Part 4: Towards High-Assurance Cryptographic Software
Implementing & Verifying Crypto Protocols in F*
Sign-then-Encrypt Protocol

Initially Knows:

$\text{Initiator } I$

$sk_i, pk_r$

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

Exchange:

$I \leftrightarrow R : req; resp$

Initially Knows:

$\text{Responder } R$

$sk_r, pk_i$

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

Exchange:

$I \leftrightarrow R : req; resp$

Is this secure?
Identity Misbinding Attack on Sign-then-Encrypt

**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**

- **cryptographic algorithms**
- **typed interfaces:**
  - **cryptographic assumptions**
- **cryptographic constructions**
- **typed interfaces:**
  - **security guarantees**
- **security protocols**
- **typed interfaces:**
  - **attacker models**
- **active adversaries**
RPC protocol using Authenticated Encryption

Sample modular verification (protocol)
Sample modular verification (crypto)

RPC using Encrypt-then-MAC

- Cryptographic schemes
- Cryptographic constructions
- Probabilistic computational indistinguishability
- Adversary Model
- Active adversaries

- AES-CBC encryption
  - ≈ IDEAL (IND-CPA)
- MAC authentication
  - ≈ IDEAL (INT-CMA)
- Encrypt-then-MAC
  - Authenticated encryption
- Secure RPC
  - RPC API
- Formatting
  - Message format
- Any typed F7 program
- Application code
- System libraries
- Networking
- Security protocols

Any typed F# program
MAC : integrity
Sample functionality:

Message Authentication Codes

```fsharp
module MAC

type text = bytes
val macsize : int

type key = bytes

type 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
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

MACs are fixed sized

Sample functionality:
Message Authentication Codes
### Sample functionality:

**Message Authentication Codes**

- **MAC keys are abstract**
- **MACs are fixed sized**
- **Msg is specified by protocols using MACs**
- **ideal F* interface**

```
module MAC

type text = bytes
val macsize : nat

type key = t

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) }
```

“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)

module MAC

type text = bytes
val macsize : int

val macsize : int

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)}
Sample computational assumption:

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

```fsharp
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
```

**CMA game (coded in F#)**

**Computational Safety**

A probabilistic polytime program calling `mac` and `verify` forges a MAC only with negligible probability.
Computational Safety for MACs

ideal system

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

- RPC protocol
  - secure RPC

= Ideal MAC

\[ \approx \]

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

safe too, with probability \( 1 - \frac{1}{\varepsilon} \)
Sample ideal functionality:

**Supporting Key Compromise**

```
module MAC

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) }

val keysize : bytes

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

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

val COERCE : b : keybytes { } -> k : key { }
```

MAC keys are abstract

MACs are fixed sized

MAC keys have concrete representations

Ideal F* interface

```
ideal F*

interface

"All verified messages have been MACed"
```

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 \( I_{\alpha} \vdash A : bool \).

For all \( P^0_{\alpha} \) and \( P^1_{\alpha} \), we have \( P^0_{\alpha} \cdot A \approx P^1_{\alpha} \cdot A. \)
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.
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

```
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

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.

```ocaml
module PKENC
open Plain
val pksize: int
type skey
type pkey = b:bytes{ PKey(b) \AE }
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 $\vdash P^b \leadsto I_{\text{PLAIN}}$.
Let $C_{\text{ENC}}$ p.p.t. CCA2-secure with $I_{\text{PLAIN}}^C \vdash C_{\text{ENC}} \leadsto I_{\text{ENC}}^C$.
Let $A$ p.p.t. with $I_{\text{PLAIN}}, I_{\text{ENC}} \vdash A : \text{bool}$.

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

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

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

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

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

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

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

```haskell

// 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) ioreRESULT_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) ioreRESULT_o

// reading data

type (;c:cn) ioreRESULT_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) ioreRESULT_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