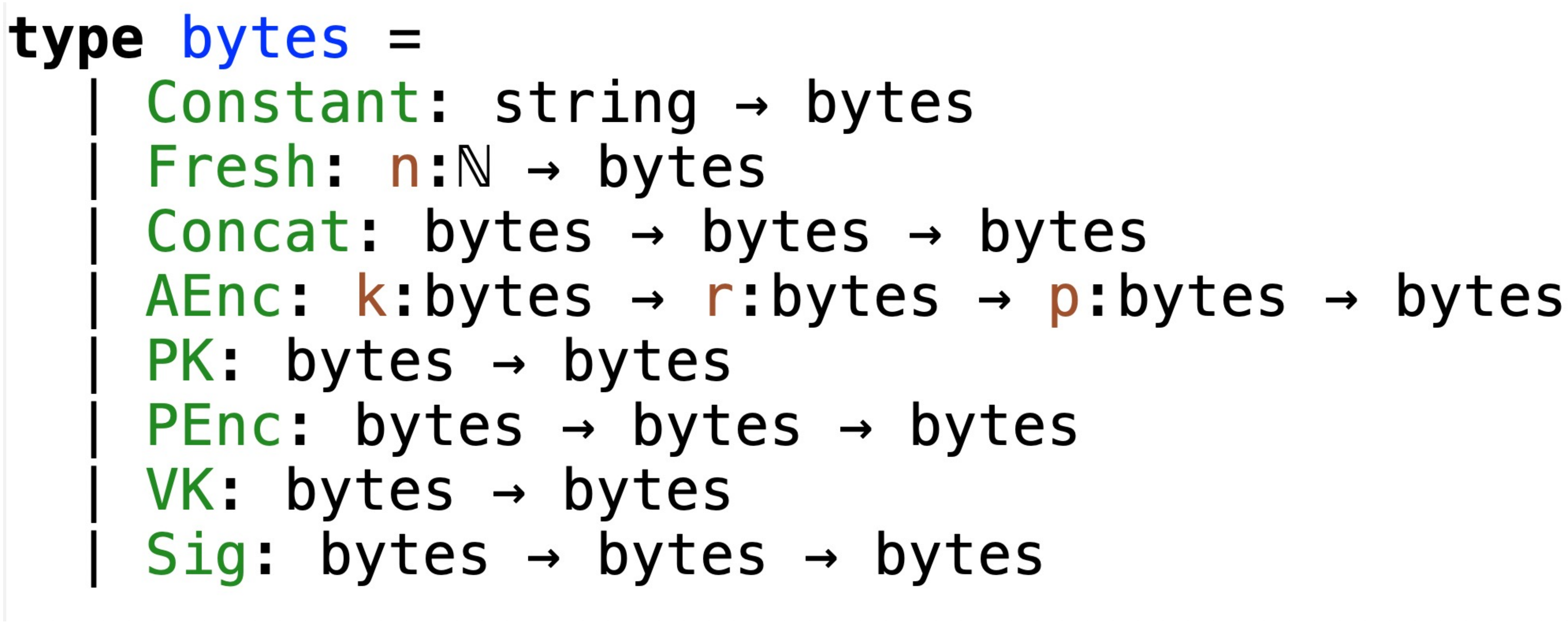
Verifying Protocols with F*

Karthikeyan Bhargavan

MPRI 2:30, September 2022

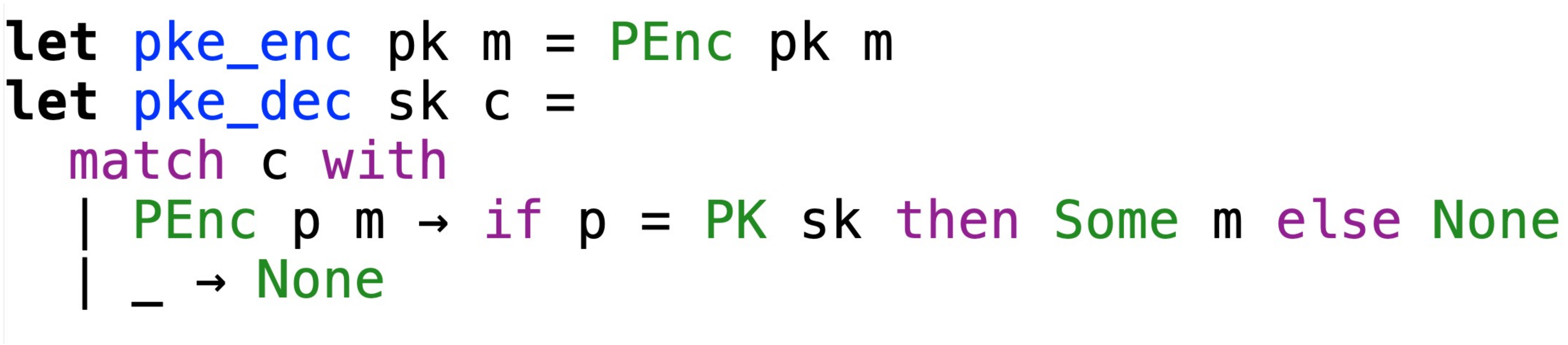
Modeling Crypto
and Protocol Execution

A symbolic model of bitstrings

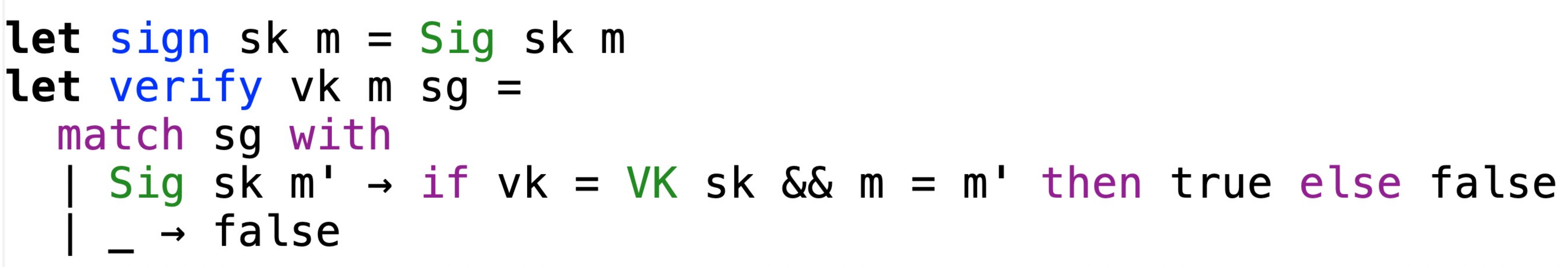


```
type bytes =
  | Constant: string → bytes
  | Fresh: n:ℕ → bytes
  | Concat: bytes → bytes → bytes
  | AEnc: k:bytes → r:bytes → p:bytes → bytes
  | PK: bytes → bytes
  | PEnc: bytes → bytes → bytes
  | VK: bytes → bytes
  | Sig: bytes → bytes → bytes
```

A symbolic model of crypto



```
let pke_enc pk m = PEnc pk m
let pke_dec sk c =
  match c with
  | PEnc p m → if p = PK sk then Some m else None
  | _ → None
```



```
let sign sk m = Sig sk m
let verify vk m sg =
  match sg with
  | Sig sk m' → if vk = VK sk && m = m' then true else false
  | _ → false
```

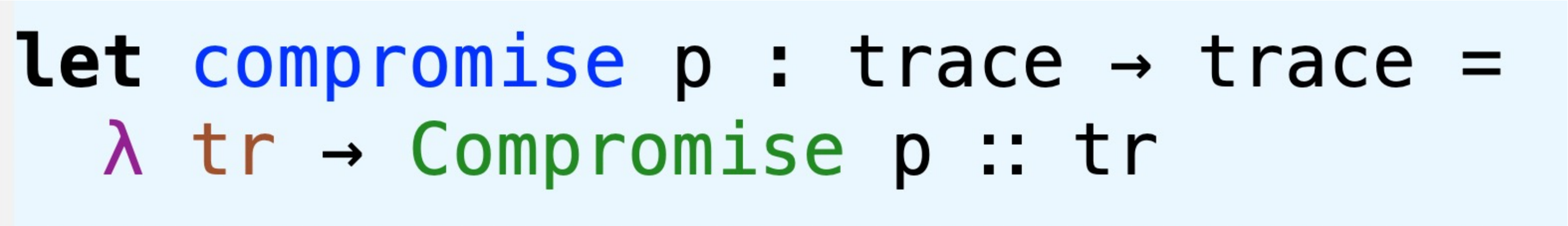
A global protocol trace

```
type principal = string

noeq type entry =
     | FreshGen: p:principal → entry
     | Send: from:principal → to:principal → msg:bytes → entry
     | Store: at:principal → state:bytes → entry
     | Event: p:principal → ev:bytes → entry
     | Compromise: p:principal → entry

type trace = list entry
```

Executing Protocol Actions (1)

```
let recv p : trace → option bytes =
  let rec recv_aux p tr : option bytes =
    match tr with
    | [] → None
    | Send from to msg::tr' → if to = p then Some msg
                              else recv_aux p tr'
    | _ :: tr' → recv_aux p tr'
  in
  recv_aux p

let retrieve p : trace → option bytes =
```

Executing Attacker Actions



```
let compromise p : trace → trace =
  λ tr → Compromise p :: tr
```

- Attacker can call **compromise p** to gain control of **p**
- Attacker can call **gen p** (for compromised **p**) to get fresh bytes
- Attacker can call **recv p** (to read any message)
- Attacker can call **retrieve p** (for compromised **p**) to read its state
- Attacker can call **send p1 p2 m** (for any m it *knows*)
- Attacker cannot call **trigger** or **store**

Attacker Knowledge

```
val attacker_knows: trace → bytes → Type0
```

- Attacker always knows **Constant s**
- Attacker learns **msg** from each **Send from to msg** in trace
- Attacker learns **st** from each **Store p st** (for compromised **p**)
- Attacker can call any crypto function with values it already knows: **concat, split, ae_enc, ae_dec, pk_enc, pk_dec, sign, hash**, …

```
type bytes =
  | Constant: string → bytes
  | Fresh: n:ℕ → bytes
  | Concat: bytes → bytes → bytes
  | AEnc: k:bytes → r:bytes → p:bytes → bytes
  | PK: bytes → bytes
  | PEnc: bytes → bytes → bytes
  | VK: bytes → bytes
  | Sig: bytes → bytes → bytes
```

```
type entry =
| FreshGen: p:principal → entry
| Send: from:principal → to:principal → msg:bytes → entry
| Store: at:principal → state:bytes → entry
| Event: p:principal → ev:bytes → entry
| Compromise: p:principal → entry
```

Reachable Traces

```
(* Some Protocol Code *)
val sendMsg1: principal → principal → trace → trace
val recvMsg1: principal → trace → trace
```

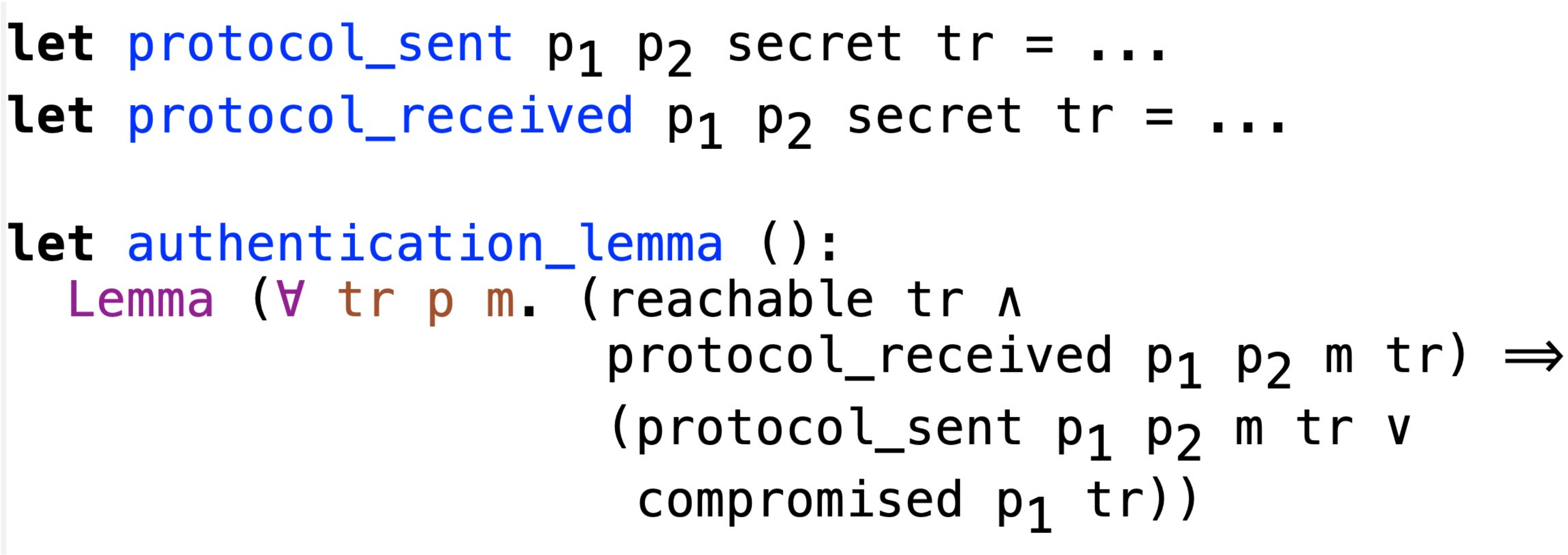
```
(* Reachability *)
let rec reachable (tr:trace) : Type =
 (∃ p1 p2 tr'. tr == sendMsg1 p1 p2 tr') /\ reachable tr') \/
 (∃ p tr'. tr == recvMsg1 p tr') /\ reachable tr') \/
 (match tr with
  | [] → ⊤
  | FreshGen p::tr' → List.mem (Compromise p) tr' ∧ reachable tr'
  | Send p1 p2 m::tr' → attacker_knows tr' m ∧ reachable tr'
  | Compromise p::tr' → reachable tr'
  | _ → ⊥)
```

Stating Secrecy Goals

```
let protocol_sent p secret tr =
  List.mem (Event p (concat (literal "Send") secret)) tr

let compromised p tr =
  List.mem (Compromise p) tr

let secrecy_lemma ():
  Lemma (∀ tr p m. (reachable tr ∧
                    protocol_sent p m tr ∧
                    attacker_knows tr m) ⟹
                    compromised p tr) =
```

- Proof by induction on the length of the trace
- Case analysis on all reachable traces
- Reason about possible attacker actions
- Reason about possible protocol actions

Stating Authentication Goals

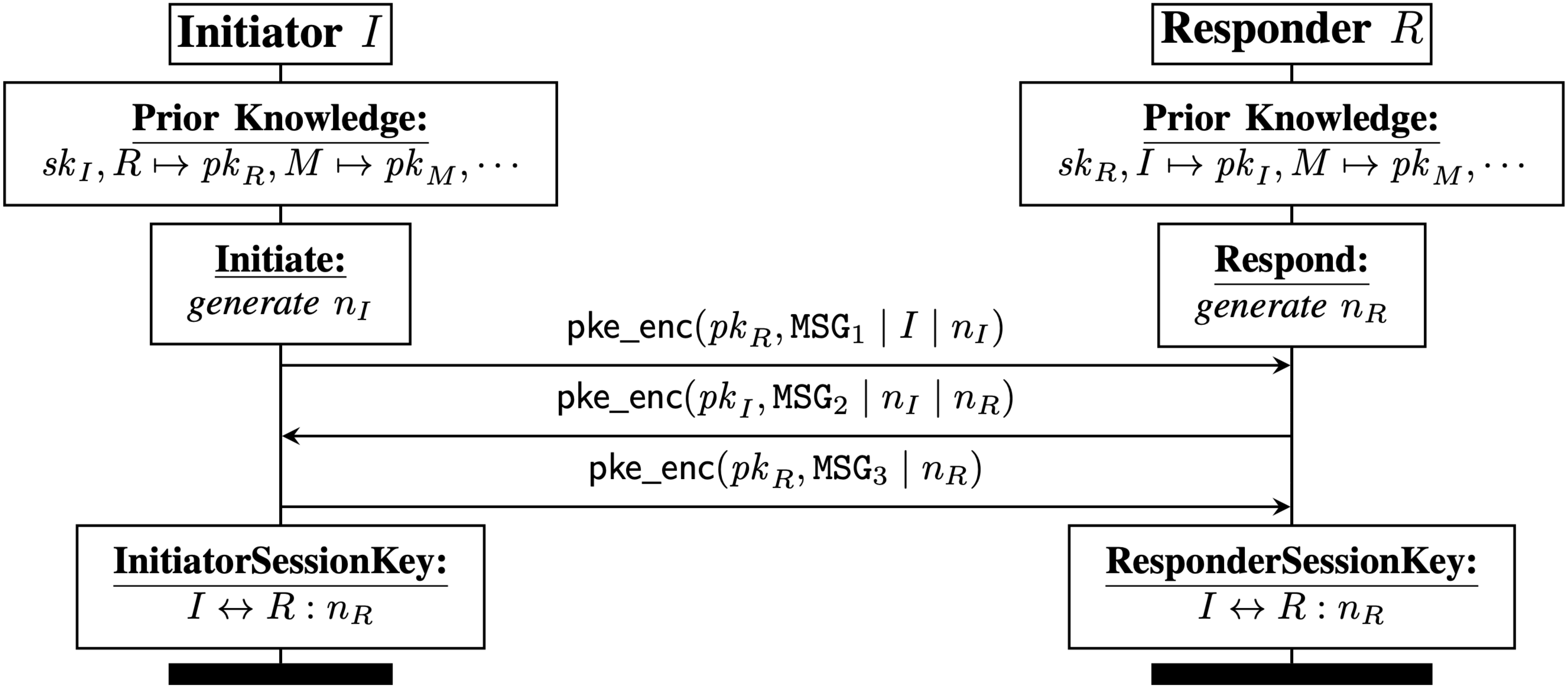


```
let protocol_sent p1 p2 secret tr = ...
let protocol_received p1 p2 secret tr = ...

let authentication_lemma ():
  Lemma (∀ tr p m. (reachable tr ∧
                    protocol_received p1 p2 m tr) ⇒
                   (protocol_sent p1 p2 m tr ∨
                    compromised p1 tr))
```

- **Correspondence Assertion:** Received p1 p2 m => Sent p1 p2 m
- Proof by induction on all reachable traces

Modular Labeled Proofs for Crypto Protocols in DY*

Needham-Schroeder Public-Key Protocol



Initiator I

Prior Knowledge:
$sk_I, R \mapsto pk_R, M \mapsto pk_M, \cdots$

Initiate:
generate n_I

Responder R

Prior Knowledge:
$sk_R, I \mapsto pk_I, M \mapsto pk_M, \cdots$

Respond:
generate n_R

pke_enc$(pk_R, \texttt{MSG}_1 \mid I \mid n_I)$

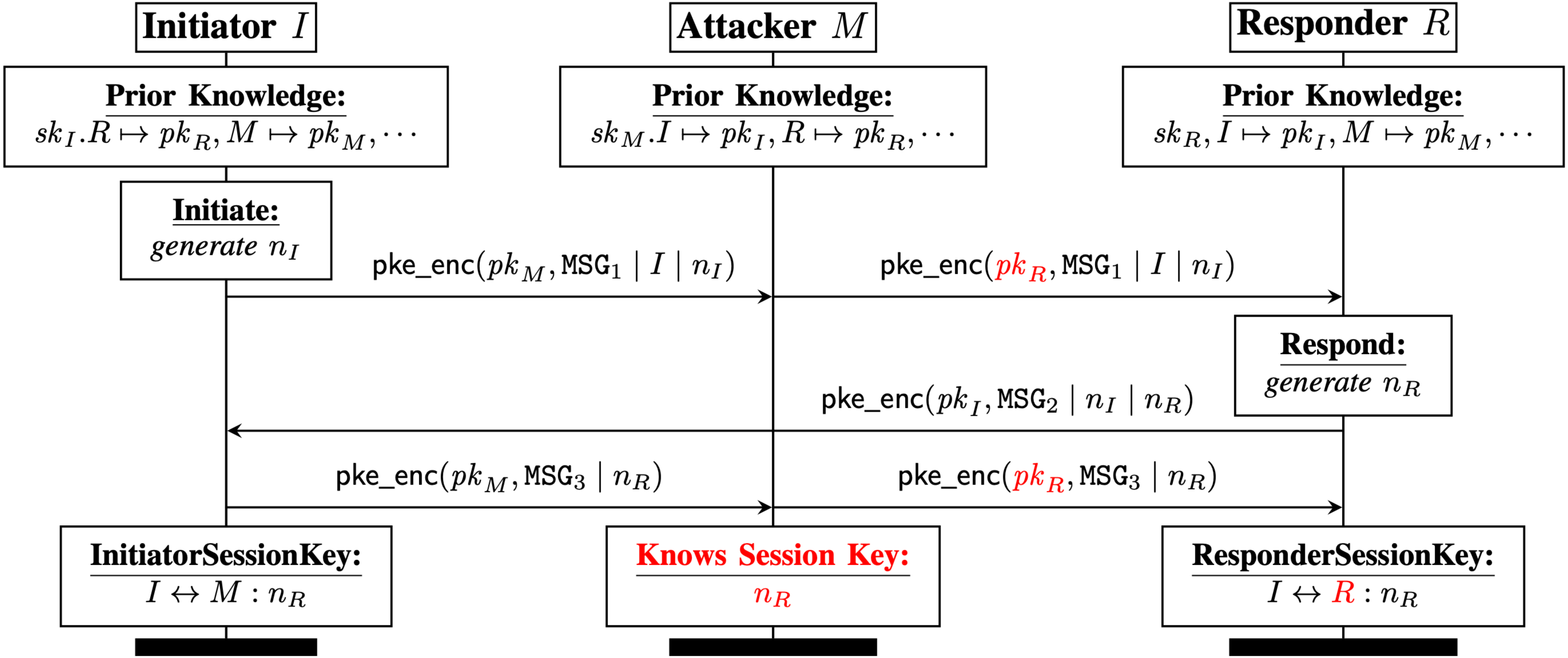
pke_enc$(pk_I, \texttt{MSG}_2 \mid n_I \mid n_R)$

pke_enc$(pk_R, \texttt{MSG}_3 \mid n_R)$

InitiatorSessionKey:
$I \leftrightarrow R : n_R$

ResponderSessionKey:
$I \leftrightarrow R : n_R$

Lowe's Attack on NS-PK



Initiator I

Attacker M

Responder R

Prior Knowledge:
$sk_I . R \mapsto pk_R, M \mapsto pk_M, \cdots$

Prior Knowledge:
$sk_M . I \mapsto pk_I, R \mapsto pk_R, \cdots$

Prior Knowledge:
$sk_R, I \mapsto pk_I, M \mapsto pk_M, \cdots$

Initiate:
generate n_I

$\text{pke\_enc}(pk_M, \texttt{MSG}_1 \mid I \mid n_I)$

$\text{pke\_enc}(pk_R, \texttt{MSG}_1 \mid I \mid n_I)$

Respond:
generate n_R

$\text{pke\_enc}(pk_I, \texttt{MSG}_2 \mid n_I \mid n_R)$

$\text{pke\_enc}(pk_M, \texttt{MSG}_3 \mid n_R)$

$\text{pke\_enc}(pk_R, \texttt{MSG}_3 \mid n_R)$

InitiatorSessionKey:
$I \leftrightarrow M : n_R$

Knows Session Key:
n_R

ResponderSessionKey:
$I \leftrightarrow R : n_R$

NS-PK in F*: Messages

```
type message =
| Msg1: i:principal → n_i: bytes → message
| Msg2: n_i: bytes → n_r:bytes → message
| Msg3: n_r: bytes → message
val serialize_message: message → bytes
val parse_message: bytes → result message
val parse_message_correctness_lemma: m:message →
  Lemma (parse_message (serialize_message m) == Success m)
```

Precise Message Formats

- serialization and parsing with correctness proofs

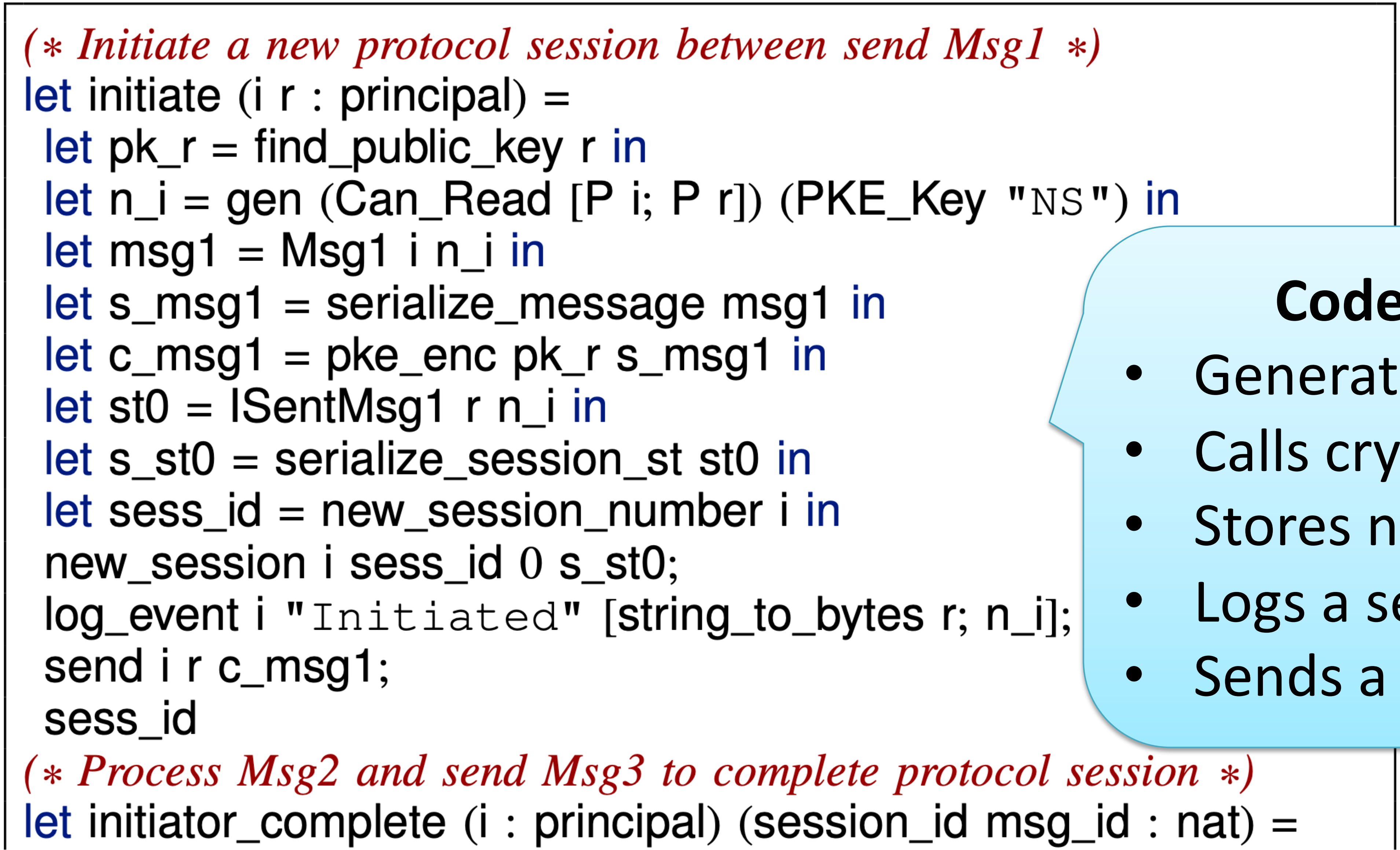
NS-PK in F*: Session State

```
type session_st =
| SecretKey: secret_key: bytes → session_st
| PublicKey: peer:principal → public_key:bytes → session_st
| ISentMsg1: r:principal → n_i:bytes → session_st
| RSentMsg2: i:principal → n_i:bytes → n_r:bytes → session_st
| ISentMsg3: r:principal → n_i:bytes → n_r:bytes → session_st
| RReceivedMsg3: i:principal → n_r:bytes → session_st
val serialize_session_st: session_st → bytes
val parse_session_st: bytes → result session_st
```

Protocol State Machine

- Stateful protocol code
- Session state storage
- Fine-grained compromise

NS-PK in F*: Protocol Code



```
(* Initiate a new protocol session between send Msg1 *)
let initiate (i r : principal) =
 let pk_r = find_public_key r in
 let n_i = gen (Can_Read [P i; P r]) (PKE_Key "NS") in
 let msg1 = Msg1 i n_i in
 let s_msg1 = serialize_message msg1 in
 let c_msg1 = pke_enc pk_r s_msg1 in
 let st0 = ISentMsg1 r n_i in
 let s_st0 = serialize_session_st st0 in
 let sess_id = new_session_number i in
 new_session i sess_id 0 s_st0;
 log_event i "Initiated" [string_to_bytes r; n_i];
 send i r c_msg1;
 sess_id
(* Process Msg2 and send Msg3 to complete protocol session *)
let initiator_complete (i : principal) (session_id msg_id : nat) =
```

Code for Initiator

- Generates a nonce
- Calls crypto functions
- Stores new session state
- Logs a security ecent
- Sends a message

How do we show this NS-PK implementation is secure?

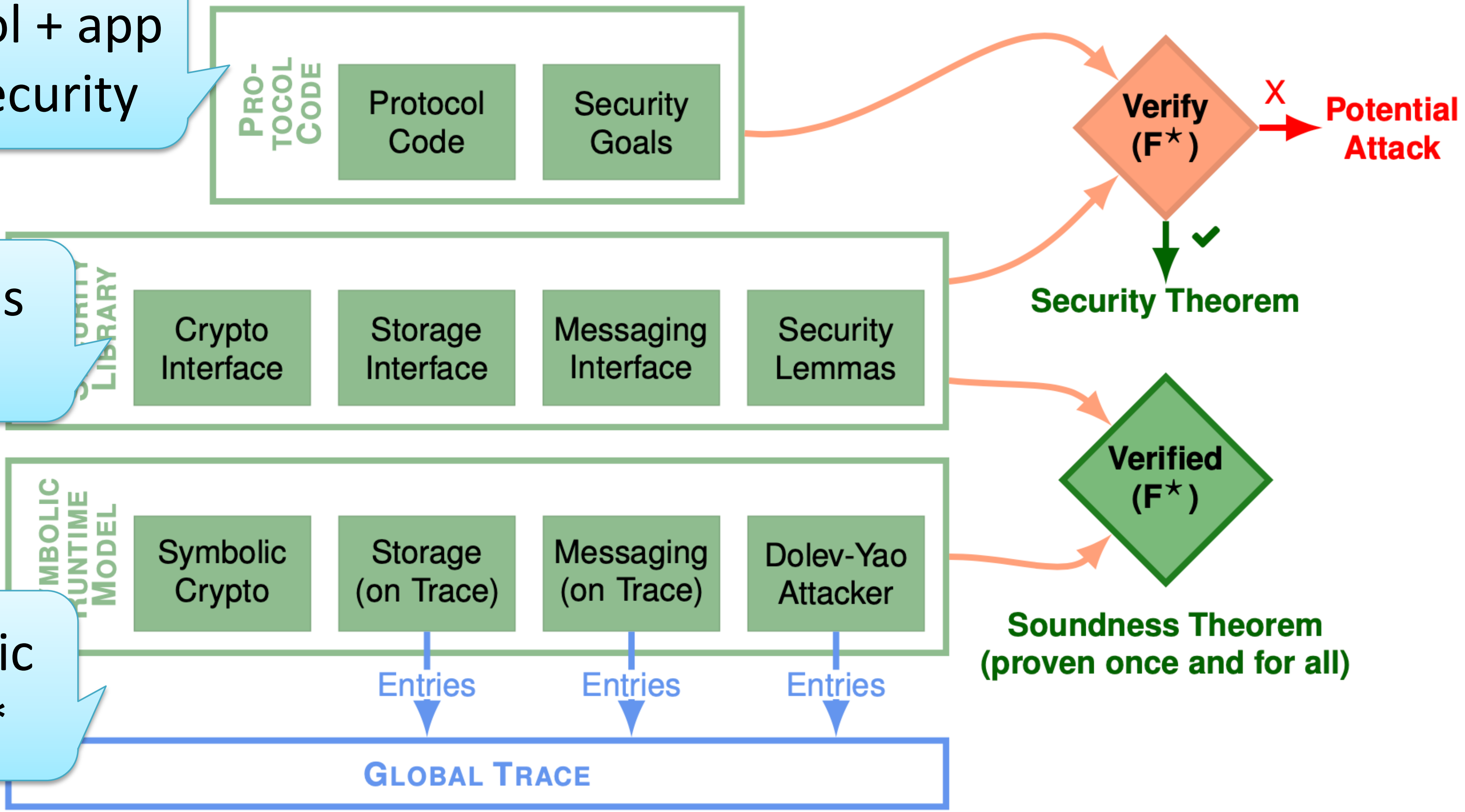
DY* Verification Architecture

[Euro S&P 2021]

Executable protocol + app code verified for security

Abstract labeled APIs proved sound in F*

Trace-based symbolic runtime model in F*



PRO-TOCOL CODE
Protocol Code
Security Goals

URITY LIBRARY
Crypto Interface
Storage Interface
Messaging Interface
Security Lemmas

MBOLIC UNTIME MODEL
Symbolic Crypto
Storage (on Trace)
Messaging (on Trace)
Dolev-Yao Attacker

Entries
Entries
Entries

GLOBAL TRACE

Verify (F★)

X

Potential Attack

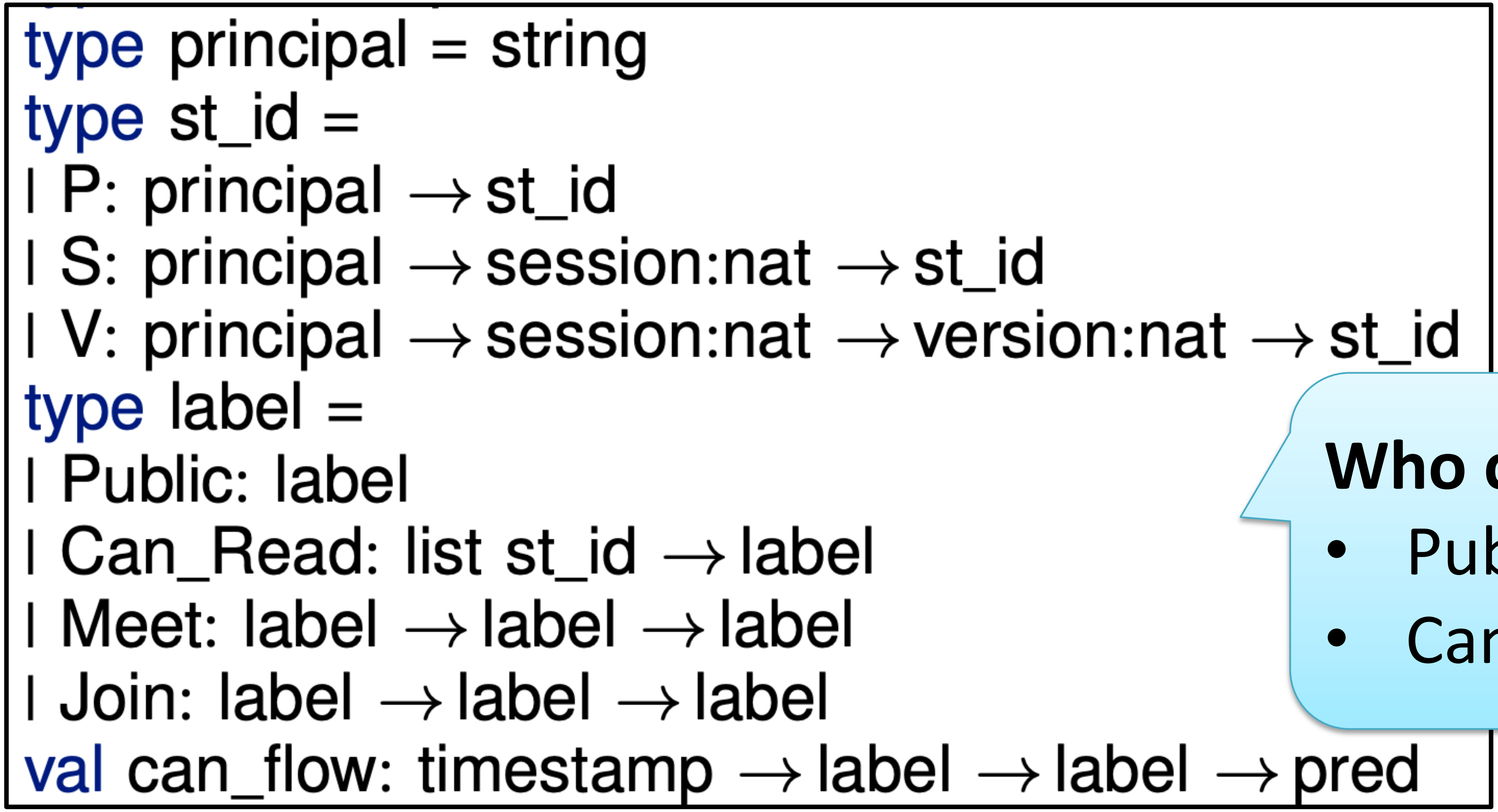
✓

Security Theorem

Verified (F★)

Soundness Theorem (proven once and for all)

Secrecy Labels for Bytstrings

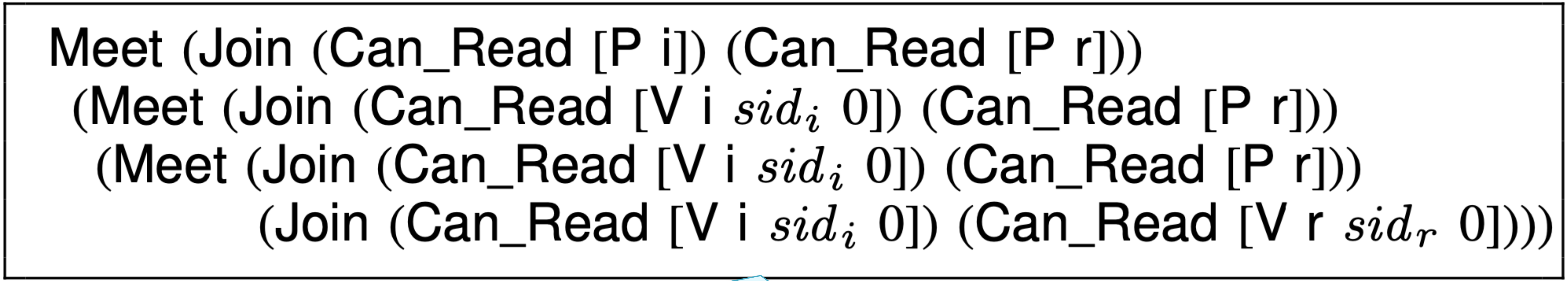


```
type principal = string
type st_id =
| P: principal → st_id
| S: principal → session:nat → st_id
| V: principal → session:nat → version:nat → st_id
type label =
| Public: label
| Can_Read: list st_id → label
| Meet: label → label → label
| Join: label → label → label
val can_flow: timestamp → label → label → pred
```

Who can read a secret?

- Public: anybody
- CanRead [P a; P b]: a or b

Secrecy Labels for Bytstrings



Meet (Join (Can_Read [P i]) (Can_Read [P r]))
(Meet (Join (Can_Read [V i sid_i 0]) (Can_Read [P r]))
(Meet (Join (Can_Read [V i sid_i 0]) (Can_Read [P r]))
(Join (Can_Read [V i sid_i 0]) (Can_Read [V r sid_r 0]))))

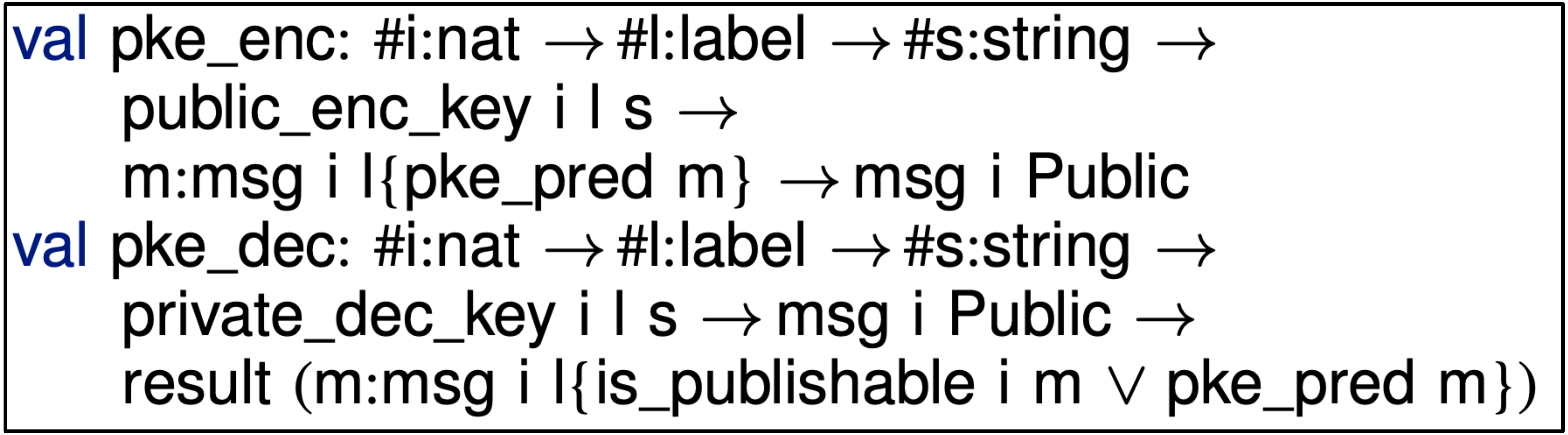
Label for session key in Signal Protocol

- Encodes channel secrecy
- Forward and Post-Compromise security

A Labeled Crypto API

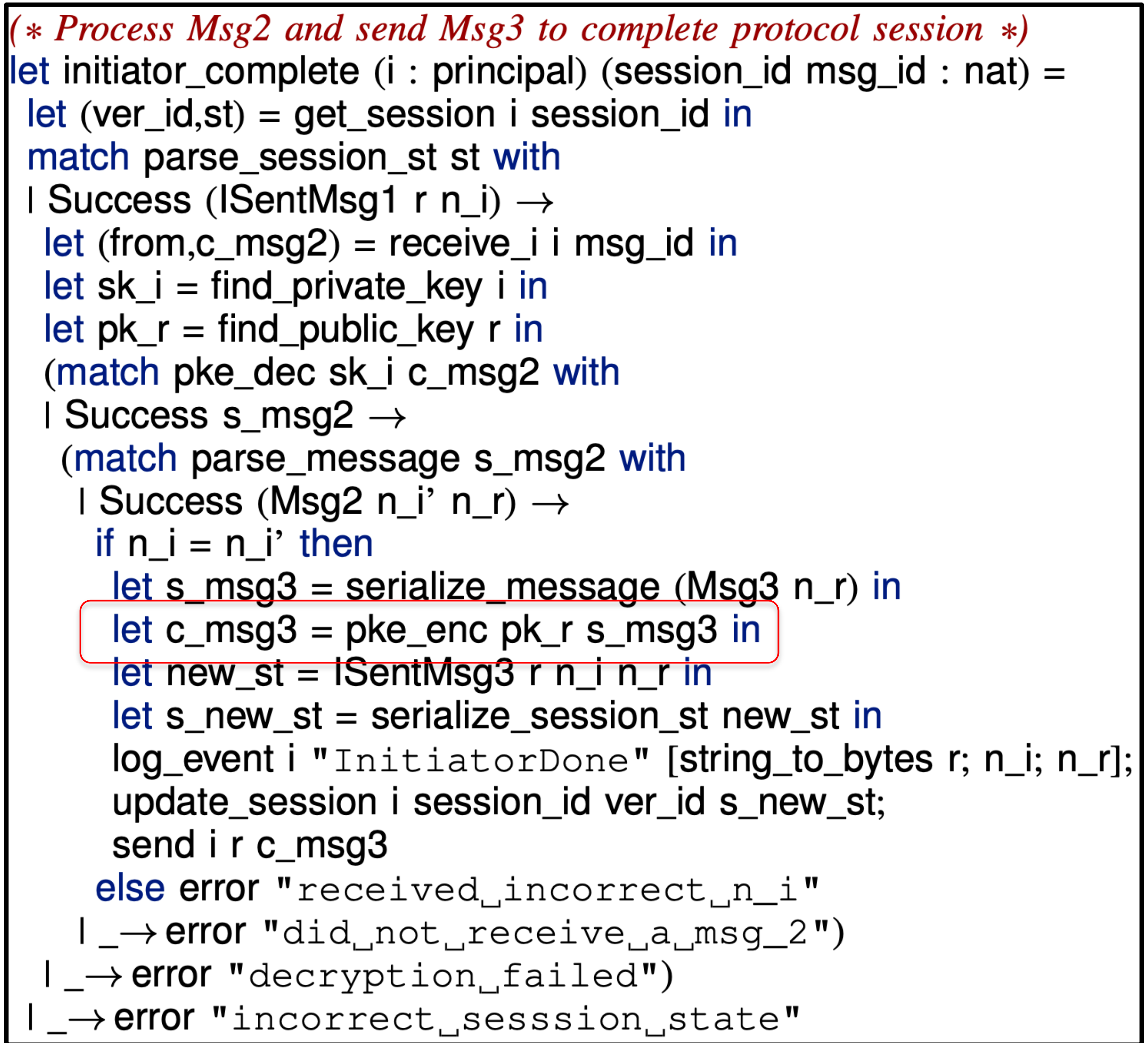
Typed Cryptographic API encodes security assumptions

Using secrecy labels and authentication predicates



```
val pke_enc: #i:nat → #l:label → #s:string →
    public_enc_key i l s →
    m:msg i l{pke_pred m} → msg i Public
val pke_dec: #i:nat → #l:label → #s:string →
    private_dec_key i l s → msg i Public →
    result (m:msg i l{is_publishable i m ∨ pke_pred m})
```

Lowe’s Attack as a Type Error



```
(* Process Msg2 and send Msg3 to complete protocol session *)
let initiator_complete (i : principal) (session_id msg_id : nat) =
 let (ver_id,st) = get_session i session_id in
 match parse_session_st st with
 | Success (ISentMsg1 r n_i) →
  let (from,c_msg2) = receive_i i msg_id in
  let sk_i = find_private_key i in
  let pk_r = find_public_key r in
  (match pke_dec sk_i c_msg2 with
  | Success s_msg2 →
   (match parse_message s_msg2 with
    | Success (Msg2 n_i’ n_r) →
     if n_i = n_i’ then
      let s_msg3 = serialize_message (Msg3 n_r) in
      let c_msg3 = pke_enc pk_r s_msg3 in
      let new_st = ISentMsg3 r n_i n_r in
      let s_new_st = serialize_session_st new_st in
      log_event i "InitiatorDone" [string_to_bytes r; n_i; n_r];
      update_session i session_id ver_id s_new_st;
      send i r c_msg3
     else error "received␣incorrect␣n_i"
    | _→ error "did␣not␣receive␣a␣msg_2")
  | _→ error "decryption␣failed")
 | _→ error "incorrect␣sesssion␣state"
```

Can n_r be sent to r?

- Does the label of n_r flow to CanRead [P r]?
- Not provable, because Lowe’s attack
- Indeed, we can implement and demonstrate symbolic attack in F*

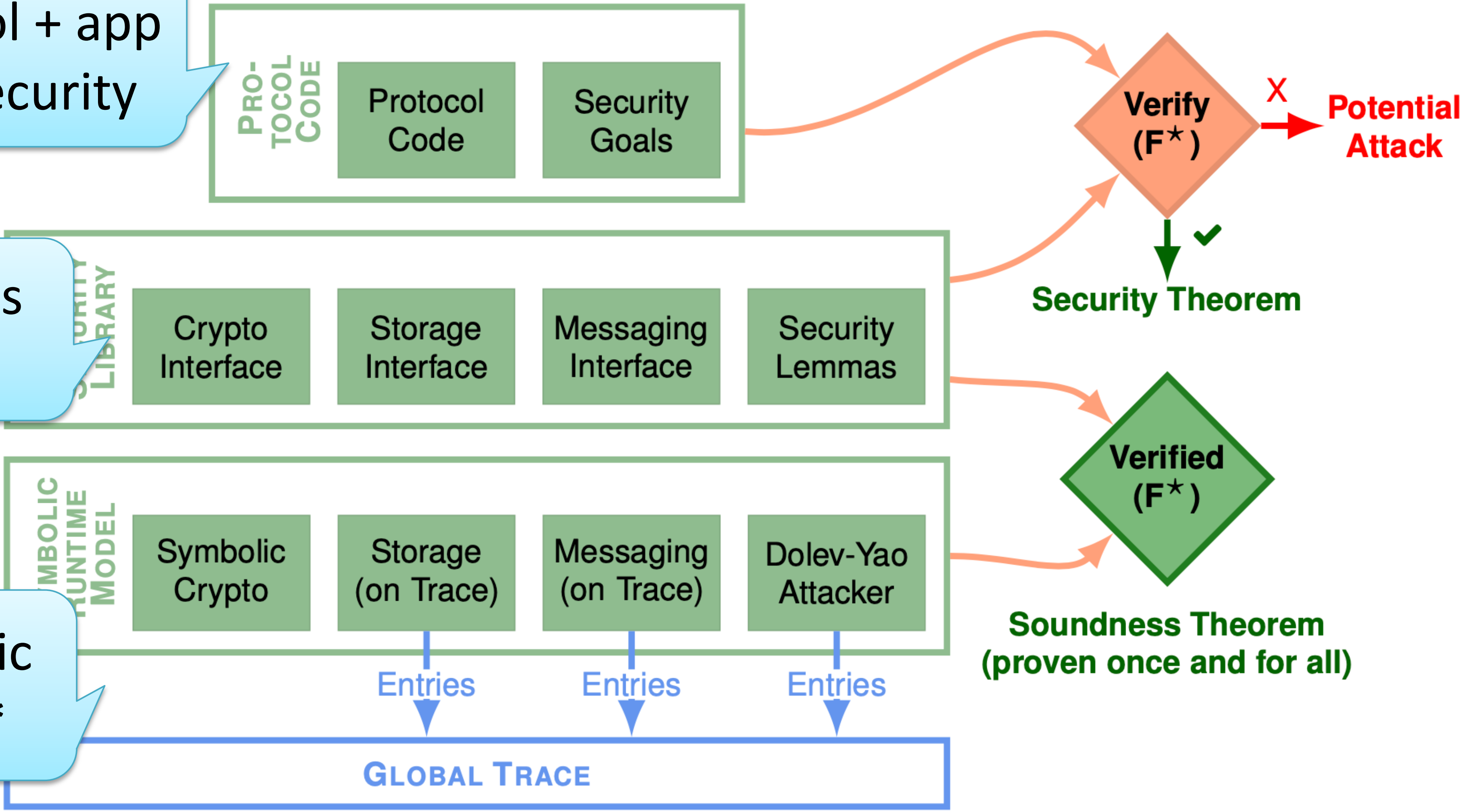
DY* Verification Architecture

[Euro S&P 2021]

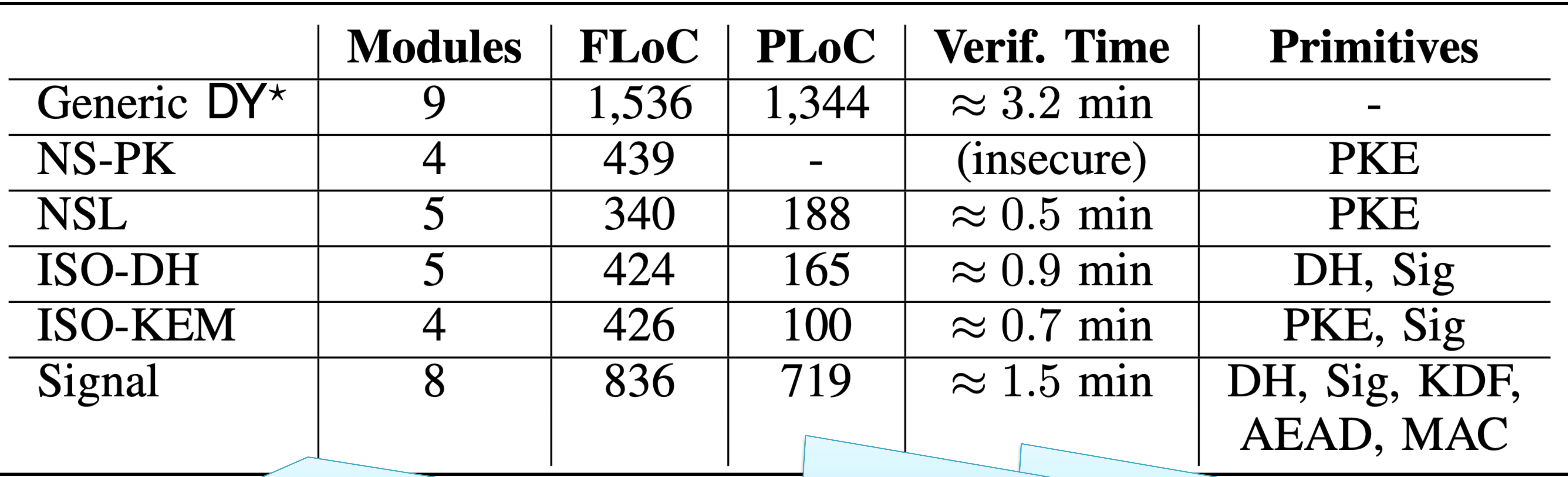
Executable protocol + app code verified for security

Abstract labeled APIs proved sound in F*

Trace-based symbolic runtime model in F*



DY*: scalable security verification



	Modules	FLoC	PLoC	Verif. Time	Primitives
Generic DY*	9	1,536	1,344	≈ 3.2 min	-
NS-PK	4	439	-	(insecure)	PKE
NSL	5	340	188	≈ 0.5 min	PKE
ISO-DH	5	424	165	≈ 0.9 min	DH, Sig
ISO-KEM	4	426	100	≈ 0.7 min	PKE, Sig
Signal	8	836	719	≈ 1.5 min	DH, Sig, KDF, AEAD, MAC

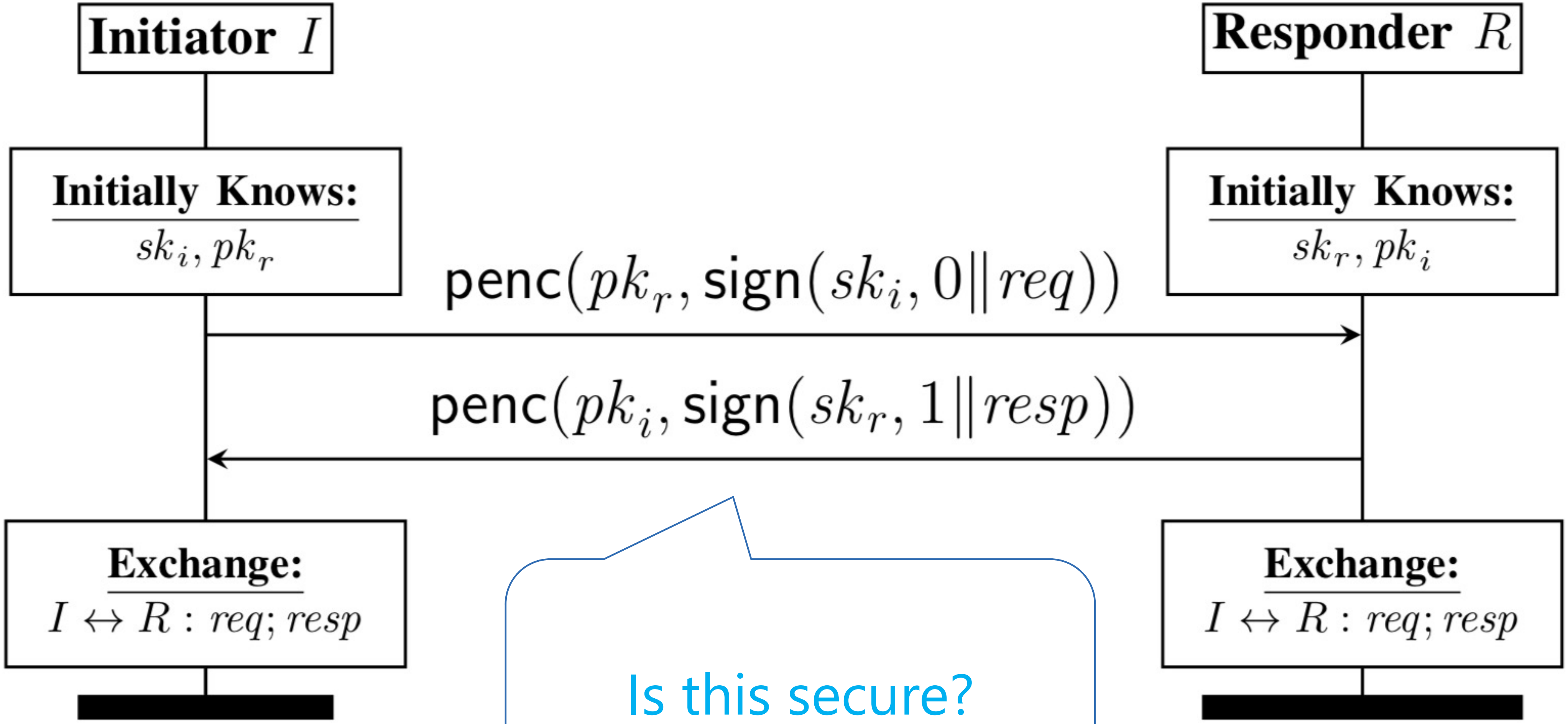
Proofs require between 50% and 90% annotation overhead

Size of annotation depends on complexity of security goals

ize

amarin

Sign-then-Encrypt Protocol



Initiator I

Initially Knows:
sk_i, pk_r

Responder R

Initially Knows:
sk_r, pk_i

$\mathsf{penc}(pk_r, \mathsf{sign}(sk_i, 0\|req))$

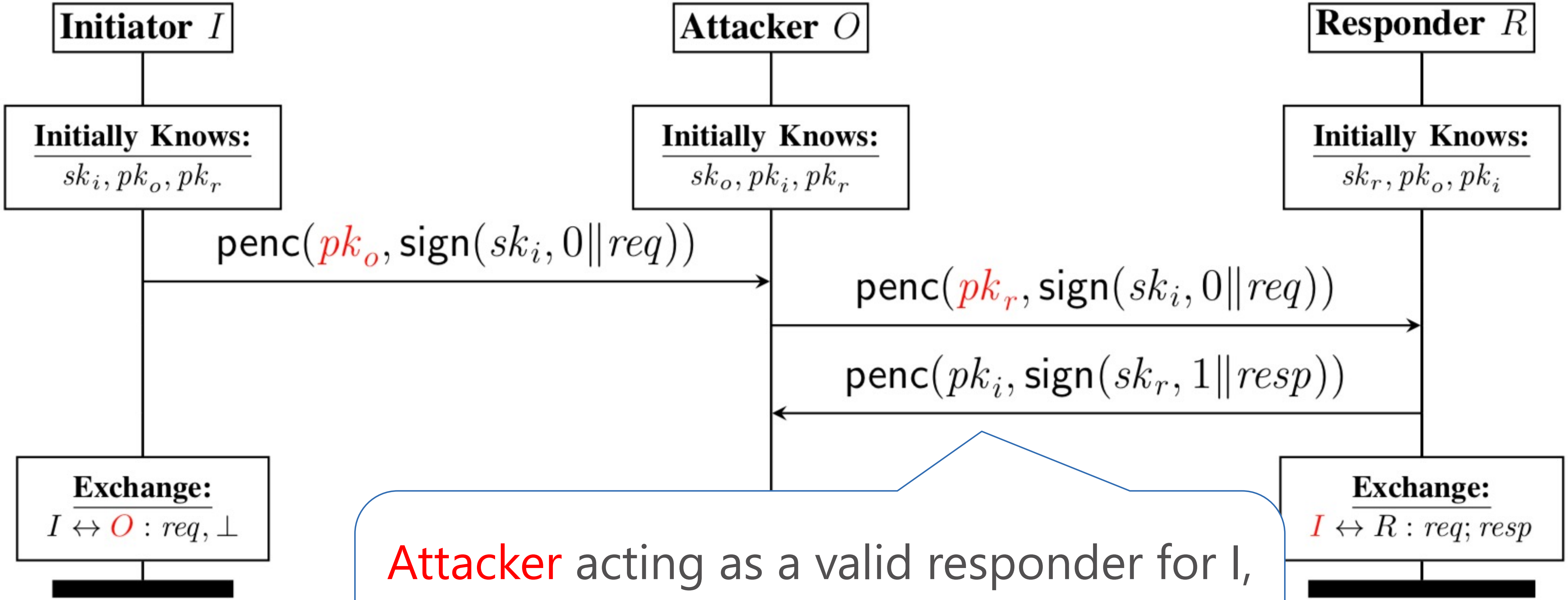
$\mathsf{penc}(pk_i, \mathsf{sign}(sk_r, 1\|resp))$

Exchange:
$I \leftrightarrow R : req; resp$

Exchange:
$I \leftrightarrow R : req; resp$

Is this secure?

Man-in-the-Middle Attack



Initiator I

Attacker O

Responder R

Initially Knows:
sk_i, pk_o, pk_r

Initially Knows:
sk_o, pk_i, pk_r

Initially Knows:
sk_r, pk_o, pk_i

$\mathsf{penc}(pk_o, \mathsf{sign}(sk_i, 0 \| req))$

$\mathsf{penc}(pk_r, \mathsf{sign}(sk_i, 0 \| req))$

$\mathsf{penc}(pk_i, \mathsf{sign}(sk_r, 1 \| resp))$

Exchange:
$I \leftrightarrow O : req, \perp$

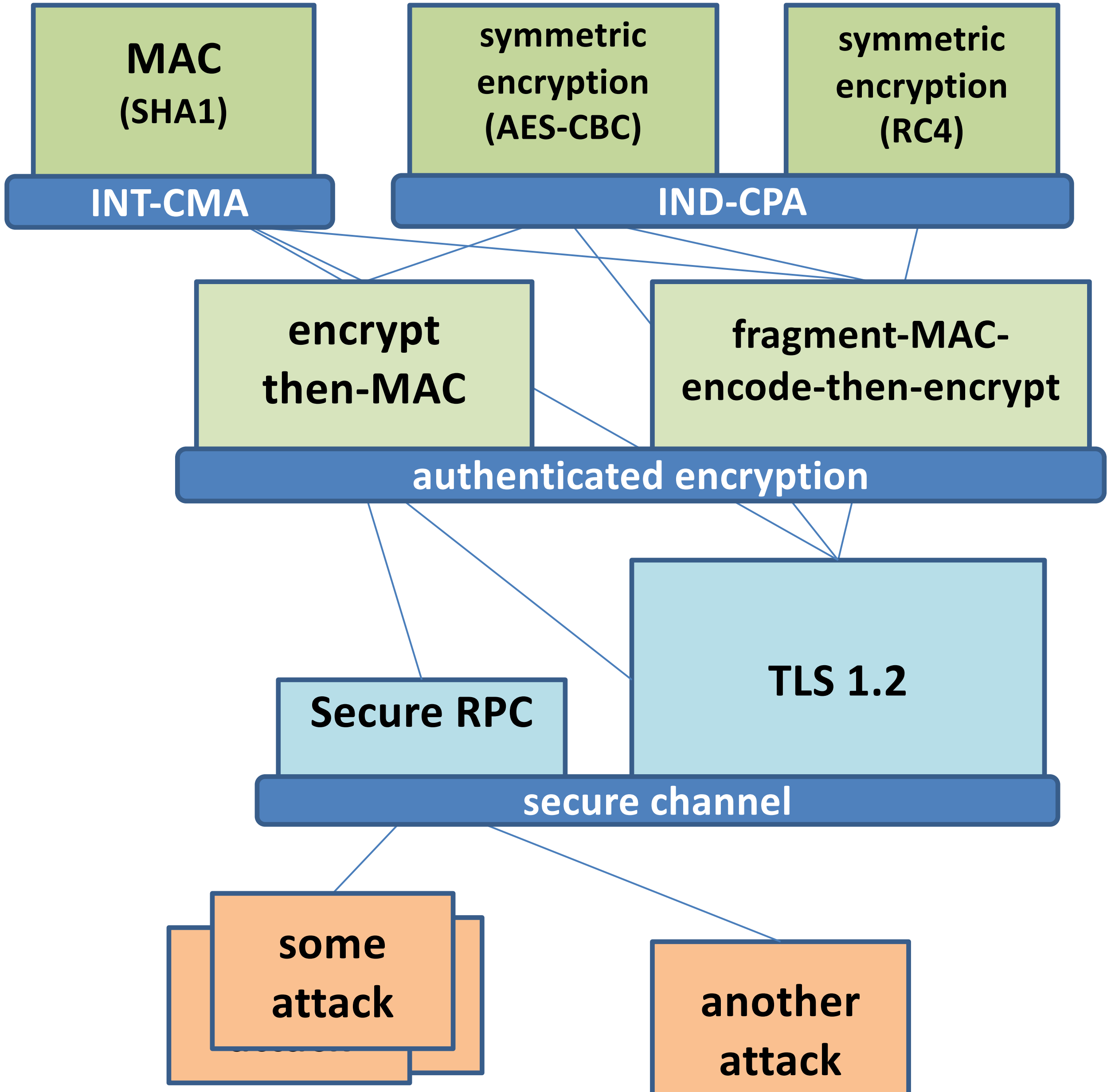
Exchange:
$I \leftrightarrow R : req; resp$

Attacker acting as a valid responder for I,
re-encrypts request to R,
causing an *identity mis-binding attack*

Implementing Sign-Then-Encrypt (demo)

Modeling Computational Assumptions

Modular Type-Based Cryptographic Verification



MAC (SHA1)

symmetric encryption (AES-CBC)

symmetric encryption (RC4)

INT-CMA

IND-CPA

encrypt then-MAC

fragment-MAC-encode-then-encrypt

authenticated encryption

Secure RPC

TLS 1.2

secure channel

some attack

another attack

cryptographic algorithms

typed interfaces: **cryptographic assumptions**

cryptographic constructions

typed interfaces: **security guarantees**

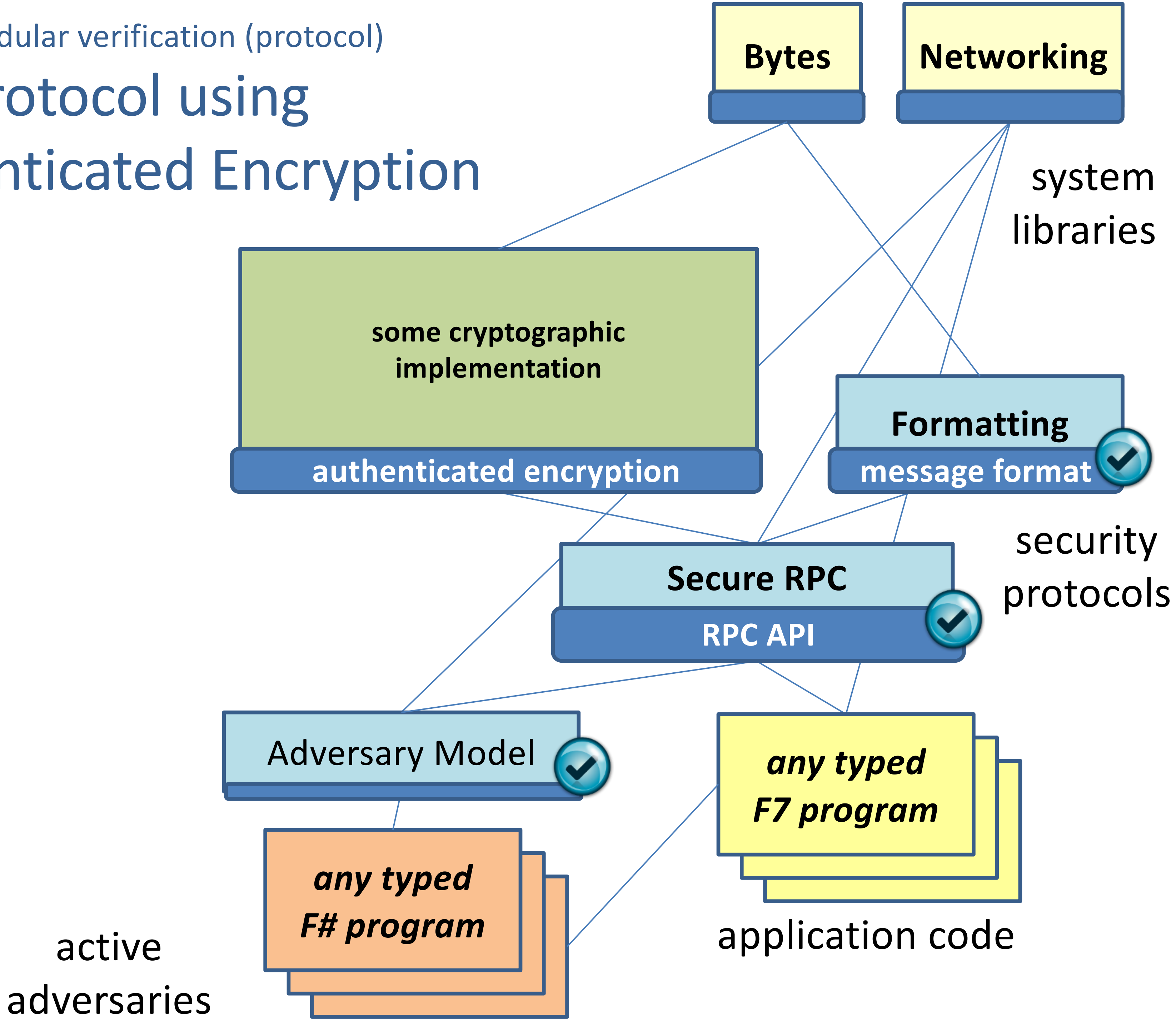
security protocols

typed interfaces: **attacker models**

active adversaries

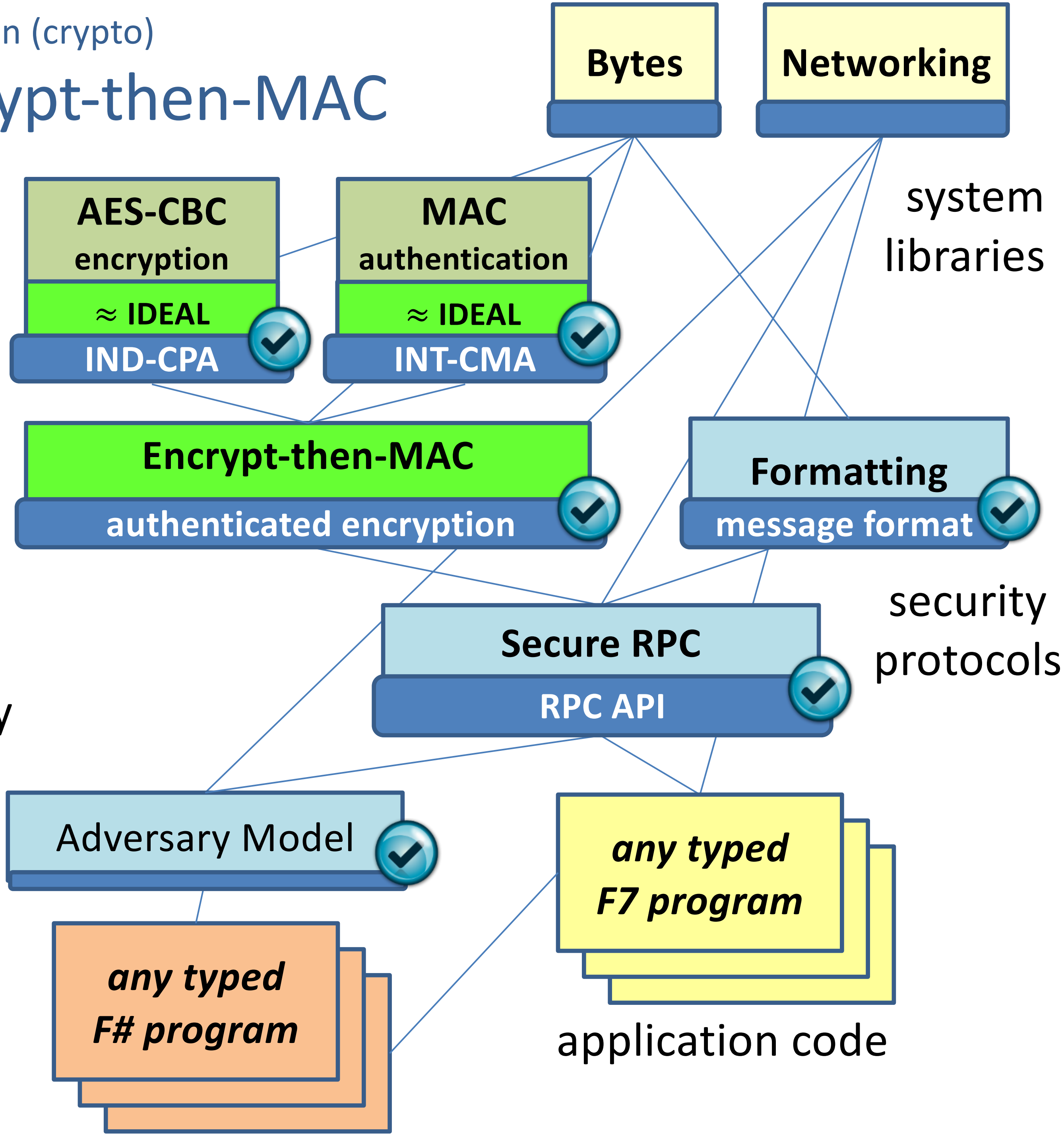
Sample modular verification (protocol)

RPC protocol using Authenticated Encryption



Sample modular verification (crypto)

RPC using Encrypt-then-MAC



Bytes

Networking

system libraries

cryptographic schemes

AES-CBC encryption
≈ IDEAL ✓
IND-CPA

MAC authentication
≈ IDEAL ✓
INT-CMA

cryptographic constructions

Encrypt-then-MAC ✓
authenticated encryption

Formatting ✓
message format

probabilistic computational indistinguishability

Secure RPC ✓
RPC API

security protocols

Adversary Model ✓

any typed F# program

active adversaries

any typed F7 program

application code

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 F*
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

ideal F* interface

MAC keys are abstract

```
module MAC

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

predicate Msg of key * text

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"

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}
predicate Msg of key * text
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)

ideal F* interface

```
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)
```

concrete F* implementation (using real crypto)

Sample computational assumption:

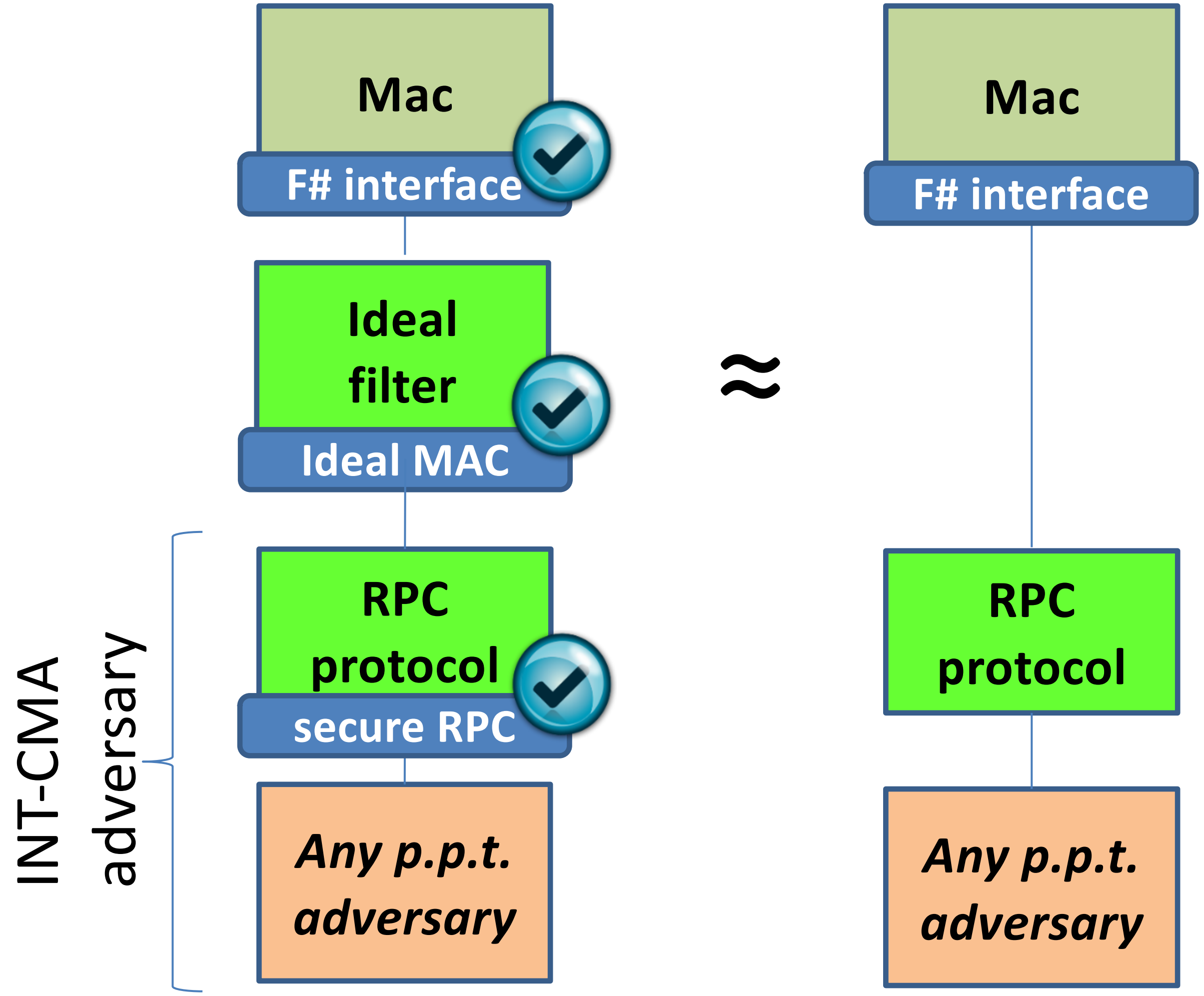
Resistance to Chosen-Message Existential Forgery Attacks (INT-CMA)

```
module INT_CMA_Game
open Mac
Let private k = GEN()
let private log = ref []
let mac t =
  log := t::!log
  MAC k t
let verify t m =
  let v = VERIFY k t m in
  if v && not (mem t !log) then FORGERY
  v
```

Computational Safety
a probabilistic polytime program calling **mac** and **verify** forges a MAC only with negligible probability ^{2}

CMA game
(coded in F#)

Computational Safety for MACs



ideal system

Mac

F# interface

Ideal filter

Ideal MAC

RPC protocol

secure RPC

Any p.p.t. adversary

INT-CMA adversary

perfectly safe
by typing

≈

concrete system

Mac

F# interface

RPC protocol

Any p.p.t. adversary

safe too,
with probability 1 – 1/**ε**

concrete algorithm
assumed INT-CMA computationally

error correction
making VERIFY returns
false on forgeries

sample protocol
typed against
ideal MAC interface

protocol adversary
typed against
RPC interface

Sample ideal functionality:

Supporting Key Compromise

ideal F* interface

MAC keys are abstract

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

val keysize
type keybytes = b:bytes{Length(b)=keysize}
val LEAK:   k:key{!t. Msg(k,t)} -> b:keybytes
val COERCE: b:keybytes{…}       -> k:key{…}
```

MACs are fixed sized

Msg is specified by protocols using MACs

"All verified messages have been MACed"

MAC keys have concrete representations

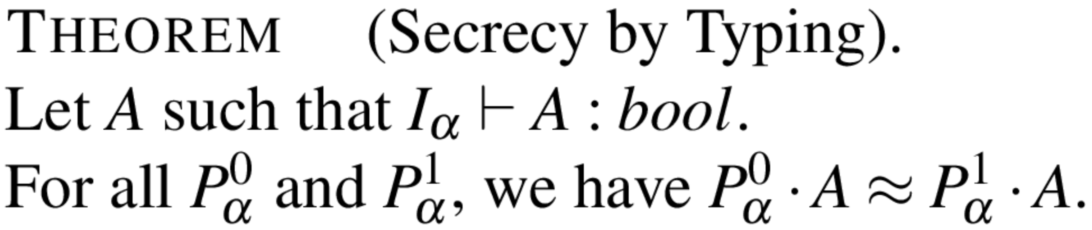
It is safe to turn keys into bytes when **all messages are verifiable**

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 $I_\alpha \vdash A : \mathit{bool}$.

For all P^0_α and P^1_α, we have $P^0_\alpha \cdot A \approx P^1_\alpha \cdot A$.

Plaintext Modules

- Encryption is parameterized by a module that abstractly define 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!



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 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

An Ideal Interface for CCA2-Secure Encryption

```
module PKENC
open Plain

val pksize: int
type skey
type pkey = b:bytes{ PKey(b) Æ}

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

val GEN: unit -> pkey * skey
val ENC: pkey -> plain -> cipher
val DEC: skey -> cipher -> plain
```

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

Typed Secrecy from CCA2-Secure Encryption

THEOREM 7 (Asymptotic Secrecy).
Let P^0 and P^1 p.p.t. secret with $\vdash P^b \rightsquigarrow I_{\mathsf{PLAIN}}$.
Let C_{ENC} p.p.t. CCA2-secure with $I^C_{\mathsf{PLAIN}} \vdash C_{\mathsf{ENC}} \rightsquigarrow I^C_{\mathsf{ENC}}$.
Let A p.p.t. with $I_{\mathsf{PLAIN}}, I_{\mathsf{ENC}} \vdash A : \mathit{bool}$.

$$P^0 \cdot C_{\mathsf{ENC}} \cdot A \approx_\varepsilon P^1 \cdot C_{\mathsf{ENC}} \cdot A.$$

THEOREM 8 (Ideal Functionality).
Let P p.p.t. with $\vdash P \rightsquigarrow I^C_{\mathsf{PLAIN}}$ (not necessarily secret)
Let C_{ENC} p.p.t. CCA2-secure with $I^C_{\mathsf{PLAIN}} \vdash C_{\mathsf{ENC}} \rightsquigarrow I^C_{\mathsf{ENC}}$.
Let A p.p.t. with $I^C_{\mathsf{PLAIN}}, I_{\mathsf{ENC}} \vdash A$.

$$P \cdot C_{\mathsf{ENC}} \cdot A \approx_\varepsilon P \cdot C_{\mathsf{ENC}} \cdot F_{\mathsf{ENC}} \cdot A.$$

Variants: CPA & Authentication

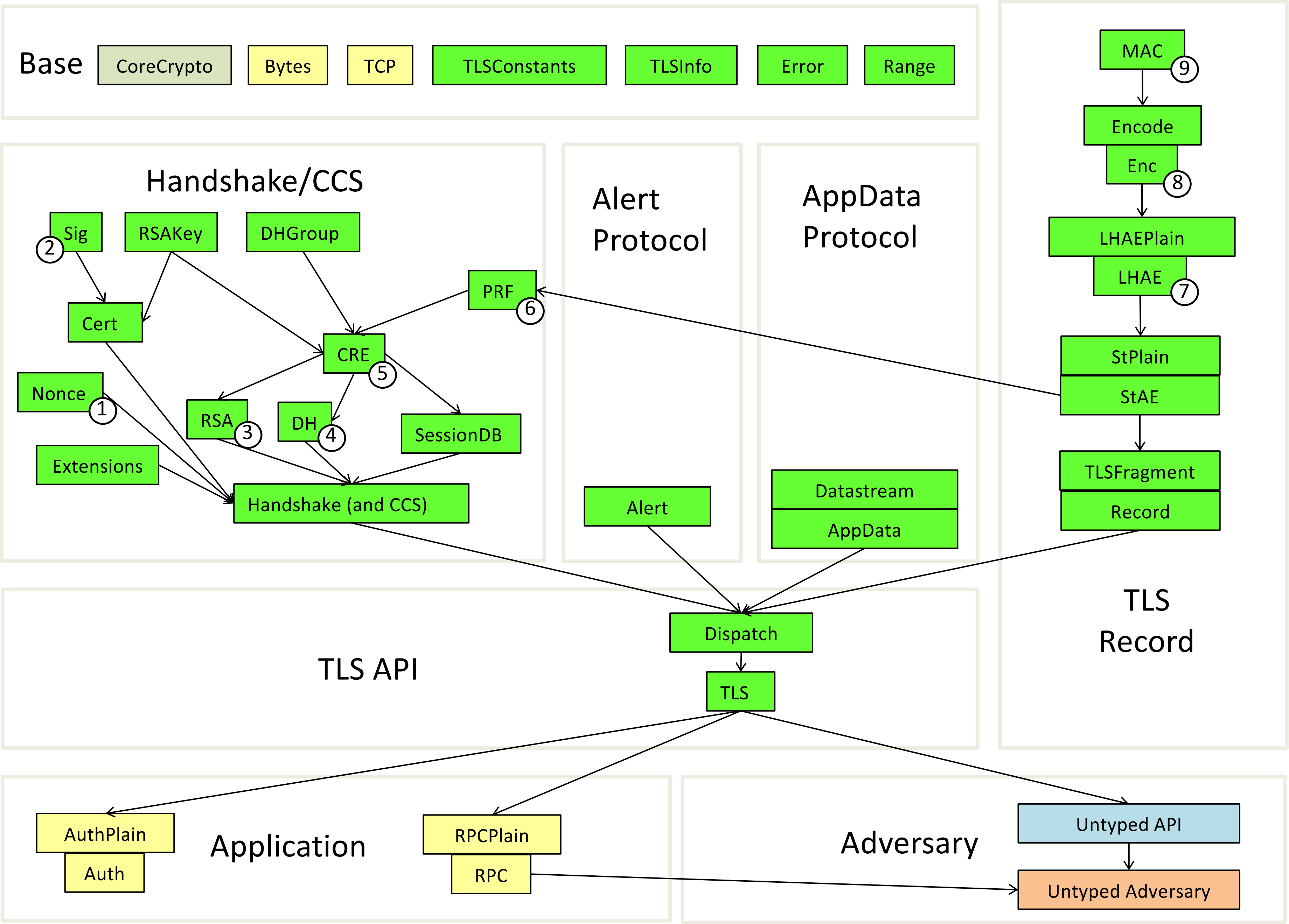
- With **CPA-secure encryption**, we have a **weaker** ideal interface that demands ciphertext integrity before decryption

```
predicate 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)

```
predicate 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
```

Modular Architecture for miTLS



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: TcpStream -> params -> (;Client) nullCn Result
val accept:  TcpStream -> params -> (;Server) nullCn Result

// triggering new handshakes, and closing connections
val rehandshake: c:cn{Role(c)=Client} -> cn Result
val request:     c:cn{Role(c)=Server} -> cn Result
val shutdown:    c:cn -> TcpStream Result

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

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

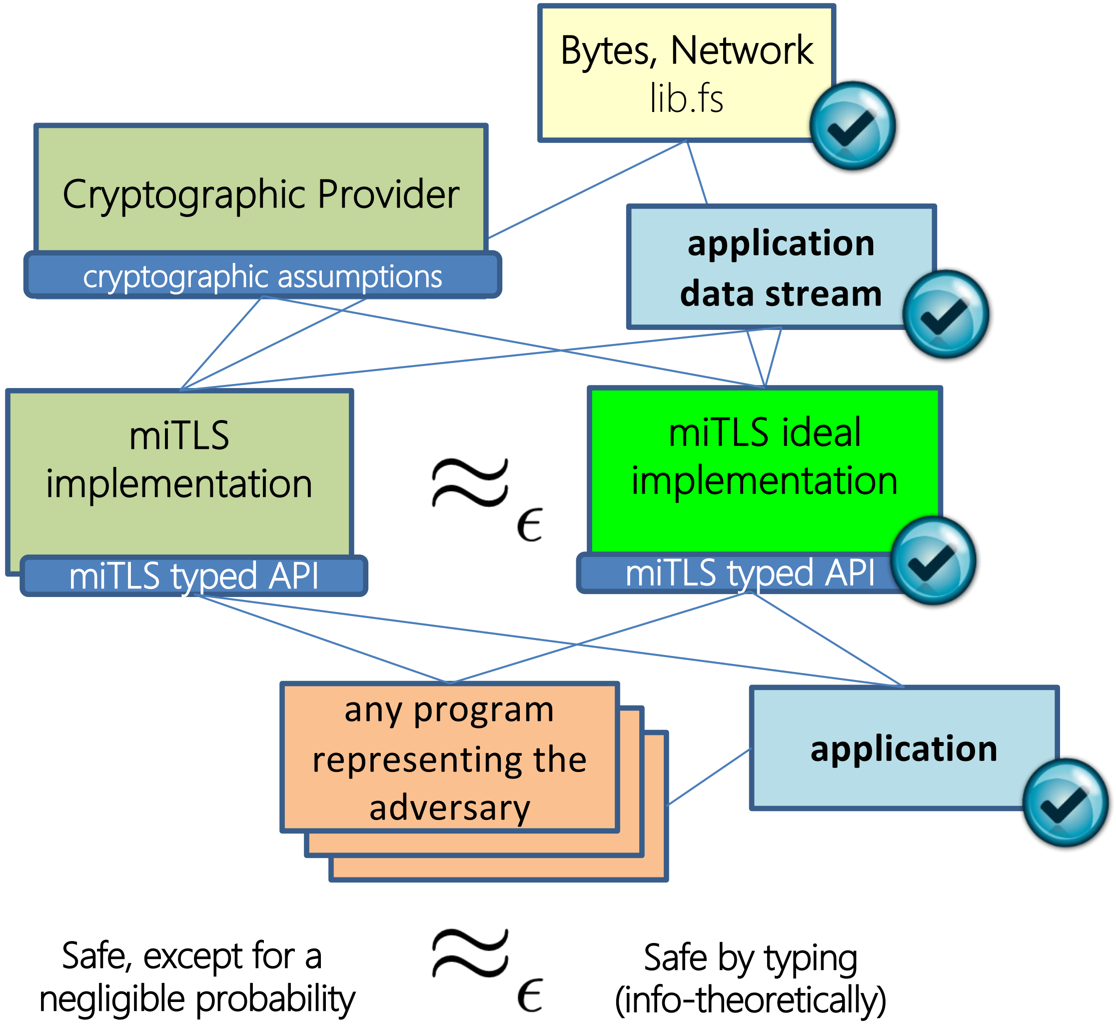
Security theorem

Main crypto result:

concrete TLS & ideal TLS are computationally indistinguishable

We prove that ideal miTLS meets its secure channel specification

using standard program verification (typing)



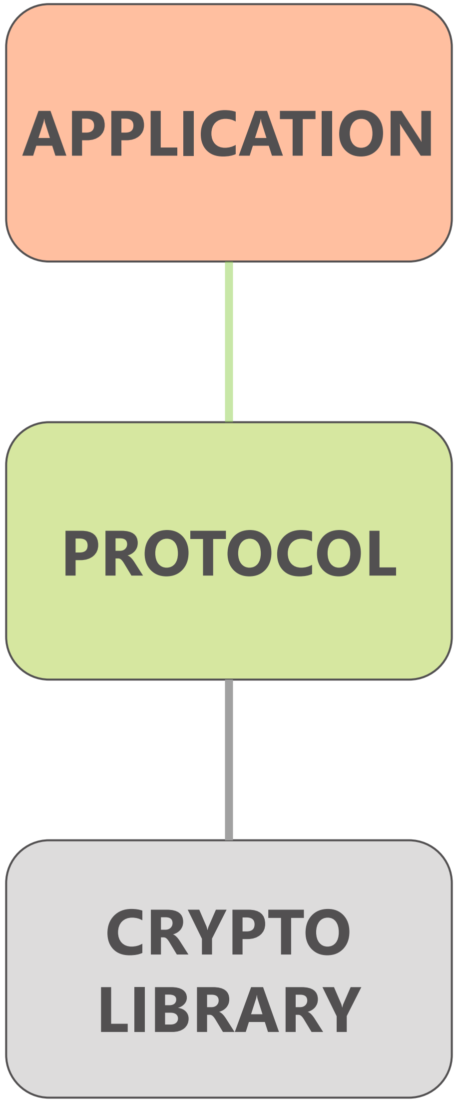
Final Thoughts

Many pitfalls in cryptographic software

- Need to verify their design+implementation
- Need to verify crypto+protocol+application

Formal security proofs for real-world crypto protocols are now feasible

- TLS 1.3 is an ongoing successful experiment
- Similar results for SSH, Signal, etc.
- Many tools: ProVerif, CryptoVerif, F*,
  Tamarin, EasyCrypt, VST
- Try them out to build your next proof, or to implement your crypto protocols securely!



End of Part IV