Concrete cryptographic security in F*
Modular Code-Based Crypto Verification

crypto hash (SHA3)
symmetric encryption (AES)
public-key encryption (RSA)

encrypt then-MAC
hybrid encryption
Secure RPC
TLS 1.2

INT-CMA
IND-CMA, CCA2
IND-CMA, CCA2
secure channels

auth. encryption

auth. encryption
typed interfaces
auth. encryption
typed interfaces
typed interfaces

crypto primitives
crypto constructions
security protocols
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 = bytes 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)

This ideal interface uses a log to specify security.

Great for F* verification.

Unrealistic: tags can be guessed.
Our ideal interface reflects the security of a chosen-message game [Goldwasser’88]

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

UF-CMA programmed in F*

```f star
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`

- **Concrete system**
  - `Mac`
    - real interface
  - `RPC protocol`
  - `Any p.p.t. adversary`

- `≈ ε`

- `UF-CMA adversary`
  - perfectly safe by typing
  - `sample protocol`
    - typed against ideal MAC interface
  - `protocol adversary`
    - typed against RPC interface

- Concrete algorithm assumed UF-CMA computationally
  - log-based error correction
    - making VERIFY returns false on forgeries
  - safe too, with probability $1 - \epsilon$
Cryptographic Integrity: Two styles for ideal MACs

**module** MAC (* stateful *)

**type** key

**val** log: mem → key → Seq msg

**val** keygen:

unit → ST key

(ensures \( \lambda h_0 \ k \ h_1 \rightarrow \log h_1 \ k = \text{empty} \))

**val** mac:

\( k:\text{key} \rightarrow m:\text{msg} \rightarrow \text{ST tag} \)

(ensures \( \lambda h_0 \ t \ h_1 \rightarrow \log h_1 \ k = \log h_0 \ k + + m \))

**val** verify:

\( k:\text{key} \rightarrow m:\text{msg} \rightarrow t:\text{tag} \rightarrow \text{ST bool} \)

(ensures \( \lambda h_0 \ b \ h_1 \rightarrow b \implies \text{mem} \ (\log h_0 \ k) \ m \))

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

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


**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 \implies p m\})
Security programming example

Authenticated RPC
Authenticated RPC

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

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

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.
Authenticating RPC Test

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

Connecting to localhost:8080
Sending \{BgAyICsgMj9mhJa7iDAcW3Rrk...\} (28 bytes)
Listening at ::1:8080
Received Request 2 + 2?
Sending \{AQA0NccjcuL/WOaYSOGGtOtPm...\} (23 bytes)
Received Response 4
Authenticated RPC: Is this Protocol Secure?

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

Security depends on the following:

(1) The function $\text{hmacsha1}$ 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 $\text{request}$ and $\text{response}$ 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

cryptographic primitives

HMAC
mac.fst

Formatting
format.fst

Authenticated RPC
rpc.fst

Bytes, Network
lib.fst

system libraries

security protocols

typed interfaces
(modular design)

plain typed interfaces
(attacker model)

active adversaries

any typed F* program

application code

typed interfaces
(security assumptions)
Another sample crypto assumption

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

\[ \forall (x, y: \text{bytes}). H(x) = H(y) \implies x = y \]

This is modelled by maintaining an inverse, monotonic table from hash tags to hashed bytestrings.
For authentication, we often require hash algorithms to be “computationally injective”

\[ \forall (x, y: \text{bytes hashed so far}). \quad H(x) = H(y) \implies x = y \]

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

\[
\text{Game } \text{Ae}(\mathcal{A}, \text{AE})\\
\begin{align*}
& b \leftarrow \{0, 1\}; \quad L \leftarrow \varnothing; \quad k \leftarrow \text{AE.keygen()}\\
& b' \leftarrow \mathcal{A}^\text{Encrypt,Decrypt}(); \quad \text{return } (b \equiv b')
\end{align*}
\]

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

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

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

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
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