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

deq 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
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 `retriever 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

\[ \text{val attacker\_knows: } \text{trace} \rightarrow \text{bytes} \rightarrow \text{Type}_0 \]

- 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: \text{concat, split, ae\_enc, ae\_dec, pk\_enc, pk\_dec, sign, hash, ...}

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


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

(\* Some Protocol Code \*)
val sendMsg1: principal \rightarrow \text{principal} \rightarrow \text{trace} \rightarrow \text{trace}
val recvMsg1: principal \rightarrow \text{trace} \rightarrow \text{trace}

(\* Reachability \*)
let rec reachable (tr:trace): Type =
(\exists p_1 p_2 \ tr'. \ tr == sendMsg1 p_1 p_2 \ tr') \lor (\exists p \ tr'. \ tr == recvMsg1 p \ tr') \lor
(match tr with
  | [] \rightarrow \top
  | FreshGen p::tr' \rightarrow \text{List.mem} (\text{Compromise} p) \ tr' \land \text{reachable} \ tr'
  | Send p_1 p_2 m::tr' \rightarrow \text{attacker_knows} \ tr' \ m \land \text{reachable} \ tr'
  | \text{Compromise} p::tr' \rightarrow \text{reachable} \ tr'
  | _ \rightarrow \bot)

Stating Secrecy Goals

```ocaml
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 (\forall tr p m. (reachable tr \land protocol_sent p m tr \land attacker_knows tr m) \Rightarrow 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

\[
\text{let protocol\_sent p_1 p_2 secret tr = ...} \\
\text{let protocol\_received p_1 p_2 secret tr = ...} \\
\text{let authentication\_lemma ():} \\
\quad \text{Lemma (\forall tr p m. (reachable tr \land} \\
\quad \quad \text{protocol\_received p_1 p_2 m tr) \implies} \\
\quad \quad \text{(protocol\_sent p_1 p_2 m tr \lor} \\
\quad \quad \quad \text{compromised p_1 tr))}
\]

- **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:
$sk_I, R \leftrightarrow pk_R, M \leftrightarrow pk_M, \ldots$

Initiate:
genenerate $n_I$

Responder $R$

Prior Knowledge:
$sk_R, I \leftrightarrow pk_I, M \leftrightarrow pk_M, \ldots$

Respond:
genenerate $n_R$

$pk_R, MSG_1 \mid I \mid n_I$

$pke_{-}enc(pk_I, MSG_2 \mid n_I \mid n_R)$

$pke_{-}enc(pk_R, MSG_3 \mid n_R)$

InitiatorSessionKey:
$I \leftrightarrow R : n_R$

ResponderSessionKey:
$I \leftrightarrow R : n_R$
Lowe’s Attack on NS-PK

**Initiator** $I$

**Prior Knowledge:**

$sk_I.R \leftrightarrow pk_R$, $M \leftrightarrow pk_M$, ...

**Initiate:**

generate $n_I$

**Attacker** $M$

**Prior Knowledge:**

$sk_M.I \leftrightarrow pk_I$, $R \leftrightarrow pk_R$, ...

**Attack:**

$pke_{enc}(pk_M, MSG_1 | I | n_I)$

**Responder** $R$

**Prior Knowledge:**

$sk_R, I \leftrightarrow pk_I$, $M \leftrightarrow pk_M$, ...

**Respond:**

generate $n_R$

$pke_{enc}(pk_M, MSG_3 | n_R)$

$pke_{enc}(pk_I, MSG_2 | n_I | n_R)$

$pke_{enc}(pk_R, MSG_3 | n_R)$

**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

```ocaml
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
(* 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) =
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*

Global Trace

Protocol Code
Security Goals

Crypto Interface
Storage Interface
Messaging Interface
Security Lemmas

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

Verifying (F*)

Potential Attack

Security Theorem

Soundness Theorem (proven once and for all)
Secrecy Labels for Bytstrings

Who can read a secret?
• Public: anybody
• CanRead [P a; P b]: a or b

```haskell
import Data.List (intercalate)

data Principal = String

type Principal = String

data St_id =
  | P Principal -> St_id
  | S Principal -> SessionNat -> St_id
  | V Principal -> SessionNat -> VersionNat -> St_id

data Label =
  | Public: Label
  | Can_Read: List St_id -> Label
  | Meet: Label -> Label -> Label
  | Join: Label -> Label -> Label

val can_flow: Timestamp -> Label -> Label -> Pred
```
Secrecy Labels for Bytstrings

Meet (Join (Can_Read [P i]) (Can_Read [P r]))
  (Meet (Join (Can_Read [V i \textit{sid}_i \ 0]) (Can_Read [P r]))
   (Meet (Join (Can_Read [V i \textit{sid}_i \ 0]) (Can_Read [P r])))
    (Join (Can_Read [V i \textit{sid}_i \ 0]) (Can_Read [V r \textit{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 \lor 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 Session State"
    ))
  | Success _ → error "Unexpected success 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

Abstract labeled APIs proved sound in F*

Trace-based symbolic runtime model in F*

Executable protocol + app code verified for security
DY*: scalable security verification

<table>
<thead>
<tr>
<th>Modules</th>
<th>FLoC</th>
<th>PLoC</th>
<th>Verif. Time</th>
<th>Primitives</th>
</tr>
</thead>
<tbody>
<tr>
<td>Generic DY*</td>
<td>9</td>
<td>1,536</td>
<td>1,344</td>
<td>≈ 3.2 min</td>
</tr>
<tr>
<td>NS-PK</td>
<td>4</td>
<td>439</td>
<td>-</td>
<td>(insecure)</td>
</tr>
<tr>
<td>NSL</td>
<td>5</td>
<td>340</td>
<td>188</td>
<td>≈ 0.5 min</td>
</tr>
<tr>
<td>ISO-DH</td>
<td>5</td>
<td>424</td>
<td>165</td>
<td>≈ 0.9 min</td>
</tr>
<tr>
<td>ISO-KEM</td>
<td>4</td>
<td>426</td>
<td>100</td>
<td>≈ 0.7 min</td>
</tr>
<tr>
<td>Signal</td>
<td>8</td>
<td>836</td>
<td>719</td>
<td>≈ 1.5 min</td>
</tr>
</tbody>
</table>

Proofs require between 50% and 90% annotation overhead. Size of annotation depends on complexity of security goals.
Sign-then-Encrypt Protocol

Initiator $I$

Initially Knows:

$$sk_i, pk_r$$

Exchange:

$I \leftrightarrow R : req; resp$

Responder $R$

Initially Knows:

$$sk_r, pk_i$$

Exchange:

$I \leftrightarrow R : req; resp$

$penc(pk_r, sign(sk_i, 0||req))$

$penc(pk_i, sign(sk_r, 1||resp))$

Is this secure?
Man-in-the-Middle Attack

**Initiator** $I$

- Initially Knows: $sk_I, pk_O, pk_R$
- Exchange: $I \leftrightarrow O: req, \bot$

**Attacker** $O$

- Initially Knows: $sk_O, pk_I, pk_R$
- penc$(pk_O, \text{sign}(sk_I, 0||req))$
- penc$(pk_r, \text{sign}(sk_I, 0||req))$
- penc$(pk_i, \text{sign}(sk_r, 1||resp))$

**Responder** $R$

- Initially Knows: $sk_R, pk_O, pk_i$
- 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)

- encrypt then-MAC
- fragment-MAC-encode-then-encrypt

- authenticated encryption
- INT-CMA
- IND-CPA

- Secure RPC
- TLS 1.2

- 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

some cryptographic implementation

authenticated encryption

Secure RPC

RPC API

Adversary Model

any typed F# program

active adversaries

Bytes

Networking

system libraries

Formatting

message format

security protocols

any typed F# program

application code
Sample modular verification (crypto)

RPC using Encrypt-then-MAC

cryptographic schemes

cryptographic constructions

probabilistic computational indistinguishability

AES-CBC encryption
≈ IDEAL
IND-CPA

MAC authentication
≈ IDEAL
INT-CMA

Encrypt-then-MAC
authenticated encryption

Secure RPC
RPC API

Formatting
message format

Bytes
Networking

system libraries

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

```fsharp
module MAC

type text = bytes  val macsize

val key = bytes
val mac = bytes

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

This interface says nothing on the security of MACs.
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

```
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
```
module MAC

type text = bytes
val macsize : bytes

val key : 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)}

MAC keys are abstract

MACs are fixed sized

MACs are fixed sized

Msg is specified by protocols using MACs

"All verified messages have been MACed"
module MAC

open System.Security.Cryptography

let macsize = 20

let GEN() = randomBytes 16

let MAC k t = (new HMACSHA1(k)).ComputeHash t

let VERIFY k t m = (MAC k t = m)
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

CMA game (coded in F#)
Computational Safety for MACs

Ideal system

- **Mac**
  - F# interface
- **Ideal filter**
  - Ideal MAC

Concrete system

- **Mac**
  - F# interface

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

Any p.p.t. adversary

Perfectly safe by typing

\[ \text{safe too, with probability } 1 - \frac{1}{\varepsilon} \]
Sample ideal functionality:

Supporting Key Compromise

module MAC

val macsize = text = bytes

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

val keysize

type keybytes = b:bytes{Length(b)=keysize}

val LEAK: k:key{!t. Msg(k,t)} -> b:keybytes

val COERC: b:keybytes{...} -> k:key{...}

MAC keys are abstract

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

\[ I_\alpha = \alpha, \ldots, x : T_\alpha, \ldots \]
\[ P_\alpha \text{ range over pure modules such that } \vdash P_\alpha \sim I_\alpha. \]

**Theorem** (Secrecy by Typing).
Let \( A \) such that \( I_\alpha \vdash A : \text{bool} \).
For all \( P^0_\alpha \) and \( P^1_\alpha \), 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

```plaintext
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
```

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

The size of plaintext is fixed (as we cannot hide it)
Ideal Interface for Authenticated Encryption

- 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**

<table>
<thead>
<tr>
<th>Module</th>
<th>AE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Open</td>
<td>Plaintext</td>
</tr>
<tr>
<td>Type key</td>
<td></td>
</tr>
<tr>
<td>Type cipher = b:bytes{Length(b)= size + 16}</td>
<td></td>
</tr>
<tr>
<td>Val GEN: unit -&gt; key</td>
<td></td>
</tr>
<tr>
<td>Val ENC: key -&gt; plain -&gt; cipher</td>
<td></td>
</tr>
<tr>
<td>Val DEC: key -&gt; cipher -&gt; plain option</td>
<td></td>
</tr>
</tbody>
</table>

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

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

```plaintext
module PKENC
open Plain
val pksize: int
type skey
type pkey = b:bytes{ PKey(b) \( \mathcal{E} \)}

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
```
Typed Secrecy from CCA2-Secure Encryption

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

\[ P^0 \cdot C_{ENC} \cdot A \approx_{\varepsilon} P^1 \cdot C_{ENC} \cdot A. \]

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

\[ P \cdot C_{ENC} \cdot A \approx_{\varepsilon} P \cdot C_{ENC} \cdot F_{ENC} \cdot A. \]
Variants: CPA & Authentication

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

  \[
  \text{predicate } \textit{Encrypted} \text{ of } \text{key} \times \text{cipher} \\
  \text{val } \text{ENC: } k : \text{key} \rightarrow \text{plain} \rightarrow c : \text{cipher}\{\text{Encrypted}(k, c)\} \\
  \text{val } \text{DEC: } k : \text{key} \rightarrow c : \text{cipher}\{\text{Encrypted}(k, c)\} \rightarrow \text{plain}
  \]

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

  \[
  \text{predicate } \textit{Msg} \text{ of } \text{key} \times \text{plain} \text{ // defined by protocol} \\
  \text{val } \text{ENC: } k : \text{key} \rightarrow p : \text{plain}\{\text{Msg}(k, p)\} \rightarrow \text{cipher} \\
  \text{val } \text{DEC: } k : \text{key} \rightarrow \text{cipher} \rightarrow p : \text{plain}\{\text{Msg}(k, p)\} \text{ option}
  \]
Modular Architecture for miTLS

Base: CoreCrypto, Bytes, TCP, TLSConstants, TLSInfo, Error, Range

Handshake/CCS:
- Cert
- RSAKey
- DHGroup
- CRE
- Nonce
- Extensions
- Handshake (and CCS)

Alert Protocol:
- Alert

AppData Protocol:
- Datastream
- AppData

TLS API:
- Dispatch

TLS Record:
- MAC
- Encode
- Enc
- LHAEPlist
- LHAE
- StPlain
- StAE
- TLSFragment
- Record

Application:
- AuthPlain
- Auth
- RPCPlain
- RPC

Adversary:
- Untyped API
- Untyped Adversary
Each application provides its own plaintext module for data streams:
- Typing ensures secrecy and authenticity at safe indexes
- Parameters select ciphersuites and certificates
- Results provide detailed information on the protocol state
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