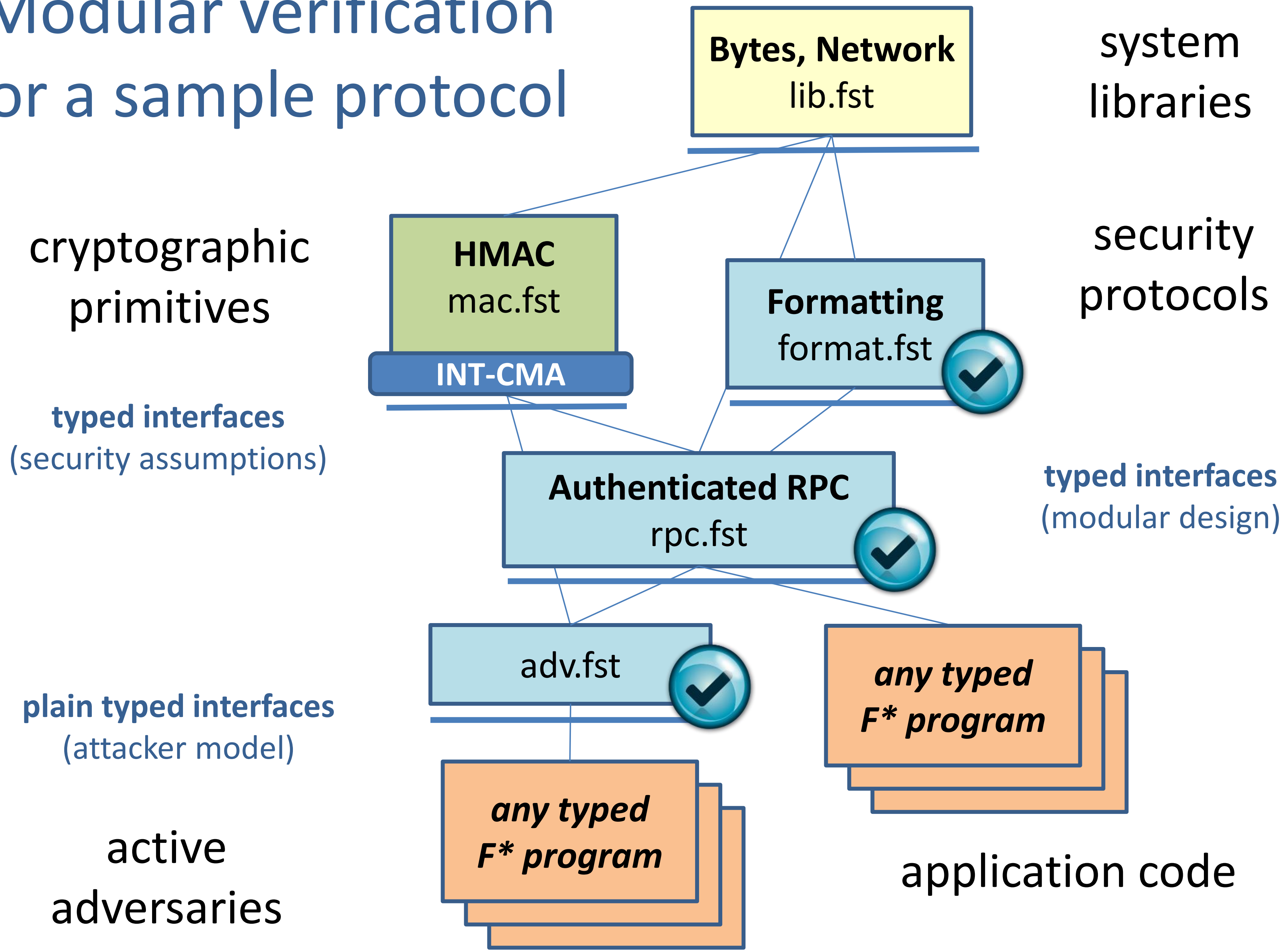
(1) The function *hmacsha1* is cryptographically secure,
so that MACs cannot be forged without knowing their key.

(2) The principals a and b are not compromised,
otherwise the adversary may just use k_{ab} to form MACs.

(3) The functions *request* and *response* are injective and their ranges are disjoint;
otherwise the adversary may use intercepted MACs for other messages.

(4) The key k_{ab} is a key shared between a and b,
used only for MACing requests from a to b and responses from b to a;
otherwise, if b also uses k_{ab} for authenticating requests from b to a,
it would accept its own reflected messages as valid requests from a.

Modular verification for a sample protocol



Test

1. $a \rightarrow b :\ utf8\ s \mid (hmacsha1\ k_{ab}\ (request\ s))$
2. $b \rightarrow a :\ utf8\ t \mid (hmacsha1\ k_{ab}\ (response\ s\ t))$

The messages exchanged over TCP are:

```
Connecting to localhost:8080
Sending {BgAyICsgMj9mhJa7iDAcW3Rrk...} (28 bytes)
Listening at ::1:8080
Received Request 2 + 2?
Sending {AQA0NccjcuL/WOaYSOGGtOtPm...} (23 bytes)
Received Response 4
```

Sample Typed Interface for Cryptography

MAC : integrity

Sample functionality:

Message Authentication Codes

```
module MAC

type text = bytes      val macsize
type key  = bytes
type mac  = bytes


val GEN   : unit -> key
val MAC   : key -> text -> mac
val VERIFY: key -> text -> mac -> bool
```

basic ML interface

This interface says nothing on the security of MACs.

Sample functionality:

Message Authentication Codes

MAC keys are abstract

```
module MAC

type text = bytes      val macsize
type key
type mac  = bytes


val GEN    : unit -> key
val MAC    : key -> text -> mac
val VERIFY: key -> text -> mac -> bool
```

Sample functionality:

Message Authentication Codes

MAC keys are abstract

MACs are fixed sized

```
module MAC
type text = bytes      val macsize
type key
type mac  = b:bytes{Length(b)=macsize}


val GEN   : unit -> key
val MAC   : key -> text -> mac
val VERIFY: key -> text -> mac -> bool
```

Sample functionality:

Message Authentication Codes

MAC keys are abstract

```
module MAC

type text = bytes        val macsize
type key
type mac  = b:bytes{Length(b)=macsize}

logic type Msg: key -> text -> Type

val GEN   : unit -> key
val MAC   : k:key -> t:text{Msg k t} -> mac
val VERIFY: k:key -> t:text -> mac
             -> b:bool{ b=true ⇒ Msg k t }
```

MACs are fixed sized

ideal F7 interface

Msg is specified by protocols using MACs

"All verified messages have been MACed"

Sample functionality:

Message Authentication Codes

MAC keys are abstract

```
module MAC
type text = bytes      val macsize
type key
type mac  = b:bytes{Length(b)=macsize}
logic type Msg: key -> text -> Type
val GEN    : unit -> key
val MAC    : k:key -> t:text{Msg k t} -> mac
val VERIFY: k:key -> t:text -> mac
            -> b:bool{ b=true ⇒ Msg k t }
```

MACs are fixed sized

Msg is specified by protocols using MACs

"All verified messages have been MACed"

ideal F7 interface

```
module RPC
assume ∀ k,q. Msg(k,Utf8(q)) <=> Request(q)
let client q =                | let server q =
  // precondition:            |   … if VERIFY k (utf8 q) m
  // Request(q)               |     then // we have Request(q)
 … send MAC k (utf8 q)        |       process q
```

sample protocol using MACs

Sample functionality:

Message Authentication Codes

MAC keys are abstract

ideal F7 interface

```
module MAC

type text = bytes        val macsize
type key
type mac  = b:bytes{Length(b)=macsize}

logic type Msg: key -> text -> Type

val GEN    : unit -> key
val MAC    : k:key -> t:text{Msg k t} -> mac
val VERIFY: k:key -> t:text -> mac
             -> b:bool{ b=true ⇒ Msg k t }
```

MACs are fixed sized

Msg is specified by protocols using MACs

"All verified messages have been MACed"

This can't be true! (collisions)

concrete F# implementation (using .NET)

```
module MAC
open System.Security.Cryptography

let macsize = 20
let GEN()    = randomBytes 16
let MAC k t = (new HASHMACSHA1(k)).ComputeHash t
let VERIFY k t m = (MAC k t = m)
```

Sample computational assumption:

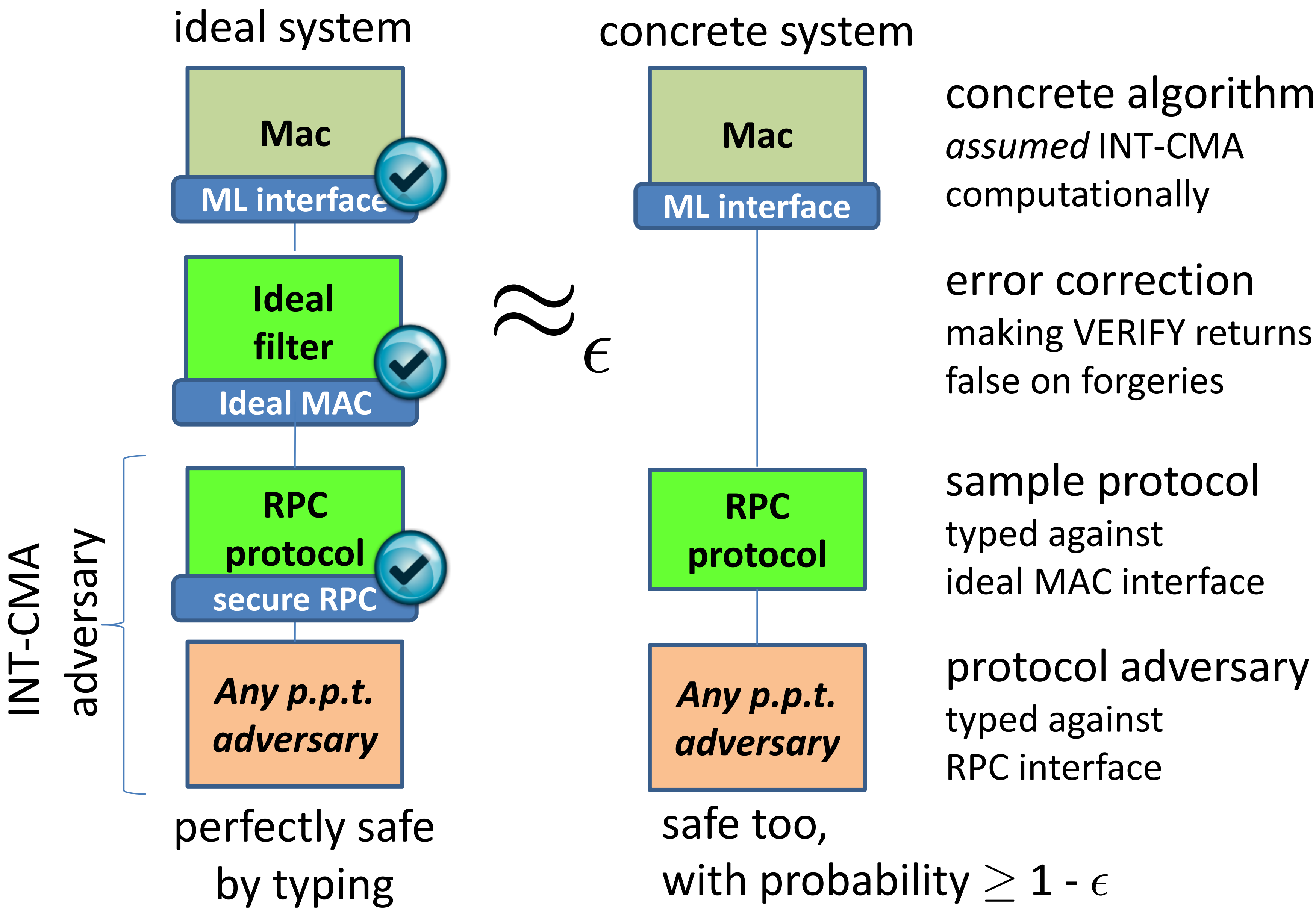
Resistance to Chosen-Message Existential Forgery Attacks (INT-CMA)

```
let k = MAC.keygen()
let log = ST.alloc []

let mac t =
  log := t::!log
  MAC.mac k t

let forgery t m =
  MAC.verify k t m
  && not (List.mem t !log)
```

Computational Safety
a probabilistic polytime program
calling **mac** returns (t,m)
such that **forgery t m** only
with negligible probability ϵ

CMA game [Goldwasser et al. 1988]
programmed in ML

Computational Safety for MACs



Ideal MAC library in F*

- Libraries are multi-instance,
  as opposed to the basic functionality

- Libraries must support key compromise
  for some of their instances

Sample Typed Interface for Cryptography

encryption : secrecy

Perfect Secrecy by Typing

- Secrecy is expressed using observational equivalences between systems that differ on their secrets
- We prove (probabilistic, information theoretic) secrecy by typing, relying on type abstraction

$I_\alpha = \alpha, \ldots, x : T_\alpha, \ldots.$
P_α range over pure modules such that $\vdash P_\alpha \leadsto I_\alpha$.

THEOREM (Secrecy by Typing).
Let A such that $I_\alpha \vdash A : bool$.
For all P_α^0 and P_α^1, we have $P_\alpha^0 \cdot A \approx P_\alpha^1 \cdot A$.

Abstract Plaintexts

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

```
module Plaintext
val size: int
assume type plain
type repr = b:bytes{length b = size}
val coerce : repr  -> plain // turning bytes into secrets
val leak   : plain -> repr  // breaking secrecy!


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

The size of plaintext is fixed (as we cannot hide it)

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

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

Plain may also implement any protocol functions that operates on secrets

Ideal Interface for Authenticated Encryption

```
module AE
open Plaintext

type key
type cipher = b:bytes{length b = size + 16}

val keygen: unit-> key
val encrypt: key -> plain  -> cipher
val decrypt: key -> cipher -> option plain
```

- 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

`encrypt k p`	encrypts instead zeros to c & and logs `(k,c,p)`
`decrypt k c`	returns `Some(p)` when `(k,c,p)` is in the log, or `None` otherwise

An Ideal Interface for CCA2-Secure Encryption

```
module PKE
open Plain

val pksize: int
type skey
type pkey

val ciphersize: int
type cipher = b:bytes{Length b=ciphersize}

val keygen: unit -> pkey * skey
val encrypt: pkey -> plain -> cipher
val decrypt: skey -> cipher -> plain
```

- Its **ideal implementation** encrypts zeros instead of plaintexts so it never accesses plaintext representations, and can be typed parametrically

Sample computational assumption:

Indistinguishability against Chosen Plaintexts & Ciphertexts Attacks

```
module CCA2
open RSA_OAEP
let pk,sk = keygen()
let log = ref []
let b = sample bool
let encryptOracle p0 p1 =
  let p = if b then p0 else p1
  let e = encrypt pk p
  log := e::!log
  e
let decryptOracle c =
  if c in !log
  then None
  else Some(decrypt sk c)
```

CCA2 game
(coded in ML)

Asymptotic security
a probabilistic polytime program calling **encrypt** and **decrypt** guesses which plaintexts are encrypted only with a negligible advantage

Variants: CPA & Authentication

- With **CPA-secure encryption**, we have a **weaker** ideal interface that demands ciphertext integrity before decryption

```
assume type Encrypted of key * cipher

val ENC: k:key -> plain -> c:cipher{Encrypted k c}
val DEC: k:key -> c:cipher{Encrypted k c} -> plain
```

- With **authenticated encryption**, we have a **stronger** ideal interface that ensure plaintext integrity (much as MACs)

```
assume type Msg of key * plain // defined by protocol

val ENC: k:key -> p:plain{Msg k p} -> cipher
val DEC: k:key -> cipher -> p:plain{Msg k p} option
```

Sample Cryptographic Constructions

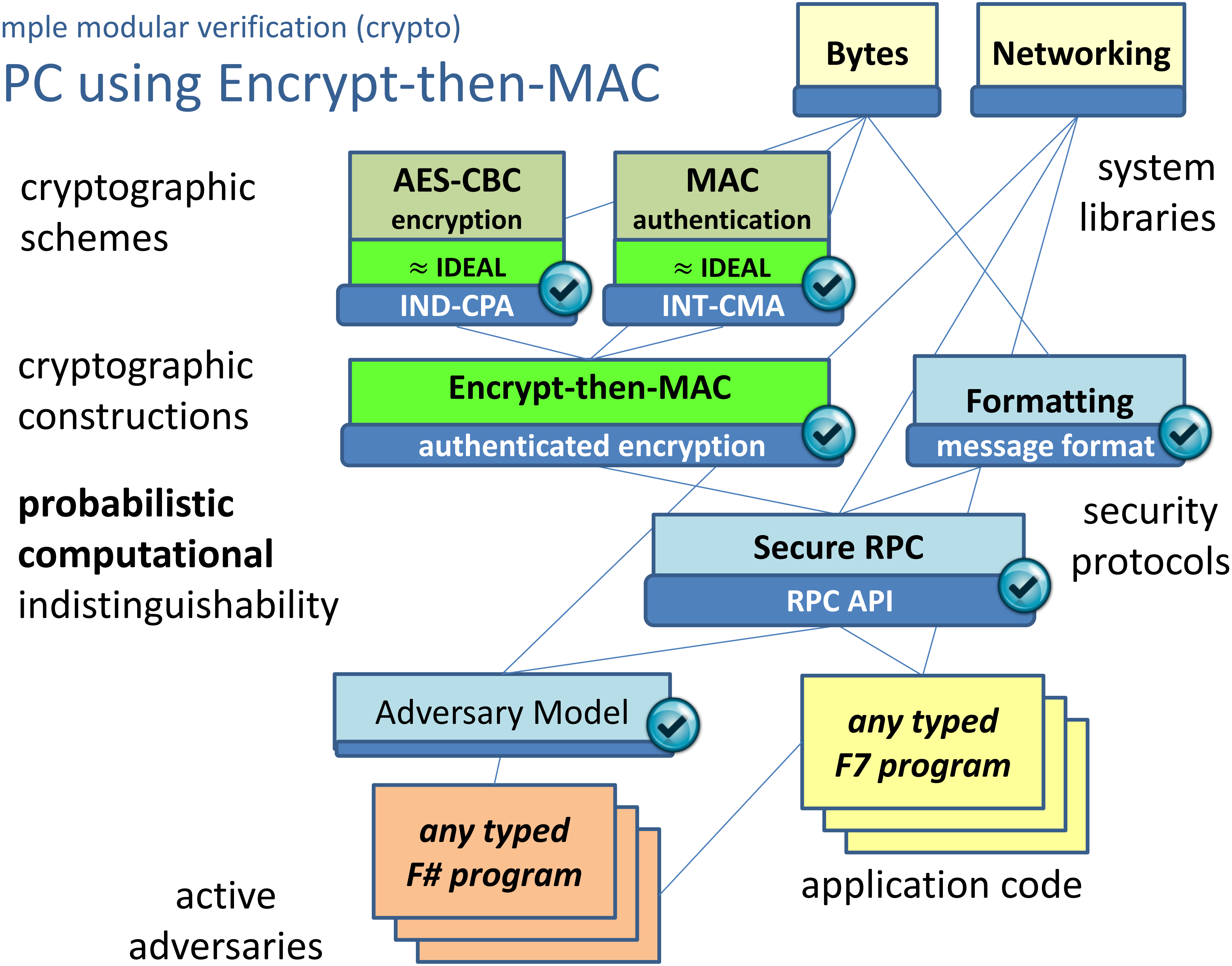
- We can program and verify sample crypto constructions such as hybrid encryption and encrypt-then-MAC

```
module HybridEnc
let pksize     = PKEnc.pksize     + SymEnc.ciphersize
let ciphersize = PKEnc.ciphersize + SymEnc.ciphersize

let keygen() = PKEnc.GEN()

let encrypt pk plain =
  let k = SymEnc.keygen ()
  append (PKEnc.encrypt pk k) (SymEnc.encrypt k plain)

let decrypt sk cipher =
  let c0,c1 = split PKEnc.ciphersize cipher
  SymEnc.decrypt (PKEnc.decrypt sk c0) c1
```

- We prove these constructions secure by typechecking against interfaces of Plain, SymEnc, and PKEnc

Sample modular verification (crypto)

RPC using Encrypt-then-MAC



Towards TLS: adding **Type Indexes**

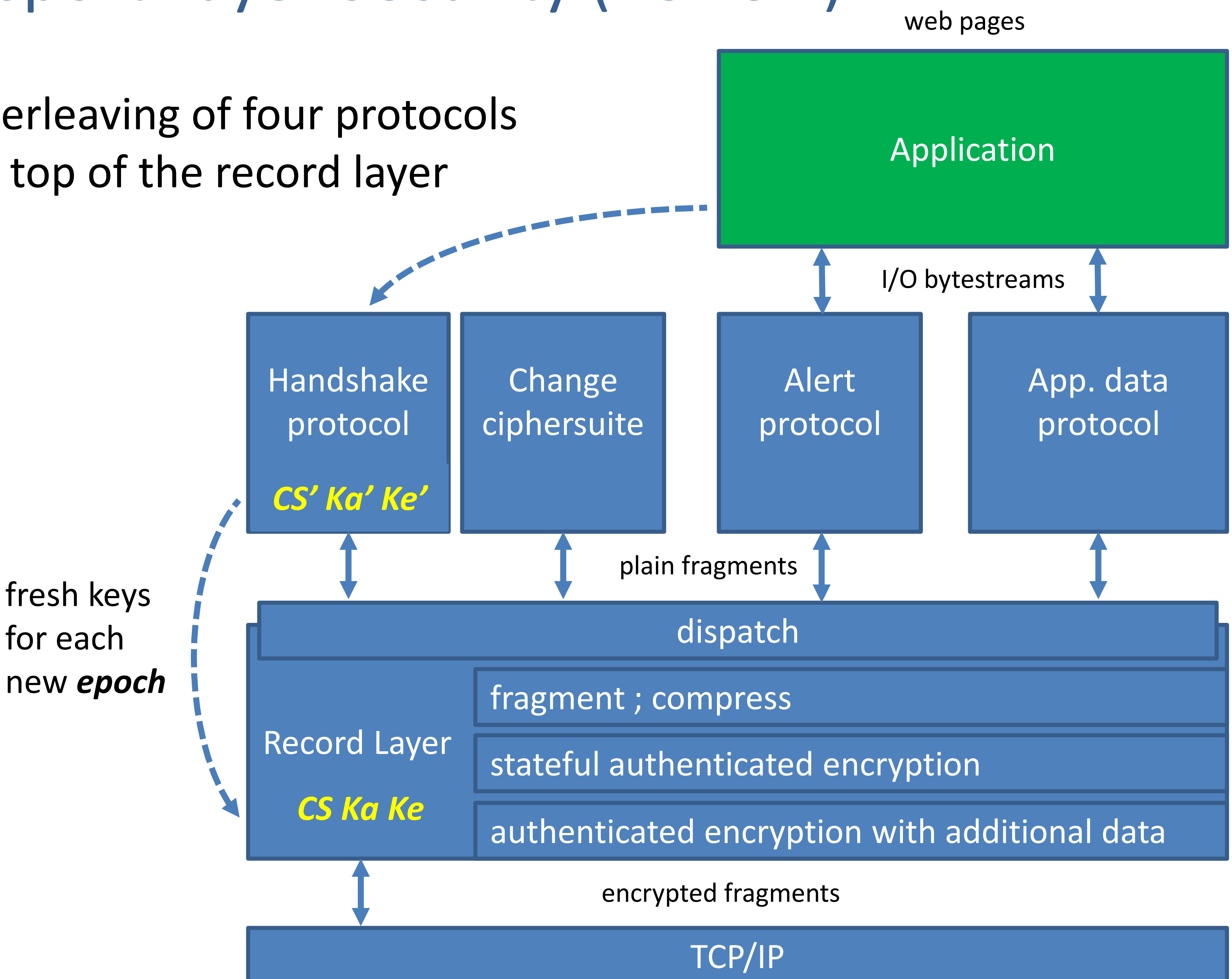
- Within TLS, we keep track of many keys, for different algorithms & sessions
- We use finer ideal functionalities that provide **conditional security** only for "good" keys
  - generated by algorithms assumed **computationally strong**; and
  - for sessions between **honest** participants (not those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

```
module AE
open Plain
type key (a:algorithm)(id:sessionID)
(…)
val keygen: a:algorithm -> s:sessionID -> key a s
val leak:
  a:algorithm -> s:sessionID {weak a || corrupt s} ->
  key a s -> bytes
val coerce:
  a:algorithm -> s:sessionID {weak a || corrupt s} ->
  bytes -> key a s
```

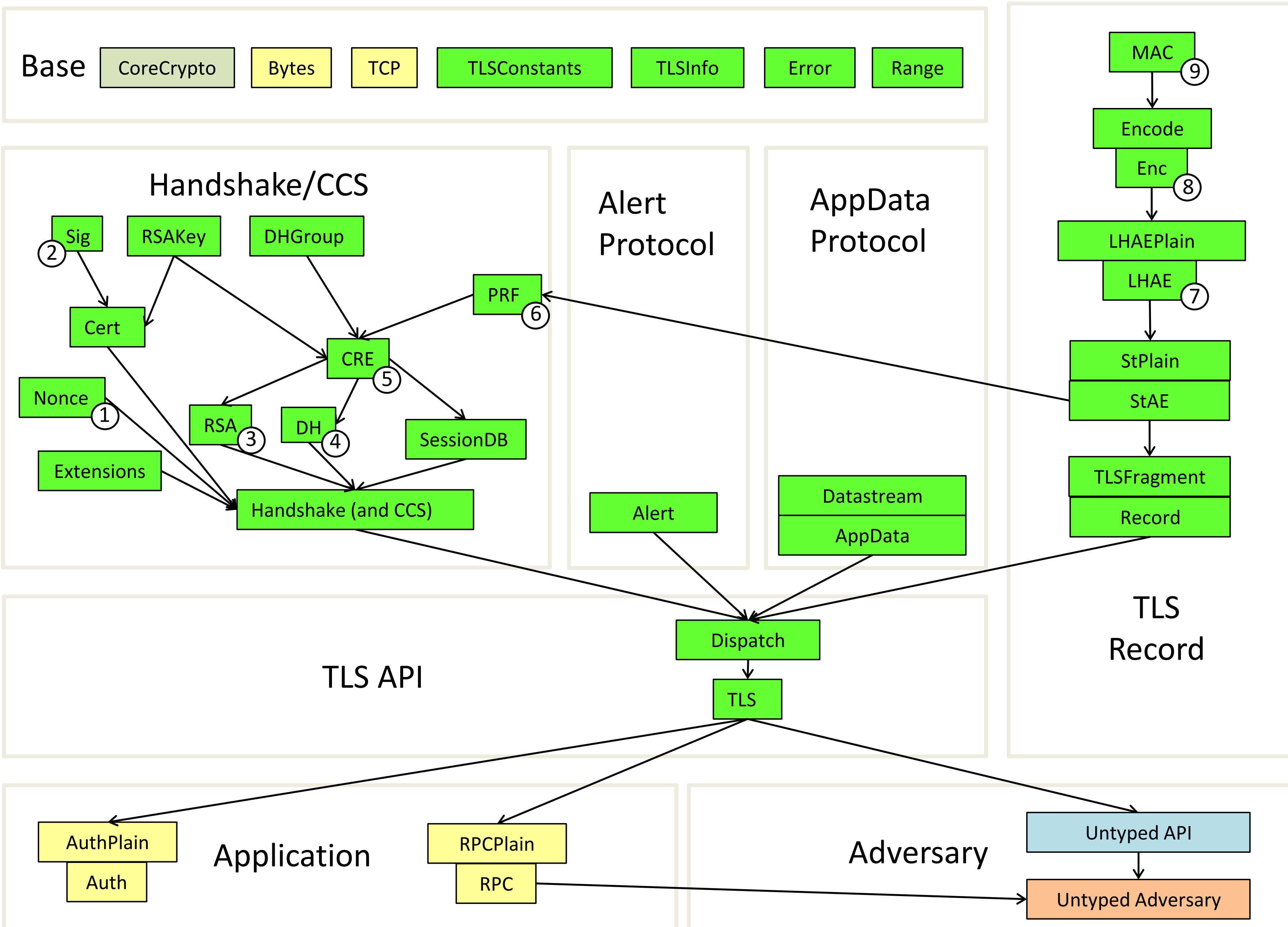
The type of the key generated for this algorithm used only for this session

Transport Layer Security (Review)

- Interleaving of four protocols on top of the record layer



Modular Architecture for miTLS



Transport Layer
(not the handshake)

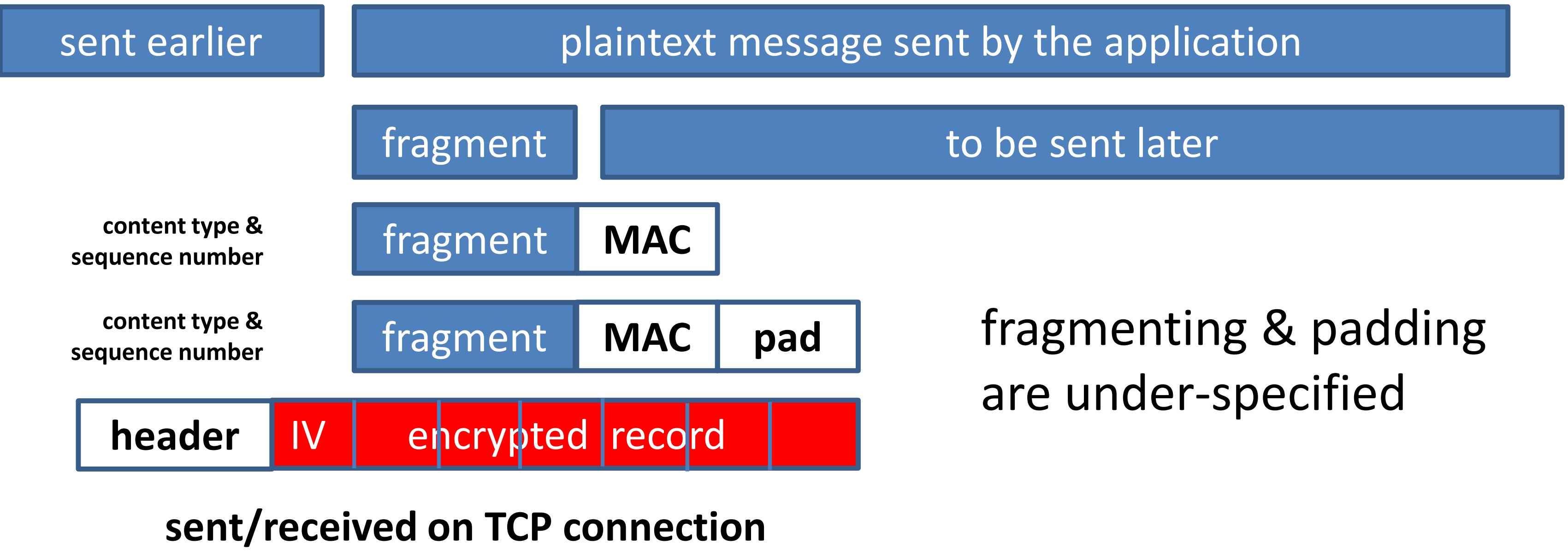
Verifying the miTLS reference implementation

agile
length-hiding
stateful

Authenticated Encryption

for fragment streams
with additional data

Fragment; MAC; Encode; then Encrypt

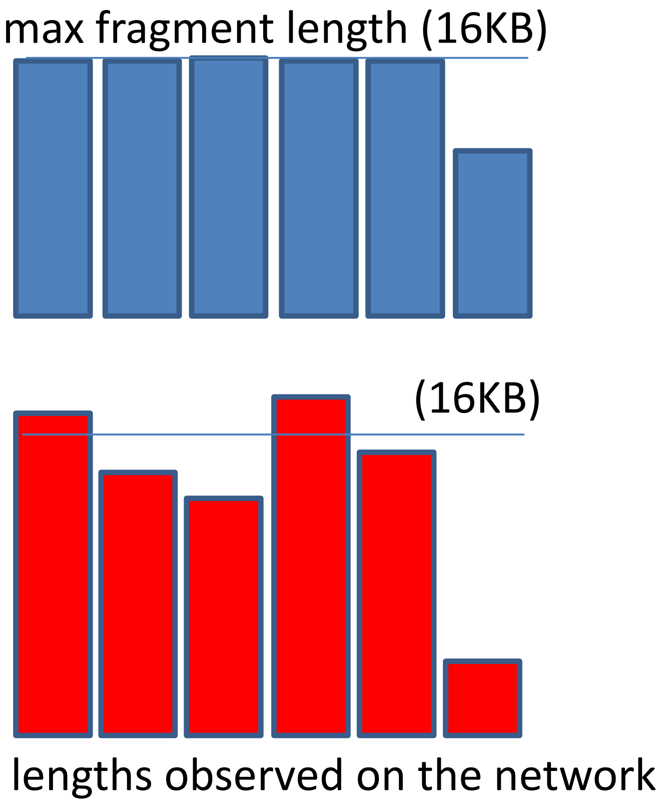


fragmenting & padding are under-specified

- TLS decodes the decrypted text *before* authentication; potentially leaking secret information (via "padding oracles")
- Security relies on joint ciphertext integrity (INT-CTXT)
  The proof is ad hoc (for CBC) and depends on |MAC| > |Block|
  (recent attack & proof by Paterson et al. at ASIACRYPT'11)

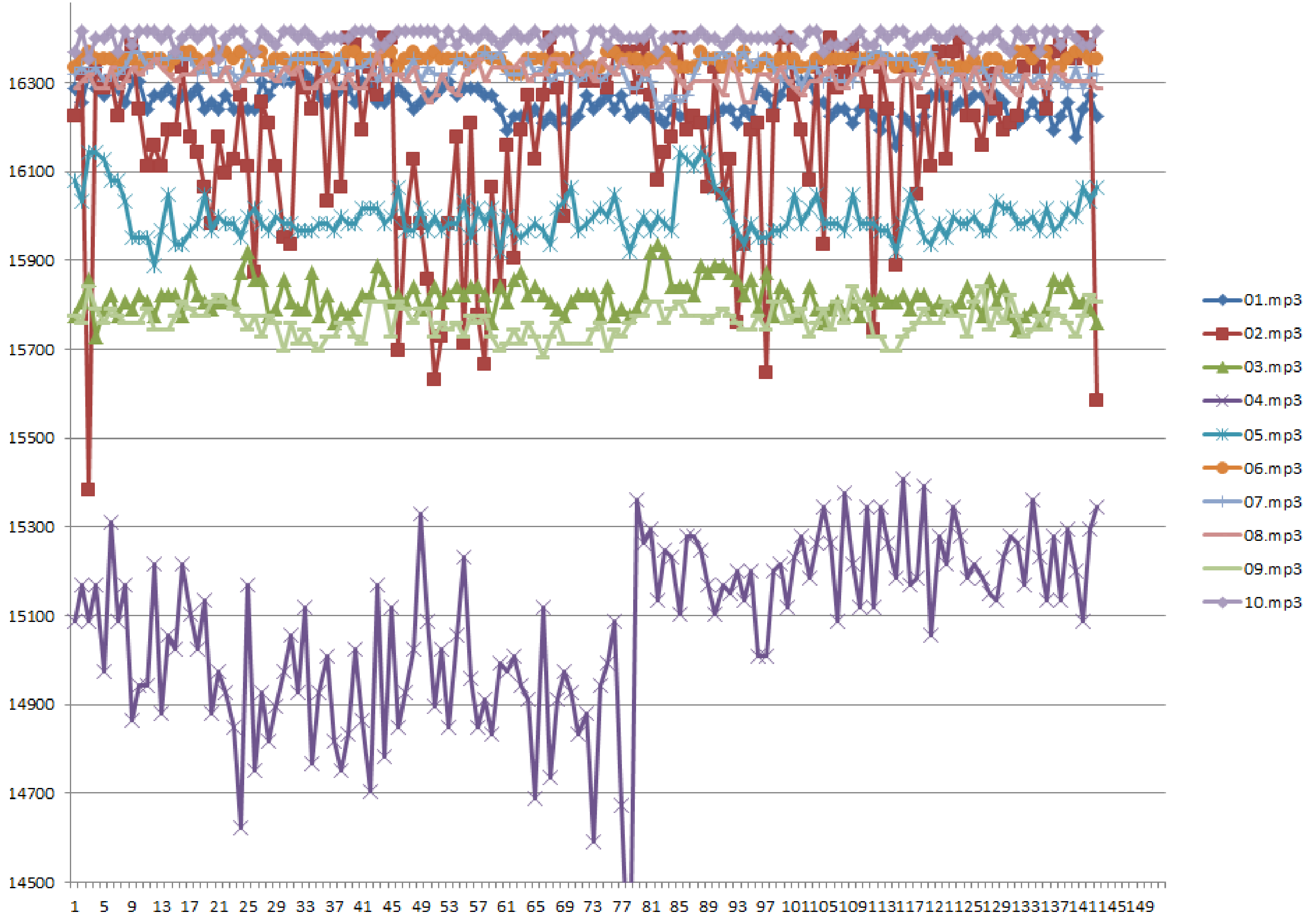
Fragment-then-Compress?

- Large messages are sliced into many fragments
- When encoded, each fragment is *independently* compressed
- An eavesdropper can record the sequence of fragment ciphertext lengths, and obtain **precise message fingerprints**
  - leaking much more than the total message length



Fragment-then-Compress?

- Experimental data: downloading songs over HTTPS:



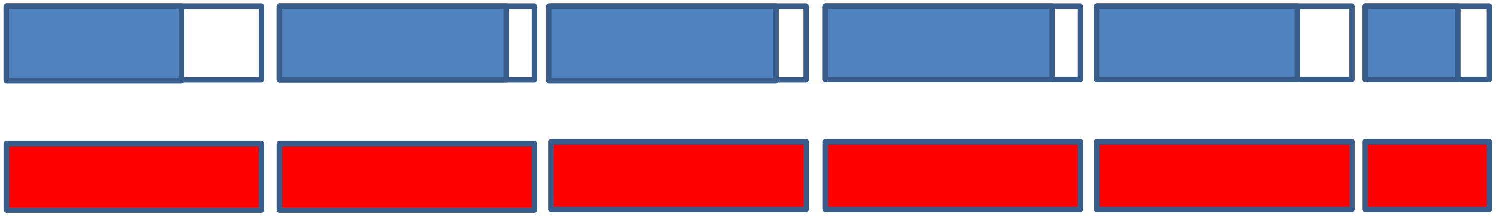
Our approach: disable compression, then

Hide secret lengths within public ranges

- The application chooses its own plaintext range, e.g. any secret URL of size 0..200 bytes

Formally, we index our type of plaintext fragments by their range & sequence number in the stream too. By typing, we check that

- Fragmentation and padding depends *only on the range & ciphersuite,* not on the secret message length & content



Abstract Plaintext Fragments

```
module PlainAEAD
type data (ki:KeyInfo) = b:bytes{…}
type fragment (ki:KeyInfo) (rg:range) (ad:data)

val leak:
  ki:KeyInfo{not(Safe ki))} -> rg:range -> ad:data ->
  fragment ki rg ad -> b:bytes{length b in rg}

val coerce:
  ki:KeyInfo{not(Safe ki)} -> rg:range -> ad:data ->
  b:bytes{length b in rg} -> fragment ki rg ad
```

- Abstract plaintext fragments are indexed by
  - **key info** including negotiated algorithms and connection info
  - **range** for the (secret) plaintext length
  - **additional data**, encoding e.g. TLS version & fragment number
- Type abstraction yields *conditional* security
  for plaintexts with safe key info

Authenticated Encryption in TLS

```
module PlainAEAD
type data (ki:KeyInfo) = b:bytes{…}
type fragment (ki:KeyInfo) (rg:range) (ad:data)
```

```
module AEAD
val encrypt:
  ki:KeyInfo -> key ki -> ad: data ki ->
  rg:range -> p: fragment ki rg ad -> c:cipher ki { CTXT ki ad p c }
val decrypt:
  ki:KeyInfo -> key ki-> ad: data ki ->
  c: cipher{length c = cipherLength ki} ->
  r: option (rg: range * fragment ki rg ad)
  { safe ki => forall p. r = Some p <=> CTXT ki ad p c }
```

- encryption & decryption with a safe index do not access the plaintext bytes (IND-CPA)
- decryption with a safe index succeeds on correctly-encrypted ciphertexts, returns an error otherwise (INT-CTXT)

Main TLS API

The TLS API & ideal functionality

- Our API is similar but more informative than mainstream APIs
  - We run on the caller's thread,
    letting the application do the scheduling & multiplexing
  - We give *more control* to the application code,
    and reflect *more information* from the underlying TLS state
    (lengths, fragmentation, authorization queries)
    - More precise security theorems
    - More flexibility for experiments & testing

- We can implement safe & simple APIs on top of it
- Sample applications using our API
    - Secure RPCs (with one connection per call)
    - Password-based client authentication
    - Basic HTTPS clients and servers (for interoperability testing)

our main TLS API (outline)

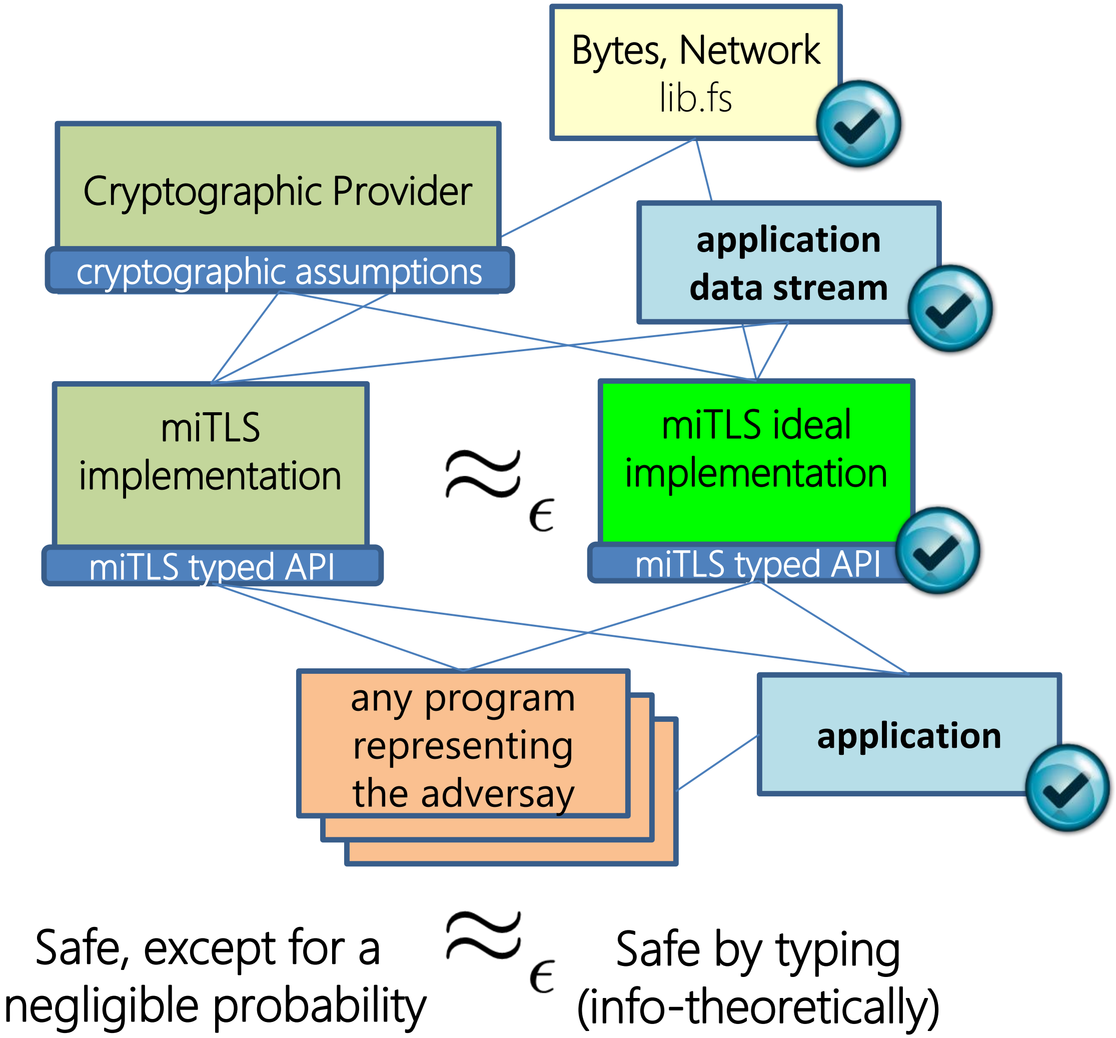
Each application provides its own plaintext module for data streams:

- Typing ensures secrecy and authenticity at safe indexes

Each application creates and runs session & connections in parallel

- Parameters select ciphersuites and certificates
- Results provide detailed information on the protocol state

```
type cn // for each local instance of the protocol

// creating new client and server instances
val connect: tcp -> params -> result (c:cn{role c = Client})
val accept:  Tcp -> params -> result (c:cn{role c = Server})

// triggering new handshakes, and closing connections
val rehandshake: c:cn{role c = Client} -> result (c:cn …)
val request:     c:cn{role c = Server} -> result (c:cn …)
val shutdown:    c:cn -> result tcp

// writing data
type ioresult_o (c:cn) (data:msg_o c) =
| WriteComplete of c':cn …
| WritePartial  of c':cn * rest:(;c') msg_o
| MustRead      of c':cn …
val write: c:cn -> data: msg_o c -> ioresult_o c data

// reading data
type ioresult_i (c:cn) =
| Read      of c':cn * data:(;c) msg_i
| CertQuery of c':cn …
| Handshake of c':cn …
| Close     of tcp
| Warning   of c':cn * a:alertDescription
| Fatal     of a:alertDescription
val read : c:cn -> ioresult_i c
```

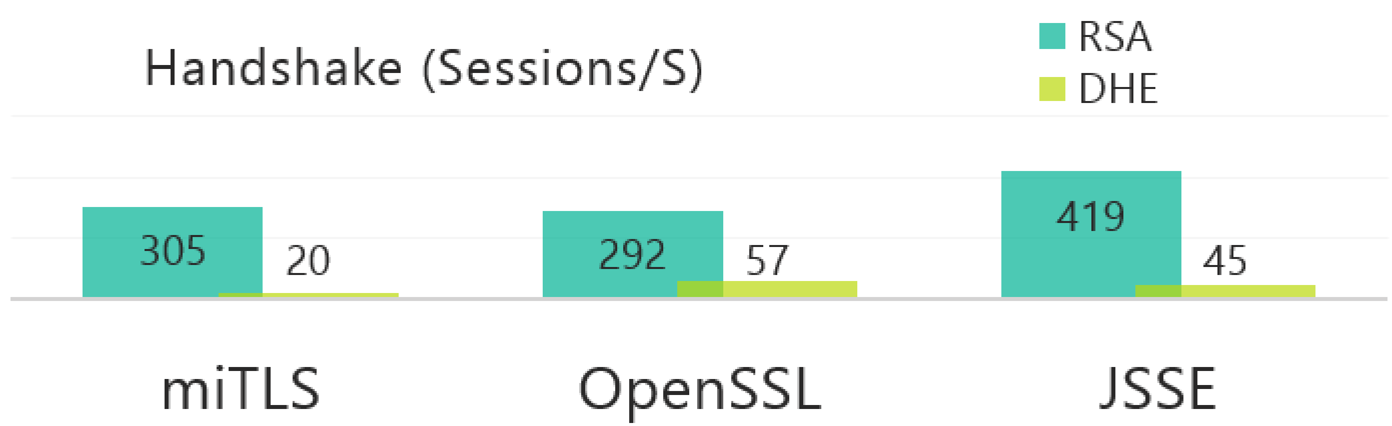


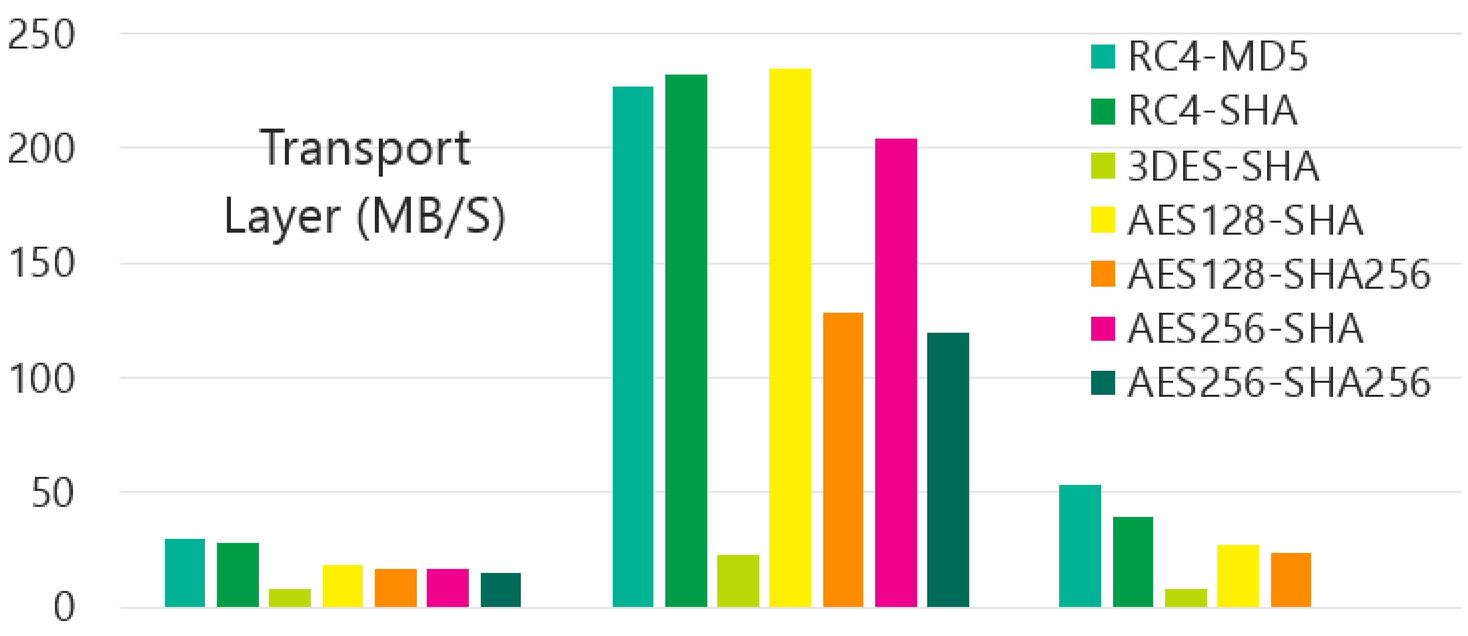
Main crypto result: **concrete TLS and ideal TLS are indistinguishable**

Our typed ideal API for TLS thus yields application security by typing

7,000 lines of F#
checked against
3,000 lines of F7
type annotations
+
3,000 lines of EasyCrypt
for the core key exchange

Interoperability & Performance





miTLS: A Verified Reference Implementation for TLS

We get **strong**, **usable**, **conditional** application security

We trust...

1. verification tools: F7, F*, Z3, EasyCrypt
   - now: mechanized theory using Coq/SSReflect
   - next: certified F* tools (POPL'12) and SMT solver
2. cryptographic assumptions
   - now: concrete reductions using Easycrypt
   - next: mechanized proofs with relational probabilistic F* (POPL'14)
3. the F# compiler and runtime: Windows and .NET
   - next: minimal TCB running e.g. on isolated core
4. core cryptographic providers
   - next: correctness for selected algorithms (elliptic curves)

OLDER SLIDES

An Implementation of TLS with Verified Cryptographic Security

Our ideal API provides strong, modular, usable, **conditional** application security by typing.

We trust

- **automated typechecking**: F7 and Z3
  - Now: mechanized type theory
  - Next: certified typechecker (F*, POPL'12) and SMT solver
- **cryptographic assumptions**, with handwritten proofs
  - Next: better concrete reductions, with tighter bounds
  - Next: mechanized proofs a la Certicrypt & Easycrypt
- **the F# compiler and runtime**: Windows and .NET
- **core cryptographic providers**
  - Next: correctness proofs for selected algorithms (elliptic curves)

We account for some side-channels, but not for timing analysis

An Implementation of TLS with Verified Cryptographic Security

Summary

- We verify protocol implementations by **typechecking**
  - Verification is modular
  - We use abstract types and refinements to specify cryptography
  - We capture standard (probabilistic polynomial time) assumptions
  - We precisely control composition using typed interfaces
  - Except for new crypto libraries, proofs are automated & fast
- We are working towards applications certified using Coq
  - New: self-certification for the typechecker
  - Next: cryptographic transformations behind typed interfaces
- Our approach and libraries are language-independent
  - So far we use F# & F7

Summary

- Implementation details cryptographically matter
  - We re-discovered classic attacks, found new ones
- We verify protocol implementations by **typechecking**
  - Verification is modular
  - We use abstract types and refinements to specify cryptography
  - We capture standard (probabilistic polynomial time) assumptions
  - We precisely control composition using typed interfaces
  - Except for new crypto, proofs are automated & fast
- Not yet another work on (simplified) TLS verification
  - A full-fledged, interoperable implementation,
verified down to concrete wire formats
  - Reduced to common computational cryptographic assumptions
  - Reasonable performance (but could be faster)
reuse of buffer space needs finer verification tools
- Our approach and libraries are language-independent
  - So far we use F# & F7
- Yet not the final word on TLS
  - Handshake; formal certification
  - RFCs and practice still evolving

Internal interface for LH-AEAD

- Ranges are public, lengths are secret
- *Conditional* security guarantees
- Constraints on inputs and outputs (excluding runtime error)
- IND CPA & INT-CTXT

```
Module AE_Plain
type (;ki:KeyInfo) data = b:bytes{…}
type (;ki:KeyInfo,rg:range,ad:data) plain

val COERCE:
  ki:KeyInfo{not(Safe(ki))} -> rg:range -> ad:data ->
  b:bytes{Length(b) in rg} ->(;ki,rg,ad) plain
val LEAK:
  ki:KeyInfo{not(Safe(ki))} -> rg:range -> ad:data ->
  (;ki,rg,ad) plain -> b:bytes{Length(b) in rg}
```

```
predicate CTXT of KeyInfo * data * plain * cipher
type (;ki:KeyInfo) key // possibly stateful
type (;ki:KeyInfo) keyrepr         = b:bytes{Length(b)=…}

val GEN:   ki:KeyInfo -> (;ki)key
val COERCE:
  ki:KeyInfo{not(Auth(ki))} -> (;ki)keyrepr -> (;ki)key
val LEAK:
  ki:KeyInfo{not(Auth(ki))} -> (;ki)key -> (;ki)keyrepr

val ENC: ki:KeyInfo -> (;ki)key -> ad:(;ki)data -> rg:range
  p:(;ki,rg,ad) plain -> c:cipher
  { Length(c)=RangeCipher(ki,rg) /\ CTXT(ki,ad,p,c) }

val DEC: ki:KeyInfo -> (;ki)key -> ad:(;ki)data ->
  c:cipher -> (;ki,CipherRange(ki,c),ad) plain Result
  { Auth(ki) => !p. res = Correct(p) <=> CTXT(ki,ad,p,c) }
```

The Handshake: Challenges

- Negotiates protocol version, handshake method and algorithms, authenticated encryption method and algorithms
- Authenticates peers from their certificates
- Derive connection keys

- Full handshake takes up to 3 rounds with 11 messages
- Abbreviated handshake often possible
  - Go straight to connection-key derivation
  - Do not negotiate and establish shared secret
- Key commitment
  - The “Finished” messages already use the key being established

Internal interface for Handshake & CCS protocols (simplified)

- New keys are delivered *before* handshake completion
- Negotiated parameters can be read off the state
- Refinements imply precise matching conversations

```
type (;r:role,o:config) state // for each local instance of the protocol
type (;ki:KeyInfo) fragment   // content type for the Handshake protocol
type (;ki:KeyInfo) ccs        // content type for the Handshake protocol

// Control Interface
val init:          r:role             -> o:config -> (;r     ,o) state
val resume:        si:SessionInfo     -> o:config -> (;Client,o) state
val rehandshake:  (;Client,idle) state -> o:config -> (;Client,o) state
val rekey:        (;Client,idle) state -> o:config -> (;Client,o) state
val request:      (;Server,idle) state -> o:config -> (;Server,o) state

// Network Interface (output)
type (;r:role,o:config,ki:KeyInfo) outgoing =
  | OutFragment of     (;r,o) state * (;ki) fragment option
  | OutCCS of        s:(;r,o) state * (;ki) ccs * (;OutKi(s)) ccs_data
  | OutComplete of   s:(;r,o) state {Complete(r,o,s)}
  | ...
val nextFragment:
  r:role -> o:config -> ki:KeyInfo ->
  (;r,o) state -> (;r,o,ki) outgoing

// Network Interface (input)
type (;r:role,o:config) incoming =
  | InTLSVersion of    (;r,o) state * ProtocolVersion
  | InComplete of    s:(;r,_) state {Complete(r,o,s)}
  | ...
val recvFragment:
  r:role -> o:config -> ki:KeyInfo ->
  (;r,o) state -> (;ki) fragment -> (;r,o) incoming
val recvCCS:
  r:role -> o:config -> ki:KeyInfo ->
  (;r,o) state -> (;ki) ccs -> s:(;r,o) state * (;InKi(s)) ccs_data
```

The Handshake, ideally

- Our interface abstracts over many details of the Handshake protocol
  - Handshake messages and their formats
  - Certificate formats and public key infrastructure
  - Database of past sessions, available for abbreviated handshakes

- A key index is ***safe*** when
  - Its ciphersuite is cryptographically strong; and
  - Its peer authentication materials are trustworthy
    e.g. the private key for the peer certificate
    is used only by compliant handshake sessions

- For instances with safe indexes, the (typed) idealized handshake
  - Generates fresh abstract keys instead of calling the concrete KDF
  - Drops "Complete" notifications not preceded by a send-Finished event with matching parameters in a compliant peer instance.

Our codebase for TLS 1.2

- We trust
  - The F# compiler
  - System libraries, including those for base cryptographic implementations (Windows CNG)
  - A rather complex runtime environment (.NET)
- LOCs and performance numbers
- In principle, our approach applies to C code (at some cost)

The TLS API (aka ideal functionality)

- Our API is similar but more precise than others, say OpenSSL
  - The RFC does not specify any API
  - We give more control to the application code, and reflect more details of the underlying TLS state (lengths and fragmentation; authorization queries,…)
    - More precise theorems
    - More flexibility for experiments & interop
  - We can implement more abstract APIs on top of it
  - Sample verified applications using our API
    - Secure RPCs (with one connection per call)
    - Basic HTTPS clients and servers (for interop testing)

Conclusions (TLS)

- Implementation details cryptographically matter
  - We re-discovered classic attacks, and found a few new ones
  - We need automation to relate standard crypto assumptions to concrete message processing
- Not yet another work on (simplified) TLS verification
  - A full-fledged, interoperable implementation of TLS 1.2
  - Verified down to concrete wire formats
  - Reduced to common computational cryptographic assumptions
  - Reasonable performance (but could be faster)
    reuse of buffer space needs finer verification tools
- Yet not the final word on TLS
  - RFCs and practice still evolving
  - Attackers outside our model: timing, differential power, etc
  - We stop at low-level crypto interfaces

Summary

- We verify protocol implementations by **typechecking**
  - Verification is modular
  - We use abstract types and refinements to specify cryptography
  - We capture standard (probabilistic polynomial time) assumptions
  - We precisely control composition using typed interfaces
  - Except for new crypto libraries, proofs are automated & fast
- We are working towards applications certified using Coq
  - New: self-certification for the typechecker
  - Next: cryptographic transformations behind typed interfaces
- Our approach and libraries are language-independent
  - So far we use F# & F7