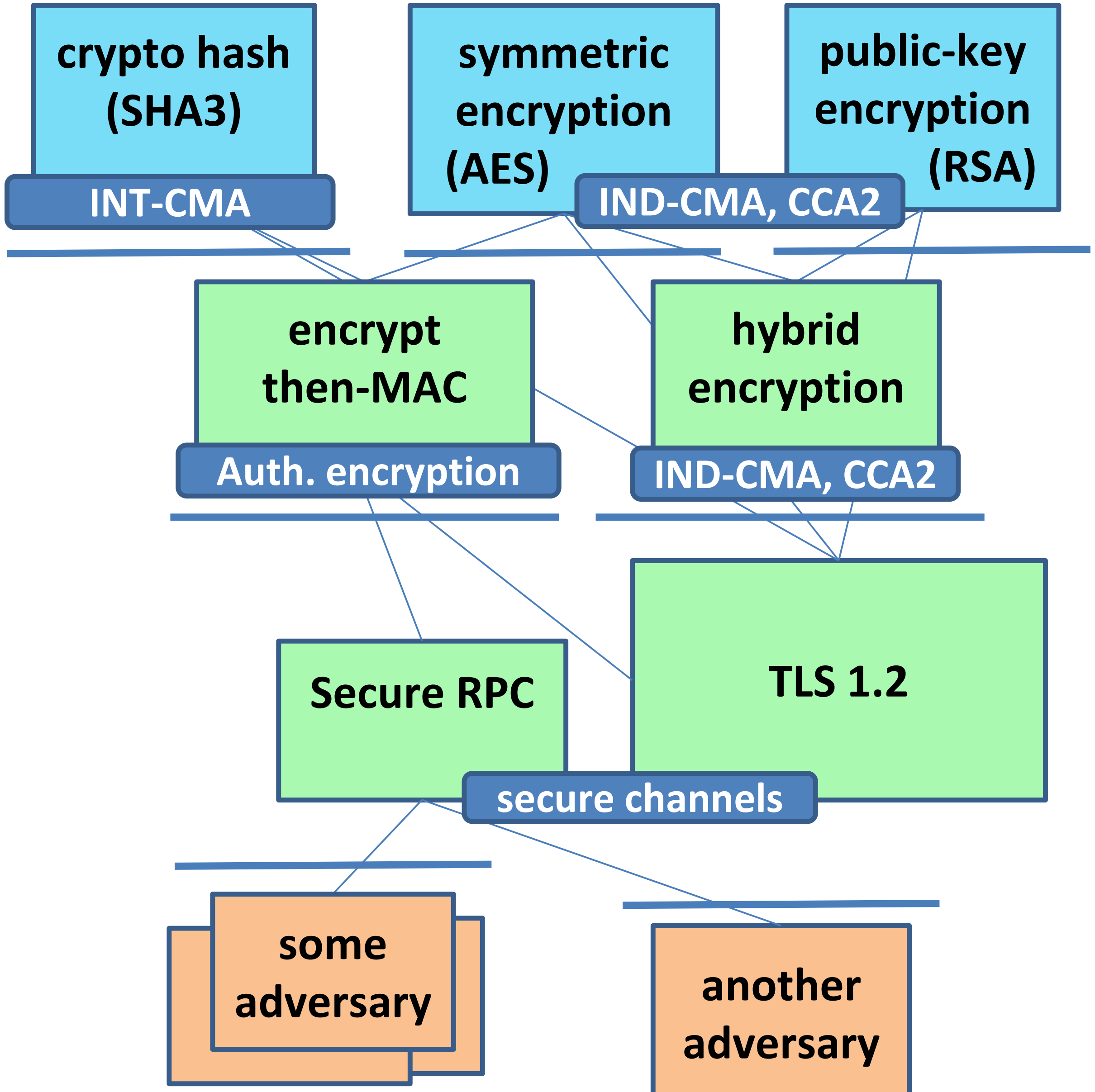
Concrete cryptographic security in F*

Modular Code-Based Crypto Verification



crypto hash (SHA3)

INT-CMA

symmetric encryption (AES)

public-key encryption (RSA)

IND-CMA, CCA2

encrypt then-MAC

Auth. encryption

hybrid encryption

IND-CMA, CCA2

Secure RPC

TLS 1.2

secure channels

some adversary

another adversary

crypto primitives

typed interfaces
(game-based security assumption)

crypto constructions

typed interfaces
(game-based security guarantees)

security protocols

typed interfaces
(attacker model)

active adversaries

Security programming example:

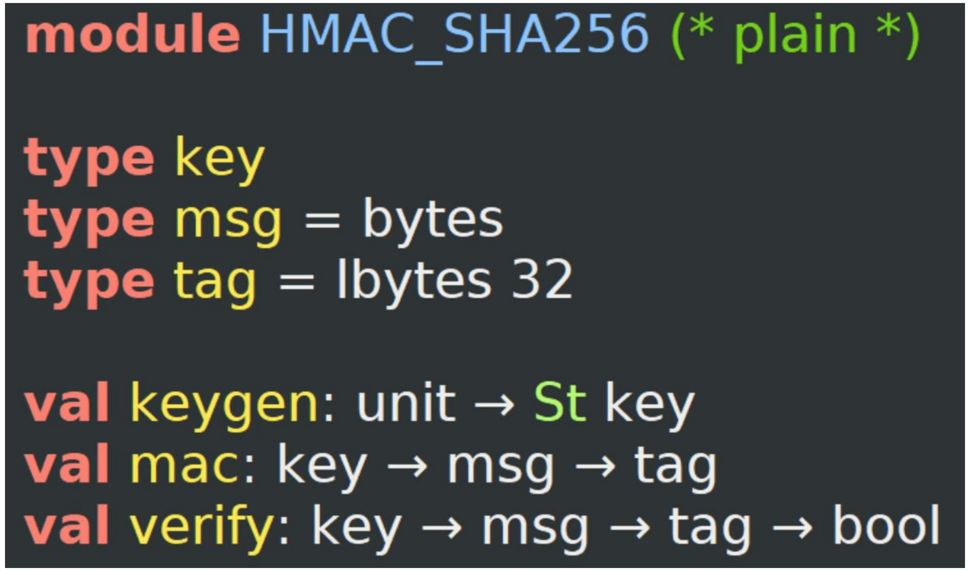
Access Control Lists

Example: access control for files

Untrusted client code may call a trusted,
defensive library for accessing files

- Trusted code sets up security policy as a typed API
- Typechecking client code enforces policy compliance
- Untrusted code deals with dynamic checks and errors
  - preconditions capture policy requirements
  - postconditions enable re-use of dynamic checks

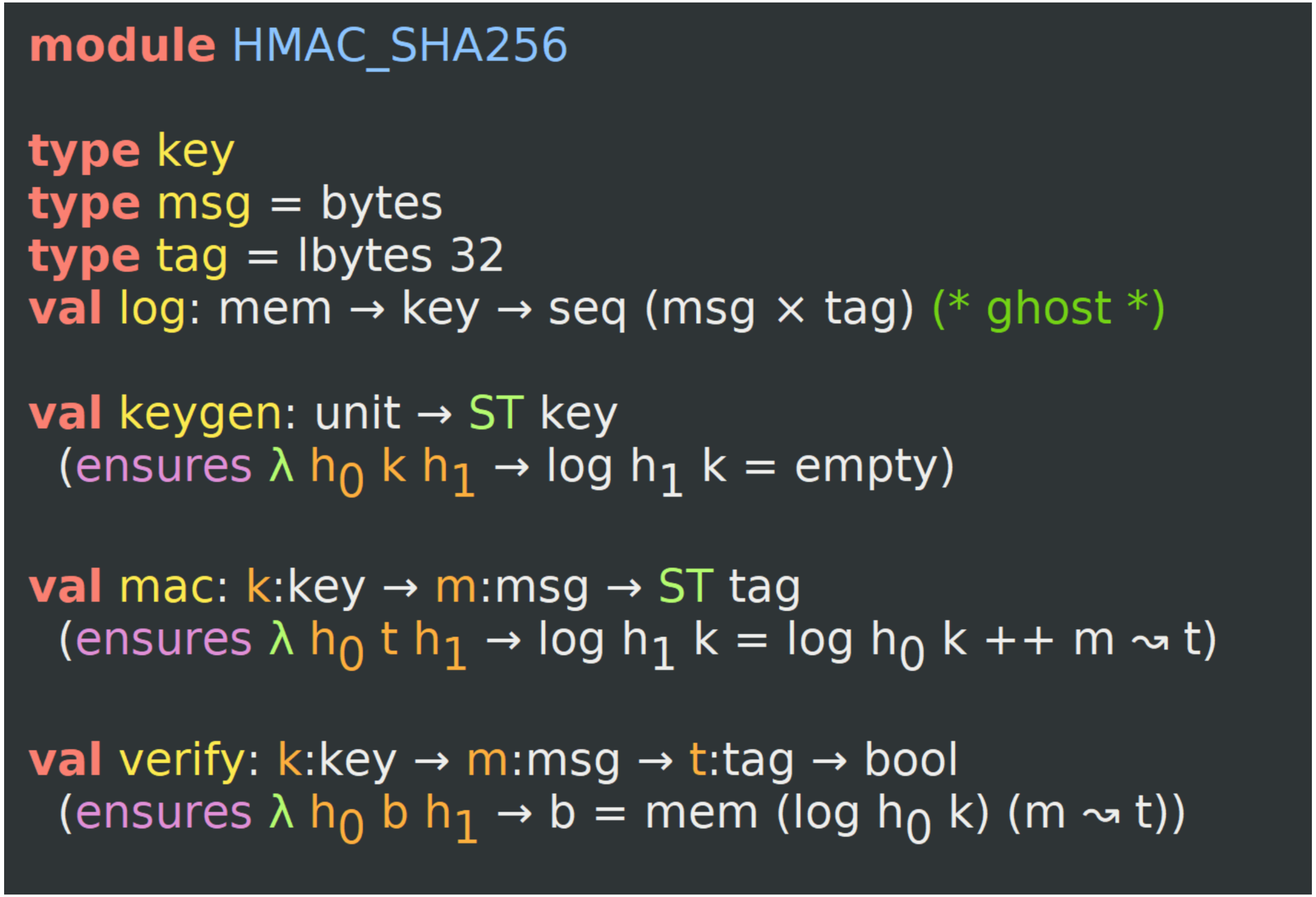
Cryptographic Integrity:
Message Authentication Codes (MAC)



```
module HMAC_SHA256 (* plain *)

type key
type msg = bytes
type tag = lbytes 32

val keygen: unit → St key
val mac: key → msg → tag
val verify: key → msg → tag → bool
```

This plain interface says nothing about the security of MACs!

Cryptographic Integrity: UF-CMA security (1/3)



```
module HMAC_SHA256

type key
type msg = bytes
type tag = lbytes 32
val log: mem → key → seq (msg × tag) (* ghost *)

val keygen: unit → ST key
  (ensures λ h0 k h1 → log h1 k = empty)

val mac: k:key → m:msg → ST tag
  (ensures λ h0 t h1 → log h1 k = log h0 k ++ m ↝ t)

val verify: k:key → m:msg → t:tag → bool
  (ensures λ h0 b h1 → b = mem (log h0 k) (m ↝ t))
```

This **ideal interface** uses a log to specify security

Great for F* verification.

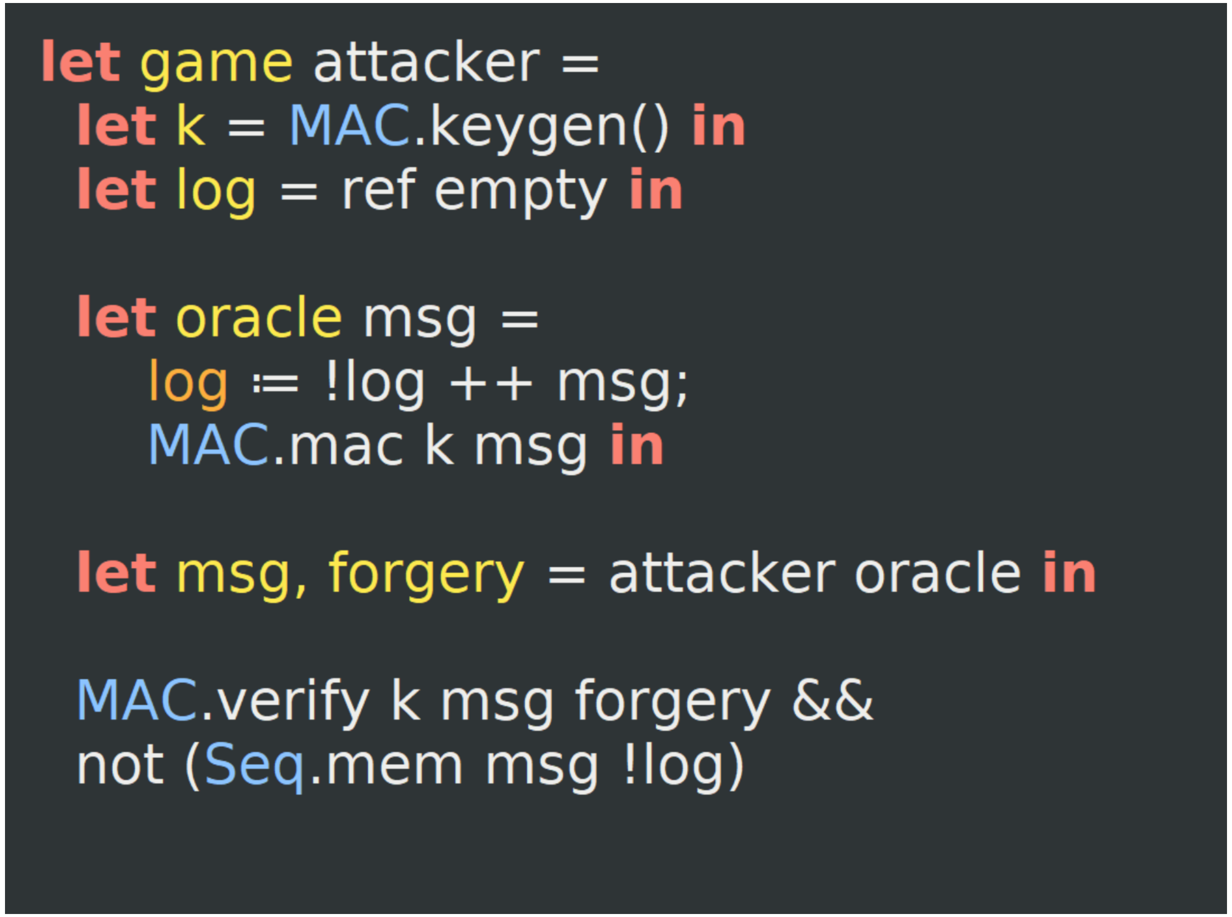
Unrealistic: tags can be guessed

Cryptographic Integrity: UF-CMA security (2/3)

Our ideal interface reflects the security of a **chosen-message game** [Goldwasser'88]

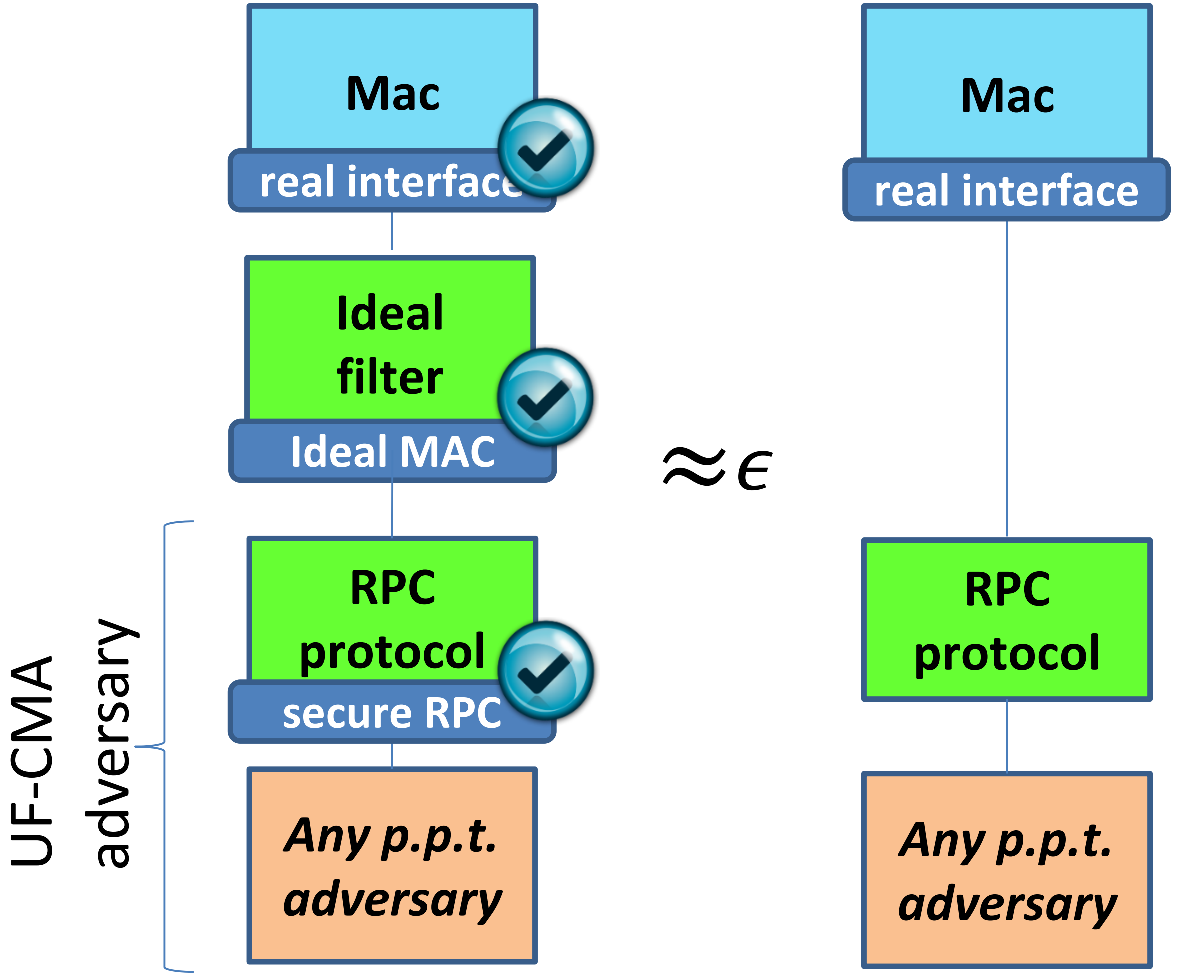
The MAC scheme is
ϵ-UF-CMA-secure
against a class of probabilistic, computationally bounded attackers when the game returns **true** with probability at most ϵ.

UF-CMA programmed in F*



```
let game attacker =
  let k = MAC.keygen() in
  let log = ref empty in

  let oracle msg =
      log := !log ++ msg;
      MAC.mac k msg in

  let msg, forgery = attacker oracle in

  MAC.verify k msg forgery &&
  not (Seq.mem msg !log)
```

Cryptographic Integrity: UF-CMA security (3/3)



ideal system

Mac

real interface

Ideal filter

Ideal MAC

RPC protocol

secure RPC

Any p.p.t. adversary

UF-CMA adversary

perfectly safe
by typing

$\approx \epsilon$

concrete system

Mac

real interface

RPC protocol

Any p.p.t. adversary

safe too,
with probability $\mathbf{1} - \boldsymbol{\epsilon}$

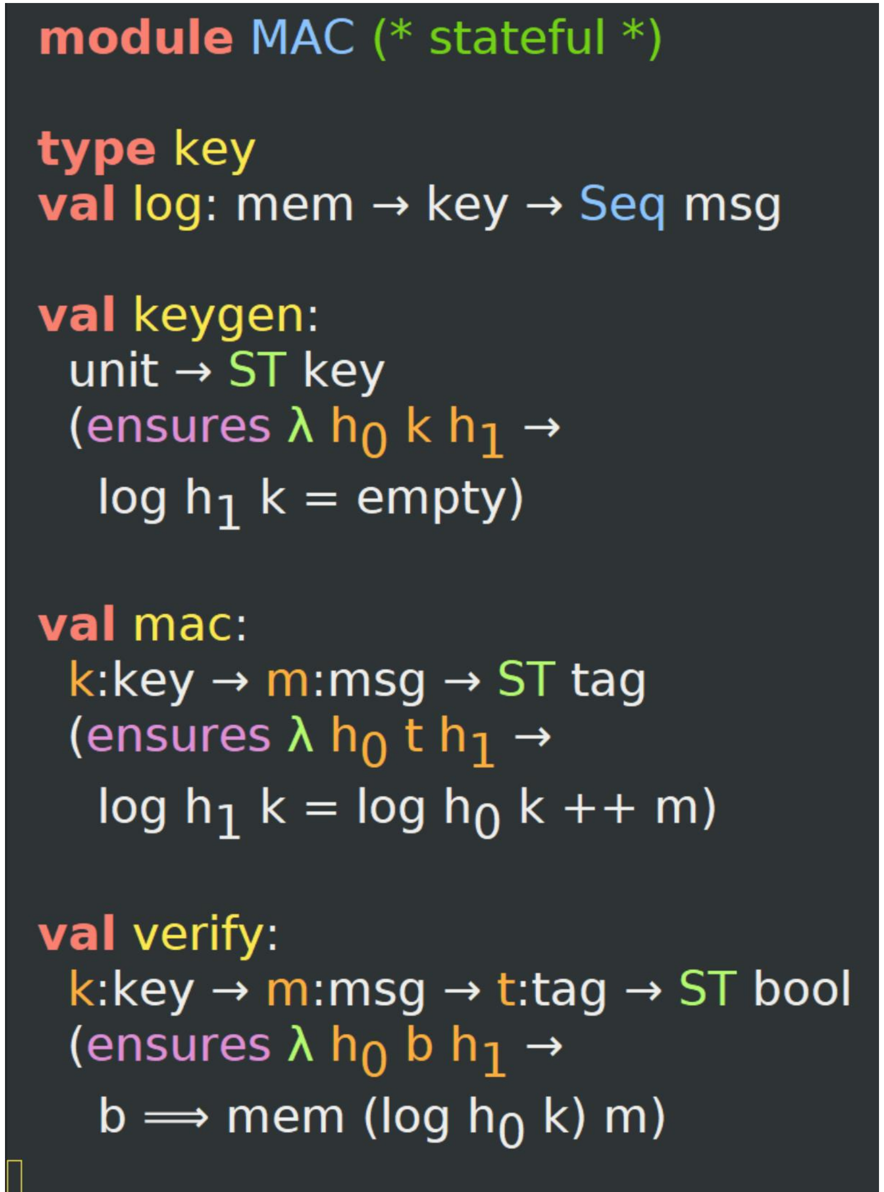
concrete algorithm
assumed UF-CMA computationally

log-based error correction
making VERIFY returns
false on forgeries

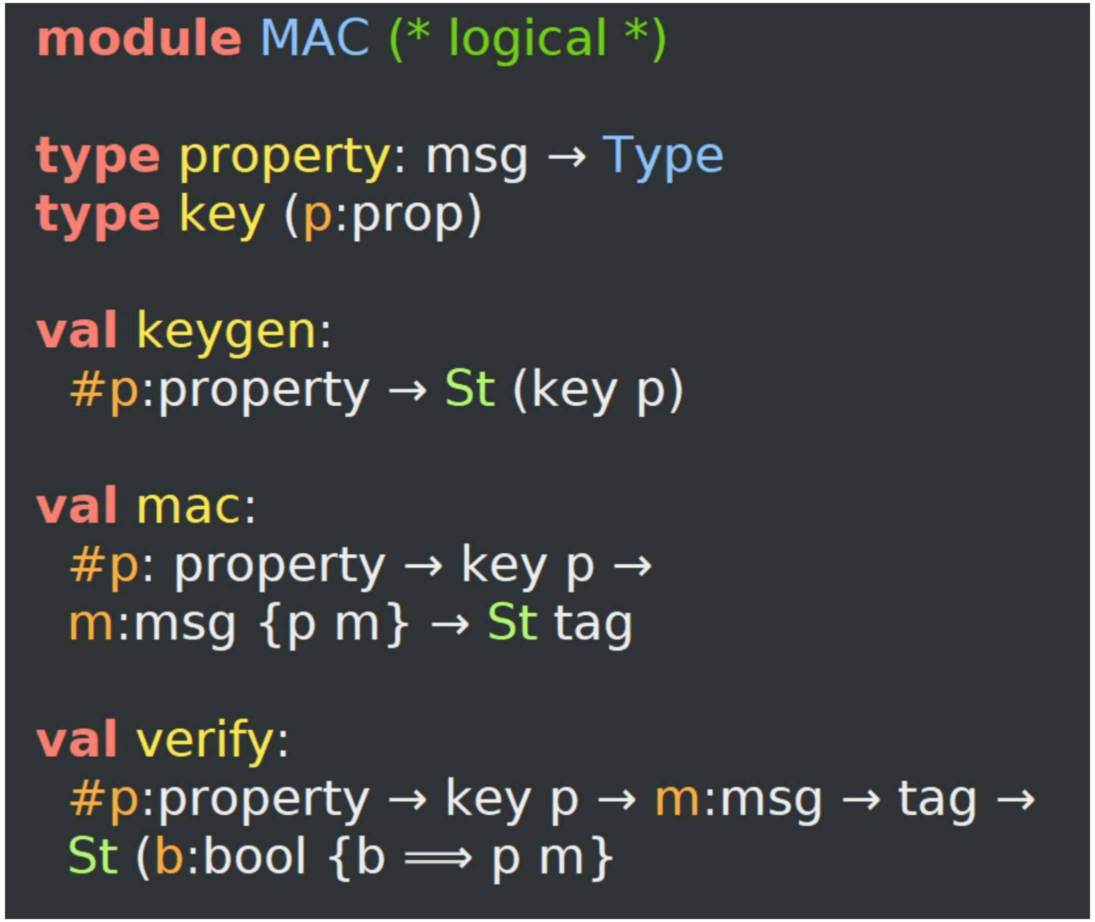
sample protocol
typed against
ideal MAC interface

protocol adversary
typed against
RPC interface

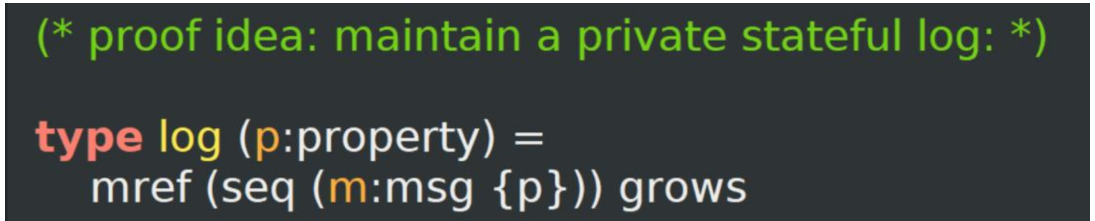
Cryptographic Integrity: Two styles for ideal MACs



```
module MAC (* stateful *)

type key
val log: mem → key → Seq msg

val keygen:
  unit → ST key
  (ensures λ h0 k h1 →
    log h1 k = empty)

val mac:
  k:key → m:msg → ST tag
  (ensures λ h0 t h1 →
    log h1 k = log h0 k ++ m)

val verify:
  k:key → m:msg → t:tag → ST bool
  (ensures λ h0 b h1 →
    b ⟹ mem (log h0 k) m)
```



```
module MAC (* logical *)

type property: msg → Type
type key (p:prop)

val keygen:
  #p:property → St (key p)

val mac:
  #p: property → key p →
  m:msg {p m} → St tag

val verify:
  #p:property → key p → m:msg → tag →
  St (b:bool {b ⟹ p m})
```



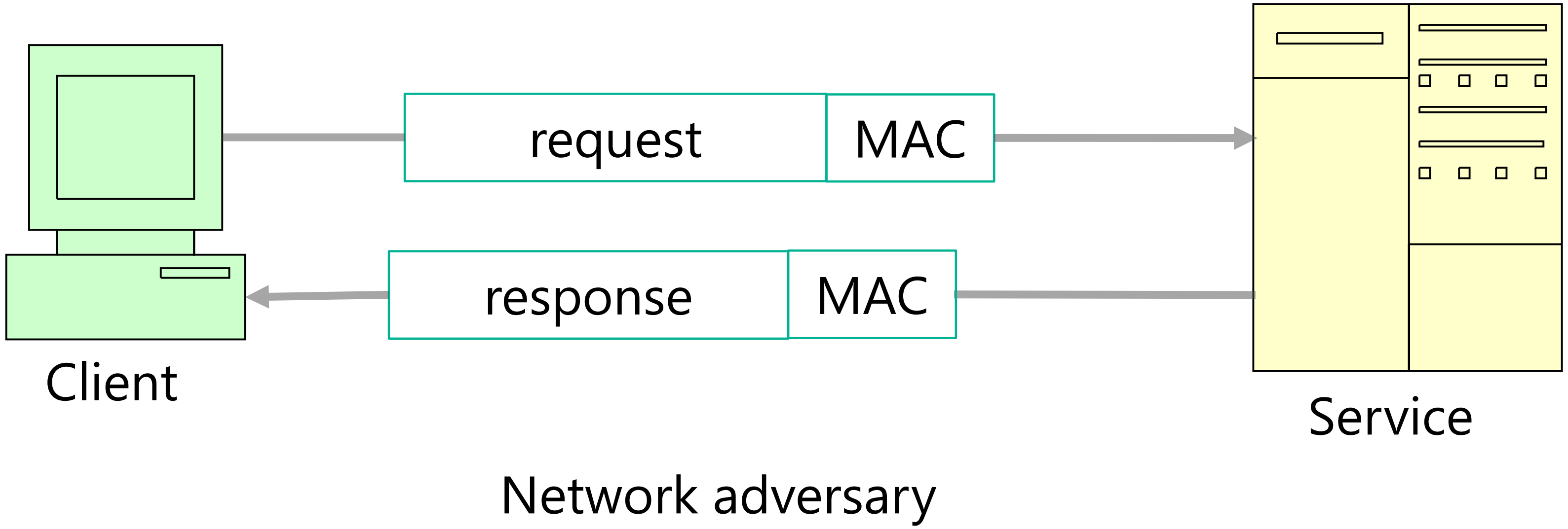
```
(* proof idea: maintain a private stateful log: *)

type log (p:property) =
   mref (seq (m:msg {p})) grows
```

Security programming example

Authenticated RPC

Authenticated RPC

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



request

MAC

response

MAC

Client

Service

Network adversary

Authenticated RPC: 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, $a\ b\ c\ \ldots$

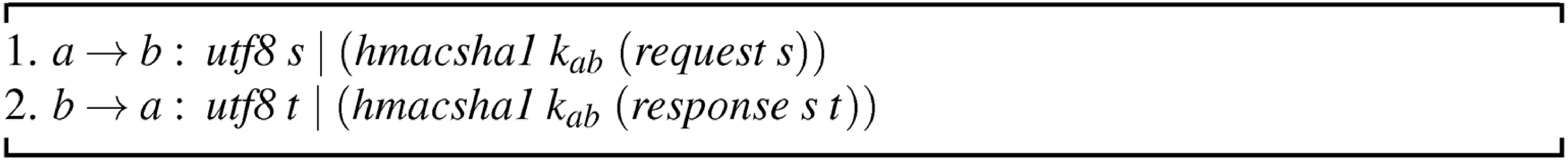
Our security goals:

- 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.

Authenticated RPC: Test



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

```
Connecting to localhost:8080
Sending {BgAyICsgMj9mhJa7iDAcW3Rrk...} (28 bytes)
Listening at ::1:8080
Received Request 2 + 2?
Sending {AQA0NccjcuL/WOaYS0GGtOtPm...} (23 bytes)
Received Response 4
```

Authenticated RPC: 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:

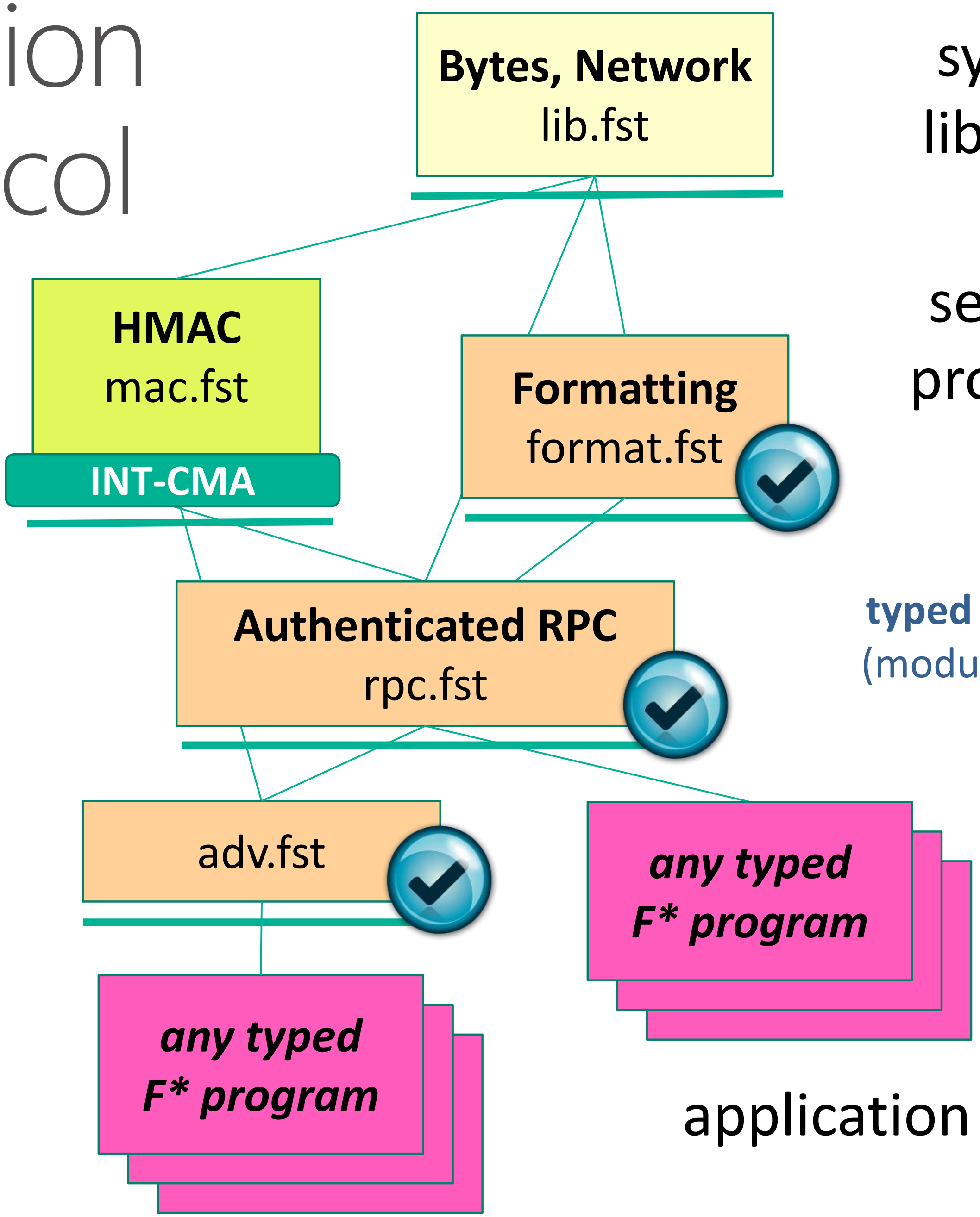
(1) The function *hmacsha1* is cryptographically secure,
so that MACs cannot be forged without knowing their key.

(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 sample protocol



Bytes, Network
lib.fst

system libraries

cryptographic primitives

HMAC
mac.fst
INT-CMA

Formatting
format.fst

security protocols

typed interfaces
(security assumptions)

Authenticated RPC
rpc.fst

typed interfaces
(modular design)

adv.fst

any typed F* program

plain typed interfaces
(attacker model)

any typed F* program

active adversaries

application code

Another sample crypto assumption

Collision Resistance

Hash Functions & Collision Resistance

For authentication, we often require
hash algorithms to be "computationally injective"

$$\forall\ (x\ y\!: \mathbf{bytes}).\, H(x) = H(y) \Longrightarrow x = y$$

This is modelled by maintaining an inverse, monotonic table from hash tags to hashed bytestrings

Hash Functions & Collision Resistance

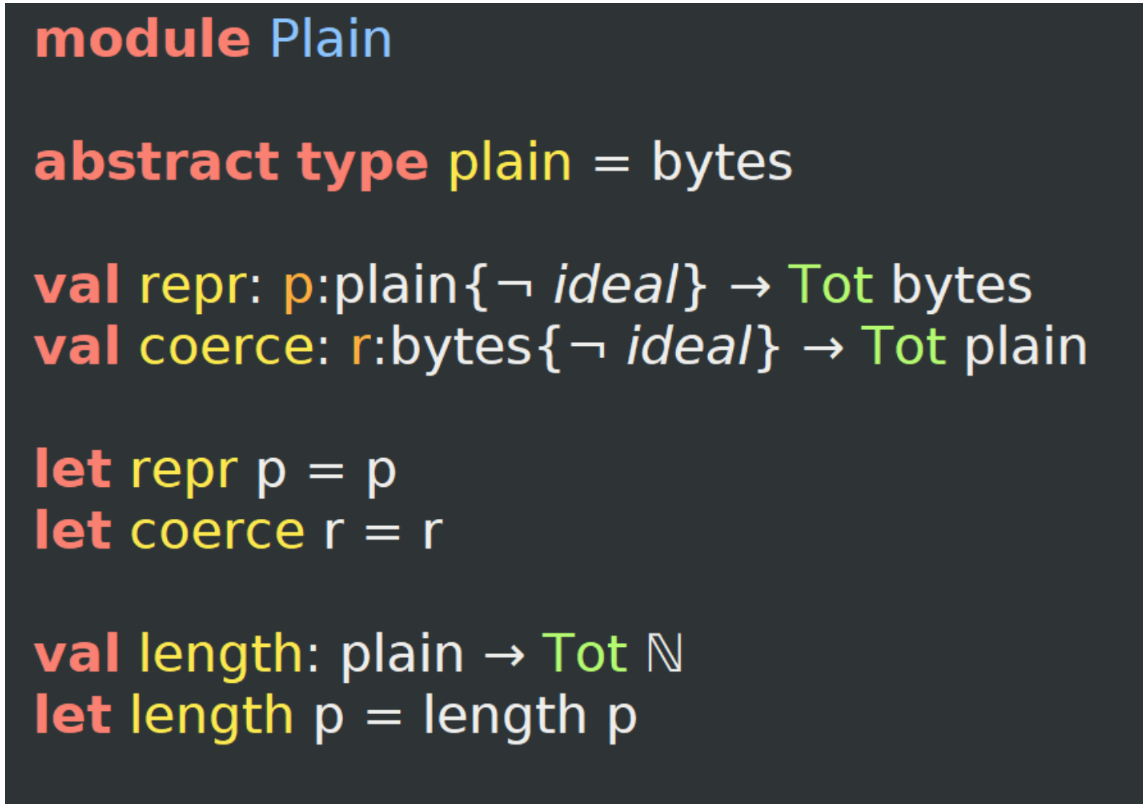
For authentication, we often require hash algorithms to be “computationally injective”

$$\forall\ (x\ y\text{: bytes hashed so far}).\, H(x) = H(y) \Longrightarrow x = y$$

This is modelled by maintaining an inverse, monotonic table from hash tags to hashed bytestrings

Authenticated Encryption

Cryptographic Confidentiality
Indistinguishability under Chosen-Plaintext Attacks



```
module Plain

abstract type plain = bytes

val repr: p:plain{¬ ideal} → Tot bytes
val coerce: r:bytes{¬ ideal} → Tot plain

let repr p = p
let coerce r = r

val length: plain → Tot ℕ
let length p = length p
```

We rely on type abstraction:

Ideal encryption never accesses the plaintext, is info-theoretically secure.

Authenticated Encryption: Game-based security assumption

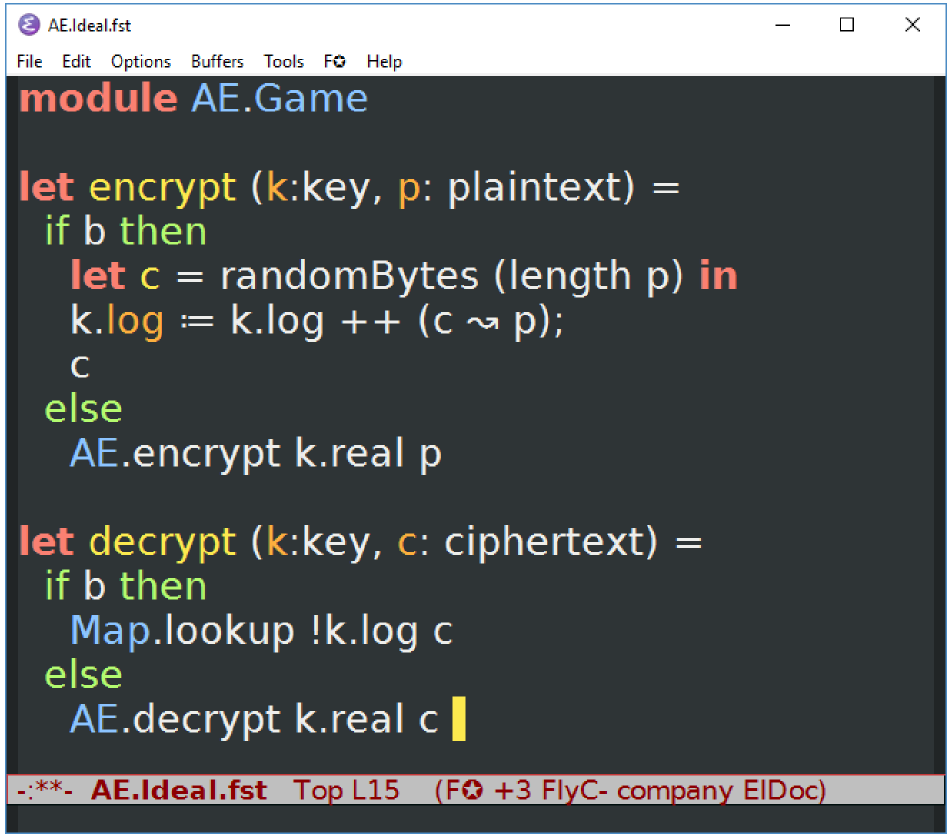
We program this game in F* parameterized by a real scheme AE and the flag b

```
Game Ae($\mathcal{A}$, AE)
    $b \xleftarrow{\$} \{0,1\};\ L \leftarrow \varnothing;\ k \xleftarrow{\$} \mathsf{AE.keygen}()$
    $b' \leftarrow \mathcal{A}^{\mathsf{Encrypt},\mathsf{Decrypt}}();\ \mathbf{return}\ (b \overset{?}{=} b')$
```

```
Oracle Encrypt($p$)
    if $b$ then $c \xleftarrow{\$} \mathsf{byte}^{\ell_c};\ L[c] \leftarrow p$
    else $c \leftarrow \mathsf{AE.encrypt}\ k\ p$
    return $c$
```

```
Oracle Decrypt($c$)
    if $b$ then $p \leftarrow L[c]$
    else $p \leftarrow \mathsf{AE.decrypt}\ k\ c$
    return $p$
```

Definition 1 (AE-security): Given AE, let $\epsilon_{\mathsf{Ae}}(\mathcal{A}[q_e, q_d])$ be the advantage of an adversary $\mathcal{A}$ that makes q_e queries to Encrypt and q_d queries to Decrypt in the $\mathsf{Ae}^b(\mathsf{AE})$ game.



```
module AE.Game

let encrypt (k:key, p: plaintext) =
  if b then
    let c = randomBytes (length p) in
    k.log ≔ k.log ++ (c ↝ p);
    c
  else
    AE.encrypt k.real p

let decrypt (k:key, c: ciphertext) =
  if b then
    Map.lookup !k.log c
  else
    AE.decrypt k.real c
```

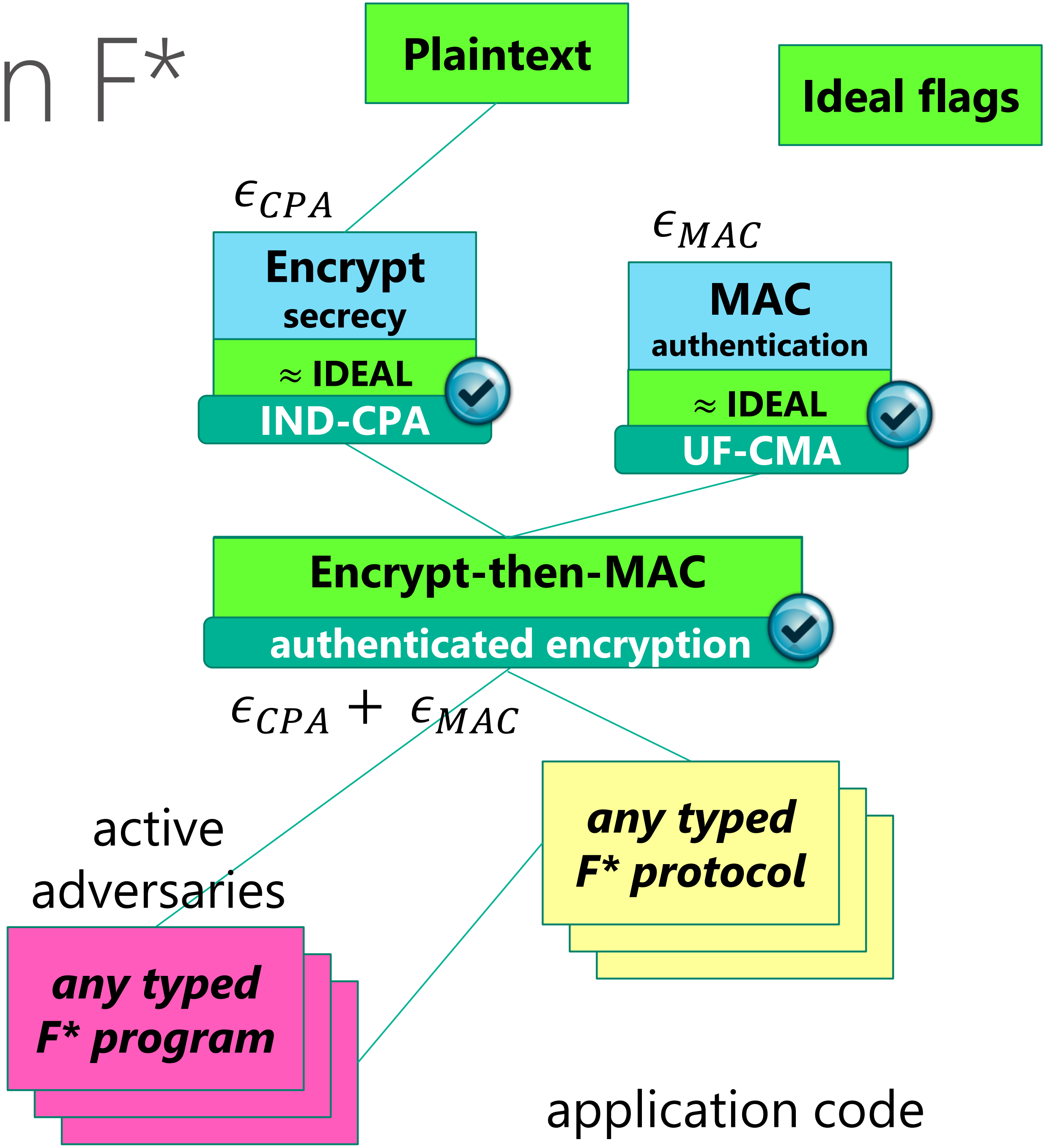
```
-:**-  AE.Ideal.fst   Top L15    (F✪ +3 FlyC- company ElDoc)
```

We capture its security using types to keep track of the content of the log

Encrypt-then-MAC in F*

Code follows the structure of the construction & its proof

- For each functionality, we have a separate module
- …and an interface that captures its security
- Idealization is conditional, controlled by flags whose values are unknown at verification-time
- The top-level proof consists of gradually setting flags for all crypto assumptions



Plaintext

Ideal flags

ϵ_{CPA}

Encrypt
secrecy
≈ IDEAL
IND-CPA

ϵ_{MAC}

MAC
authentication
≈ IDEAL
UF-CMA

Encrypt-then-MAC
authenticated encryption

$\epsilon_{CPA} + \epsilon_{MAC}$

active adversaries

any typed F* protocol

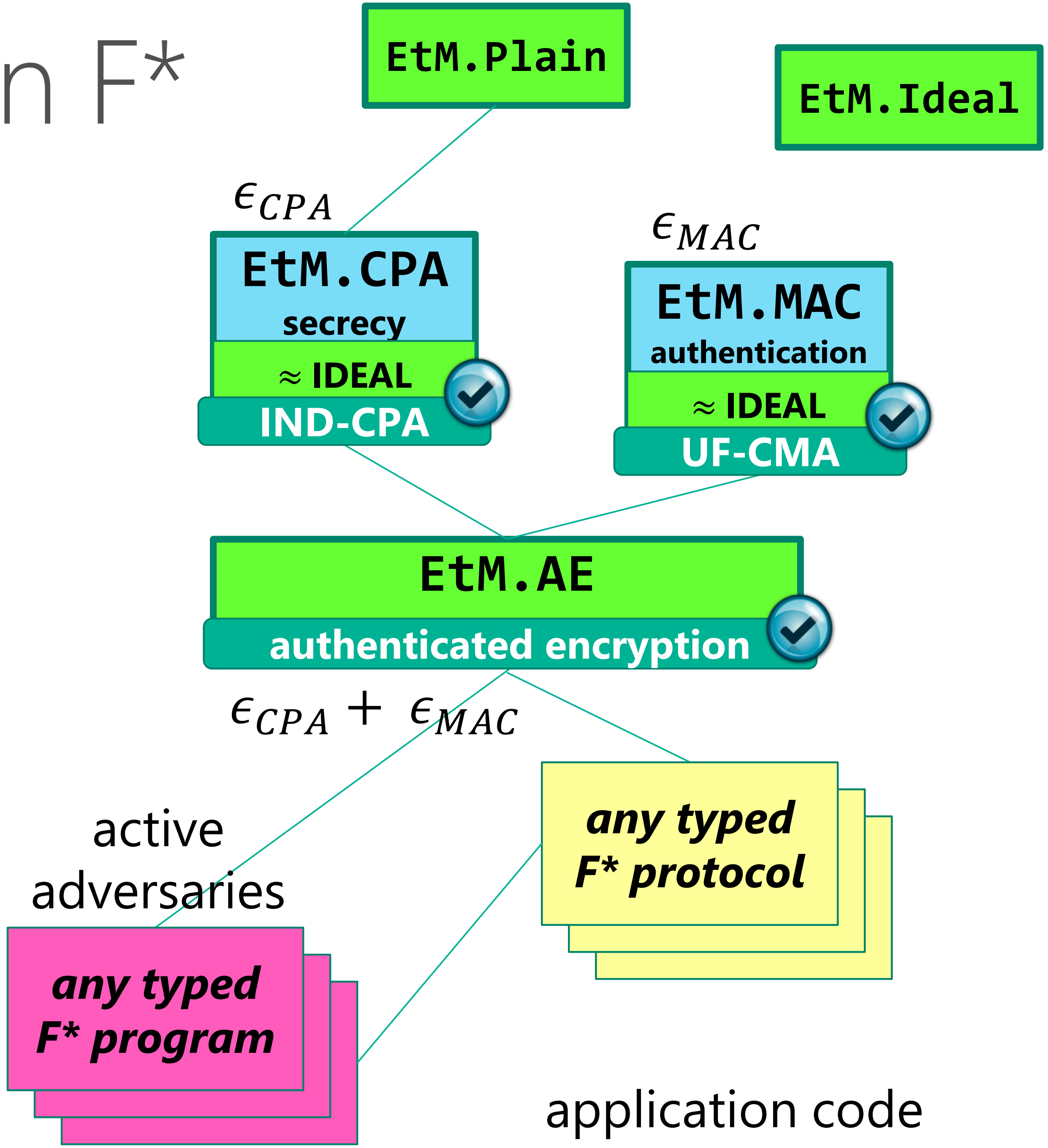
any typed F* program

application code

Encrypt-then-MAC in F*

Code follows the structure
of the construction & its proof

- For each functionality,
  we have a separate module
- …and an interface
  that captures its security
- Idealization is conditional,
  controlled by flags whose values
  are unknown at verification-time
- The top-level proof consists of
  gradually setting flags for all
  crypto assumptions



`EtM.Plain`

`EtM.Ideal`

ϵ_{CPA}

`EtM.CPA`
secrecy
≈ IDEAL
IND-CPA

ϵ_{MAC}

`EtM.MAC`
authentication
≈ IDEAL
UF-CMA

`EtM.AE`
authenticated encryption

$\epsilon_{CPA} + \epsilon_{MAC}$

active
adversaries

any typed F* program

any typed F* protocol

application code