Cryptographic Protocol Verification

Thanks for the slides to Cedric Fournet
Crypto protocols go wrong

- Historically, one keeps finding simple attacks against protocols
  - even carefully-written, widely-deployed protocols,
    even a long time after their design & deployment
  - simple = no need to break cryptographic primitives

- Why is it so difficult?
  - breaking functional abstractions
  - concurrency + distribution + cryptography
    - Little control on the runtime environment
  - active attackers
    - hard to test
  - implicit assumptions and goals
    - Authenticity, secrecy
The Needham-Schroeder problem

In Using encryption for authentication in large networks of computers (CACM 1978), Needham and Schroeder didn’t just initiate a field that led to widely deployed protocols like Kerberos, SSL, SSH, IPSec, etc.

They threw down a gauntlet.

“Protocols such as those developed here are prone to extremely subtle errors that are unlikely to be detected in normal operation.

The need for techniques to verify the correctness of such protocols is great, and we encourage those interested in such problems to consider this area.”
The Needham-Schroeder public-key authentication protocol (CACM 1978)

Principal A initiates a session with principal B
S is a trusted server returning public-key certificates eg { | A,KA | }_{KS^{-1}}
NA,NB serve as nonces to prove freshness of messages 6 and 7
Assuming A knows KB and B knows KA, we get the core protocol:

More precisely, the goals of the protocol are:
• After receiving message 6, A believes NA,NB shared just with B
• After receiving message 7, B believes NA,NB shared just with A

If these goals are met, A and B can subsequently rely on keys derived from NA,NB to efficiently secure subsequent messages.
A certified user $M$ can play a man-in-the-middle attack (Lowe 1995)

This run shows a certified user $M$ can violate the protocol goals:
- After receiving message 6, $A$ believes NA,NB shared just with $M$
- After receiving message 7, $B$ believes NA,NB shared just with $A$

(Writing in the 70s, Needham and Schroeder assumed certified users would not misbehave; we know now they do.)
A brief history: 1978—

We assume that an intruder can interpose a computer on all communication paths, and thus can alter or copy parts of messages, replay messages, or emit false material. While this may seem an extreme view, it is the only safe one when designing authentication protocols.

Needham and Schroeder CACM (1978)

1978: N&S propose authentication protocols for “large networks of computers”
1981: Denning and Sacco find attack on N&S symmetric key protocol
1983: Dolev and Yao first formalize secrecy properties of NS threat model using formal algebra
1987: Burrows, Abadi, Needham invent authentication logic; incomplete, but useful
1994: Hickman, Elgamal invent SSL; holes in v1, v2, but v3 fixes these, very widely deployed
1994: Ylonen invents SSH; holes in v1, but v2 good, very widely deployed
1995: Lowe finds insider attack on N&S asymmetric protocol; rejuvenates interest in FMs
circa 2000: Several FMs for “Dolev & Yao problem”: tradeoff between accuracy and approximation
circa 2007: Many FMs developed; several deliver both accuracy and automation
2014: dozens of attacks against mainstream TLS implementations
Symbolic Cryptography

- **Symbolic models** reason about fully specified primitives
  - Same rules apply for the attacker as for the protocol
  - Each crypto primitives yields distinct symbolic terms
  - No probabilities, no PPT Turing machines…
Dolev-Yao

\[ \text{Mac}(k,m) \]
\[ \text{Enc}(k,m) \]

- // macs cannot be reverted

\[ \text{dec}(k,\text{Enc}(k,m)) = m \]
Dolev-Yao vs Sealing

Mac(k,m)  Enc(k,m)
- \( \text{macs cannot be reverted} \)
\( \text{dec(k,Enc(k,m))=}m \)

k: ‘a -> bytes (sealing function)
k: (‘a -> bytes) * (bytes -> ‘a) (sealing & unsealing function)

Term rewriting

Constructor applications

List(‘a * bytes) // kept secret
Computational Cryptography

• **Computational models** reason about partially-specified primitives (the less specific, the better)
  
  - *Positive assumptions*: what the protocol needs to run as intended, e.g., successful decryption when using right keys
  
  - *Negative assumptions*: what the adversary cannot do, e.g., cannot distinguish between encryptions of two different plaintexts

• Security proofs apply parametrically, for any concrete primitives that meet these assumptions
• Authentication and secrecy properties for basic crypto protocols have been formalized and thoroughly studied.

• After intense effort on symbolic reasoning, several techniques and tools are available for automatically proving these properties:
  - e.g. Athena, TAPS, ProVerif, CryptoVerif, FDR, AVISPA, F7, etc.

• We can automatically verify most security properties for detailed models of crypto protocols:
  - e.g. IPSEC, Kerberos, Web Services, Infocard, TLS.
2015: done?

• Best practice: apply formal methods and tools throughout the protocol design & review process

• Not so easy
  - Specifying a protocol is a lot of work
  - Practitioners don’t understand formal models & tools

• Some troublesome questions
  1. How to relate formal models to protocol implementations?
  2. How to relate crypto protocols to application security?
  3. How to relate symbolic models to computational cryptography?
Filling the gap between symbolic and computational models

- Even if you prove the symbolic protocol correct, attacks can still be mounted by exploiting the algebraic properties of the underlying cryptographic primitives.

- Much research has been done to fill the gap and to ensure that symbolic security entails computational security (computational soundness).
  
  - current results mostly limited to trace properties (almost no indistinguishability-based security)
  
  - proofs very long and error-prone
Models vs implementations

• Protocol specifications remain largely informal
  - They focus on message formats and interoperability, not on local enforcement of security properties

• Models are short, abstract, hand-written
  - They ignore large functional parts of implementations
  - Their formulation is driven by verification techniques
  - It is easy to write models that are safe but dysfunctional (testing & debugging is difficult)

• Specs, models, and implementations drift apart…
  - Even informal synchronization involves painful code reviews
  - How to keep track of implementation changes?
From code to model

• Our approach: we directly verify **reference implementations**
  treated as “giant” protocol models

• Executable code is more detailed than models
  - Some functional aspects can be ignored for security
  - Model extraction can safely erase those aspects

• Executable code has better tool support
  - Types, compilers, debuggers, libraries, verification tools
Specs, code, and formal tools

Protocol Standards
- TLS
- Kerberos
- WS-Security
- IPsec
- SSH

Protocol Implementations and Applications
- C/C++
- Java
- C#
- F#
- ML

Theorem Provers
- ProVerif ('01)
- F7 ('08)
- F* ('11)
- Cryptyc
- AVISPA
- Scyther
- NRL
- Securify
- Athena
- Casper

Computational Models
- Hand Proofs
  - CryptoVerif ('06)
  - EasyCrypt ('11)
  - F7 ('11)
  - RF* ('13)

Symbolic Models
- SMT Solvers
- Model Checkers
- General Verification
Goal: verify production code relying on cryptography

- Communications Protocol (IPSEC, TLS)
- Cryptographic libraries (XML security, WS* standards, TCG)
- Security Components (InfoCard, DKM, TPM)
A First Non-Cryptoish Example: Access Control in Partially Trusted Code
Defining the policy

```ocaml
module FileName
    type filename = string

module ACLs
    open FileName

(* canWrite is a function specifying whether or not a file `f` can be written *)
let canWrite (f:filename) =
    match f with
    | "C:/temp/tempfile" -> true
    | _ -> false

(* canRead is also a function ... *)
let canRead (f:filename) =
    canWrite f (* writeable files are also readable *)
    || f="C:/public/README" (* and so is this file *)
```
A Secure Typed Interface

module System.IO
    open ACLs
    open FileName
    assume val read : f:filename{canRead f} -> string
    assume val write : f:filename{canWrite f} -> string -> unit

• Refinements enforce security
module UntrustedClientCode
    open System.IO
    open FileName

let passwd  = "C:/etc/password"
let readme  = "C:/public/README"
let tmp     = "C:/temp/tempfile"

let staticChecking () =
    let v1 = read tmp in
    let v2 = read readme in
    (* let v3 = read passwd in -- invalid read, fails type-checking *)
    write tmp "hello!"
    (* ; write passwd "junk" -- invalid write, fails type-checking *)
exception InvalidRead
val checkedRead : filename -> string
let checkedRead f =
  if ACLs.canRead f then System.IO.read f else raise InvalidRead

assume val checkedWrite : filename -> string -> unit

let dynamicChecking () =
  let v1 = checkedRead tmp in
  let v2 = checkedRead readme in
  let v3 = checkedRead passwd in
  checkedWrite tmp "hello!";
  checkedWrite passwd "junk" (* this raises exception *)
Example: MAC
A Typed Interface

We define a property that must hold true whenever we MAC a message, and attach it to the key (different keys may have different properties).

```
module MAC
open Array
open SHA1

opaque type key_prop : key -> text -> Type
type pkey (p:(text -> Type)) = k:key{key_prop k == p}

val keygen: p:(text -> Type) -> pkey p
val mac: k:key -> t:text{key_prop k t} -> tag
val verify: k:key -> t:text -> tag -> b:bool{b ==> key_prop k t}
```
We define a property that must hold true whenever we MAC a message, and attach it to the key (different keys may have different properties).

The property is a precondition of `mac` and postcondition of `verify`.

We will use, for instance, this property in the next protocol we will see.
The implementation

```
val keygen: p:(text -> Type) -> pkey p
val mac: k:key -> t:text{key_prop k t} -> tag
val verify: k:key -> t:text -> tag -> b:bool{b ==> key_prop k t}
```

```
let keygen (p: (text -> Type)) =
  let k = sample keysize in
  assume (key_prop k == p);
  k

let log = ST.alloc (list entry) []

let mac k t =
  let m = sha1 k t in
  log := Entry k t m :: !log;
  m

let verify k text tag =
  let verified = sha1verify k text tag in
  let found =
    is_Some
      (List.find
        (fun (Entry k' text' tag') -> k=k' && text=text')
        !log)
  in
  verified && found
```

Symbolic crypto: the log keeps track of the macs

```
(* plain, concrete implementation (ignoring the log) *)
//verified

(* symbolic, error-correcting implementation *)
verified && found
```
Access control in untrusted code

• We can extend the example on access control with cryptographic capabilities, so that our trusted ACLs library can now issue and accept HMACSHA1 tags as evidence that a given file is readable

• Homework!
Authenticated RPC

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

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

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.
Event-Based Security Policies: 

**Assume** and **Assert**

- Suppose there is a global set of events, the log.

- To evaluate **assume** $C$, add $C$ to the log, and return ().

- To evaluate **assert** $C$, return ().
  - If $C$ logically follows from the logged formulas, we say the assertion succeeds; otherwise, the assertion fails.
  - The log is only for specification purposes; it does not affect execution.
  - Refinement types carry logical properties, from assumptions to assertions.
  - Type safety guarantees that all assertions will succeed.
Logical Specification

For simplicity, we model only two parties, so we drop identifiers from types.
Protocol Code

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

```
assume \( \text{val} \ \text{send}: \text{message} \rightarrow \text{unit} \)
assume \( \text{val} \ \text{recv}: (\text{message} \rightarrow \text{unit}) \rightarrow \text{unit} \)

let \( \text{client} \ (s:\text{string16}) = \)
    assume \((\text{Request} \ s)\);
    send \((\text{append} \ (\text{utf8} \ s) \ (\text{mac} k \ (\text{request} \ s)))\);
    recv \((\text{fun} \ \text{msg} \rightarrow \)
    if \(\text{length} \ \text{msg} < \text{macsize} \) then failwith "Too short"
    else
        let \((v, m') = \text{split} \ \text{msg} \ (\text{length} \ \text{msg} - \text{macsize})\) in
        let \(t = \text{iutf8} \ v\) in
        if verify \(k \ (\text{response} \ s \ t) \) \(m'\)
        then assert \((\text{Response} \ s \ t))\)

let \( \text{server} () = \)
    recv \((\text{fun} \ \text{msg} \rightarrow \)
    if \(\text{length} \ \text{msg} < \text{macsize} \) then failwith "Too short"
    else
        let \((v, m) = \text{split} \ \text{msg} \ (\text{length} \ \text{msg} - \text{macsize})\) in
        if \(\text{length} \ v > 65535\) then failwith "Too long"
        else
            let \(s = \text{iutf8} \ v\) in
            if verify \(k \ (\text{request} \ s) \) \(m\)
            then \(\text{assert} \ (\text{Request} \ s)\);
            let \(t = \text{"22"}\) in
            assume \((\text{Response} \ s \ t)\);
            send \((\text{append} \ (\text{utf8} \ t) \ (\text{mac} k \ (\text{response} \ s \ t)))\)
```

Protocol Code

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

```ocaml
assume val send: message -> unit
assume val recv: (message -> unit) -> unit

let client (s:string16)
    assume (Request s);
    send (append (utf8 s) (mac k (request s)));
    recv (fun msg ->
        if length msg < msize then failwith "Too short"
        else
            let (v, m') = split msg (length msg - msize) in
            let t = iutf8 v in
            if verify k (response s t) m'
            then assert (Response s t))

let server () =
    recv (fun msg ->
        if length msg < msize then failwith "Too short"
        else
            let (v, m) = split msg (length msg - msize) in
            if length v > 65535 then failwith "Too long"
            else
                let s = iutf8 v in
                if verify k (request s) m
                then (assert (Request s);
                    let t = "22" in
                    assume (Response s t);
                    send (append (utf8 t) (mac k (response s t))))

```
Protocol Code

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

```ocaml
assume val send: message -> unit
assume val recv: (message -> unit) -> unit

let client (s:string16);
  assume (Request s);
  send (append (utf8 s) (mac k (request s)));
  recv (fun msg ->
    if length msg < msize then failwith "Too short"
    else
      let (v, m') = split msg (length msg - msize) in
      let t = iutf8 v in
      if verify k (response s t) m'
      then assert (Response s t))

let server () =
  recv (fun msg ->
    if length msg < msize then failwith "Too short"
    else
      let (v,m) = split msg (length msg - msize) in
      if length v > 65535 then failwith "Too long"
      else
        let s = iutf8 v in
        if verify k (request s) m
        then
          ( assert (Request s);
            let t = "22" in
            assume (Response s t);
            send (append (utf8 t) (mac k (response s t))))
```

Protocol Code

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

```ocaml
assume \text{val} \text{send} : \text{message } \rightarrow \text{unit}
assume \text{val} \text{recv} : (\text{message } \rightarrow \text{unit}) \rightarrow \text{unit}

let \text{client } (s : \text{string16}) =
  assume (\text{Request } s);
  send (append (\text{utf8 } s) (\text{mac } k (\text{request } s)))
  \text{recv (fun } \text{msg } \rightarrow \text{)}
    \text{if length } \text{msg} < \text{macsize then failwith } "\text{Too short}"
    \text{else}
    \text{let } (v, m') = \text{split } \text{msg} (\text{length } \text{msg} - \text{macsize}) \text{ in}
    \text{let } t = \text{iutf8 } v \text{ in}
    \text{if verify } k (\text{response } s t) m' \text{ then assert (Response } s t)\text{)}

\text{let server } () =
  \text{recv (fun } \text{msg } \rightarrow \text{)}
    \text{if length } \text{msg} < \text{macsize then failwith } "\text{Too short}"
    \text{else}
    \text{let } (v, m) = \text{split } \text{msg} (\text{length } \text{msg} - \text{macsize}) \text{ in}
    \text{if length } v > 65535 \text{ then failwith } "\text{Too long}"
    \text{else}
    \text{let } s = \text{iutf8 } v \text{ in}
    \text{if verify } k (\text{response } s t) m \text{ then}
      (\text{assert (Request } s)\text{)}
      \text{let } t = "22" \text{ in}
      \text{assume (Response } s t)\text{)}
      \text{send (append (\text{utf8 } t) (\text{mac } k (\text{response } s t)))}\text{)}
```
Protocol Code

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

```ocaml
assume val send: message -> unit
assume val recv: (message -> unit) -> unit

let client (s: string16) =
    assume (Request s);
    send (append (utf8 s) (mac k (request s)));
    recv (fun msg ->
        if length msg < mcsize then failwith "Too short"
        else
            let (v, m') = split msg (length msg - mcsize) in
            let t = iutf8 v in
            if verify k (response s t) m' then assert (Response s t))

let server () =
    recv (fun msg ->
        if length msg < mcsize then failwith "Too short"
        else
            let (v, m) = split msg (length msg - mcsize) in
            if length v > 65535 then failwith "Too long"
            else
                let s = iutf8 v in
                if verify k (request s) m then
                    (assert (Request s);
                    let t = "22" in
                    assume (Response s t);
                    send (append (utf8 t) (mac k (response s t))))
```

```
We require 3 lemmas on message formats:

- requests are injective on their argument
- responses are injective on both their arguments
- requests and responses are distinct

```ocaml
type message = seq byte
val request : string -> Tot message
let request s = append tag0 (utf8 s)

val response: string16 -> string -> Tot message
let response s t =
  let lb = uint16_to_bytes (length (utf8 s)) in
  append tag1 (append lb (append (utf8 s) (utf8 t)))
```
Modular verification for a sample protocol

cryptographic primitives

typed interfaces (security assumptions)

plain typed interfaces (attacker model)

active adversaries

Formatted
format.fst

HMAC
mac.fst

INT-CMA

Authenticated RPC
rpc.fst

adv.fst

any typed F* program

Bytes, Network lib.fst

system libraries

security protocols
typed interfaces (modular design)

any typed F* program

application code
Computational Verification: MACs and CMA

Sample ideal functionalities for keyed hash functions
Computational security by typing, formally

- We adapt the F7 (ancestor of F*) typechecker
  - removed non-determinism
  - added probabilistic sampling and mutable references

- We type protocols and applications against refined typed interfaces that idealize crypto libraries

- Focus on two sample ideal functionalities
  - for MACs (trace properties)
  - for encryption (indistinguishability properties)
Probabilistic F*

- We equip RCF (ancestor of F*, but it can be done for F* too) with a probabilistic semantics (Markov chains)

\[
\text{let } x_0 = \text{sample in } \ldots \text{let } x_{n-1} = \text{sample in } (x_0, \ldots, x_{n-1})
\]

reduces in \(2n\) steps to each binary \(n\)-word with probability \(\frac{1}{2^n}\) and models a uniform random generator.

- We add a typing rule for sampling

\[
E \vdash \diamond

\]

\[
E \vdash \text{sample : bool}
\]

- All typing theorems apply unchanged (one possible trace at a time)
Probabilistic RCF

- We equip RCF with a probabilistic semantics (Markov chains)

\[
[X, L, A] \rightarrow_p [X', L', B]
\]

- The rest of the semantics is unchanged

- For example, let \( x_1 = \text{sample} \) in \( \ldots \) let \( x_n = \text{sample} \) in \((x_1, \ldots, x_n)\) reduces in \(2n\) steps to each binary \(n\)-word with total probability 1
Runtime Safety A close expression $A$ is *safe* if and only if, in all evaluations of $A$, all assertions succeed, that is, each time an `assert` is evaluated, its formula logically follows from the previously assumed formulas. More generally, we define a probabilistic notion of safety. Let $P$ be a predicate over monotonic trace properties. The probability that $P(A)$ holds in $n \geq 0$ steps is defined as $q_n \triangleq \sum_{A \rightarrow p_1A_1, \ldots, \rightarrow p_nA_n} |P(p_1 \cdots p_n)$ where the sum ranges over distinct configurations $A_1, \ldots, A_n$ up to relabeling. ($A_i$ represent configurations with main expression $A_i$, and an expression $A$ is treated as the configuration with an empty store and an empty log.) The series $(q_n)_{n \geq 0}$ is positive, increasing, and bounded by 1, so it has a limit $q \in [0,1] = \lim_{n \to \infty} q_n$. We let $\mathbb{P}[A \downarrow M]$ be $q$ when $M$ is a closed value and $P$ is “$A_n$ has expression $M$”. We similarly define the probability that $A$ terminates, letting $P$ be “$A_n$’s expression is a value”, and that $A$ fails, letting $P$ be “some $A_i$ is a failing `assert`” (i.e. $A_i = [X_i,L_i,E[\text{assert } \phi]]$ for some evaluation context $E$ and formula $\phi$ that does not follow from $L_i$). Hence, $A$ is safe if and only if it fails with probability 0.
Type safety

- We use an auxiliary judgment \( I \vdash B \leadsto I' \)
  - \( B \) is well-typed in \( I \) and exports the typed interface \( I' \)

- Formally, if \( I \vdash B \leadsto I' \) then \( I \vdash B \cdot A : T \)

- \( I' \vdash A : T \)

- Hence, if \( \emptyset \vdash C_{PR} \leadsto I_{PR} \) then \( C_{PR} \cdot A \) is safe

- \( I_{PR} \vdash A : T \)
Type safety

- We use an auxiliary judgment \( I \vdash B \leadsto I' \)
  - \( B \) is well-typed in \( I \) and exports the typed interface \( I' \)

- If \( \emptyset \vdash B \leadsto I' \) then \( I \vdash B \cdot A : T \)
  - \( \emptyset \vdash A : T \)

- Hence, if \( \emptyset \vdash C_{PR} \leadsto I_{PR} \) then \( C_{PR} \cdot A \) is safe
  - \( I_{PR} \vdash A : T \)
Type safety

• We use an auxiliary judgment $I \vdash B \leadsto I'$

- If $B$ is well-typed in $I$ and exports the typed interface $I'$

• Protocol and cryptographic/networking library

$\vdash B \cdot A : T$

• Hence, if $\emptyset \vdash C_{PR} \leadsto I_{PR}$ then $C_{PR} \cdot A$ is safe

$I_{PR} \vdash A : T$
Type safety

- We use an auxiliary judgment $I \vdash B \leadsto I'$
  - If $I \vdash B \sim I'$ and $B$ is well-typed in $I$, then $B$ exports the typed interface $I'$
- $I \vdash B \leadsto I'$
- $B \cdot A : T$
- $I \vdash A : T$
- If $\emptyset \vdash C_{PR} \leadsto I_{PR}$ then $C_{PR} \cdot A$ is safe
- $I_{PR} \vdash A : T$
Type safety

- We use an auxiliary judgment \( I \vdash B \sim I' \)
- Formally, if \( \vdash B \sim I' \) then \( I' \)
- Hence, if \( \emptyset \vdash C_{PR} \sim I_{PR} \) then \( C_{PR} \cdot A \) is safe

Protocol and cryptographic/networking library
Typed interface for the attacker
Typed attacker
All functions given to the attacker must operate on plain, unrefined types
Computational complexity

- A series of probabilities \((q_\eta)_{\eta \geq 0}\) is negligible when, for any polynomial \(p\), we have \(\lim_{\eta \to \infty} (q_\eta \ast p(\eta)) = 0\); it is overwhelming when \(1 - q_\eta\) is negligible.

- A closed expression \(A\) is asymptotically safe when its series of probabilities of failing is negligible.

- Closed expressions \(A^0\) and \(A^1\) are asymptotically indistinguishable, written \(A^0 \simeq_\varepsilon A^1\), when \(|\Pr[A^0 \downarrow M] - \Pr[A^1 \downarrow M]|\) is negligible for all closed values \(M\).

- A closed expression \(A\) has probabilistic polynomial-time complexity, or is p.p.t. for short, when there exists a polynomial \(p\) such that, for all \(\eta \geq 0\), \(A_\eta\) terminates with probability 1 in at most \(p(\eta)\) steps.
Polynomial time complexity for modules

(1) A closed first-order functional value is p.p.t. when its runtime is bounded by a polynomial in the size of its parameters. In the definition below, we let \( B \) range over modules that just bind such values—this ensures that \( B \) does not perform any computation on its own, or use any shared state.

(2) An open expression \( A \) such that \( I \vdash A : T \) is p.p.t. when, for every \( \vdash B \sim I \), the expression \( B \cdot A \) is p.p.t.

(3) A module \( F \) such that \( I \vdash F \sim I_F \) is p.p.t. when, for every p.p.t. expression \( A \) such that \( I_F \vdash A \), the open expression \( F \cdot A \) is p.p.t.
MACs in F#

Interface

```fsharp
type key
val GEN: unit → key

type text = bytes
val MAC: key → text → mac

type mac = bytes
val VERIFY: key → text → mac → bool

val macsize: int
val LEAK: key → bytes

val keysize: int
```

Implementation

```fsharp
open System.Security.Cryptography

let keysize = 16 (* 128 bits *)
let macsize = 20 (* 160 bits *)

type key = bytes

let GEN () = randomBytes keysize

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

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

let LEAK (k: key) = k
```
Ideal interface (I)

```plaintext
type mac = b:bytes {Length(b) = macsize}
predicate val Msg: key * text → bool
val GEN: unit → key
val MAC: k:key → t:text {Msg(k,t)} → mac
val VERIFY: k:key → t:text → m:mac → v:bool {v=true ⇒ Msg(k,t)}
val LEAK: k:key {!t. Msg(k,t)} → b:bytes {Length(b) = keysize}
```
Ideal interface (I)

The interpretation of the `Msg` predicate is protocol dependent

define

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

predicate val \( Msg: \text{key} \times \text{text} \rightarrow \text{bool} \)

val \( GEN: \text{unit} \rightarrow \text{key} \)

val \( MAC: \text{k: \text{key}} \rightarrow \text{t: \text{text}}\{\text{Msg}(k,t)\} \rightarrow \text{mac} \)

val \( VERIFY: \text{k: \text{key}} \rightarrow \text{t: \text{text}} \rightarrow \text{m: \text{mac}} \rightarrow \text{v: \text{bool}}\{\text{v=\text{true} \Rightarrow \text{Msg}(k,t)}\} \)

val \( LEAK: \text{k: \text{key}}\{\text{!t. \text{Msg}(k,t)}\} \rightarrow \text{b: \text{bytes}}\{\text{Length}(b) = \text{keysize}\} \)
The interpretation of the `Msg` predicate is protocol dependent.
Ideal interface (I)

\[
\text{type } \text{mac} = b:\text{bytes}\{\text{Length}(b) = \text{macsize}\}
\]

\text{predicate val } \text{Msg: key } \times \text{text } \rightarrow \text{bool}
\text{val GEN: unit } \rightarrow \text{key}
\text{val MAC: k: key } \rightarrow \text{t: text}\{\text{Msg}(k,t)\} \rightarrow \text{mac}
\text{val VERIFY: k: key } \rightarrow \text{t: text } \rightarrow \text{m: mac } \rightarrow \text{v: bool}\{\text{v=true } \Rightarrow \text{Msg}(k,t)\}
\text{val LEAK: k: key}\{\text{!t. Msg}(k,t)\} \rightarrow b:\text{bytes}\{\text{Length}(b) = \text{keysize}\}

Too good to be true!
If (i) MACs are shorter than text and
(ii) MAC and VERIFY communicate only through MACs
any implementation must be unsafe (information theory)
MACs: interfaces and implementations

a plain F# interface

... and its refinements

Concrete Mac

Ideal Mac

RPC

cannot typecheck in F7!

some concrete implementation

LINK

some sample protocol
Refined concrete interface ($I^C$)

```ocaml
type mac = b:bytes\{\text{Length}(b) = \text{macsize}\}
predicate val GEnerrated: key → bool
predicate val MACed: key * text * mac → bool
val GEN: unit → k:key \{GEnerrated(k)\}
val MAC: k:key → t:text → m:mac \{MACed(k,t,m)\}
val VERIFY: k:key → t:text → m:mac →
  v:bool\{\ GEnerrated(k) \land MACed(k,t,m) \Rightarrow v = \text{true} \ \}
val LEAK: k:key → b:bytes\{\text{Length}(b) = \text{keysize}\}
assume !k,t,m0,m1.
  GEnerrated(k) \land MACed(k,t,m0) \land MACed(k,t,m1) \Rightarrow m0 = m1
```

- Captures the functional correctness assumptions
CMA module

```
let k = GEN()
let log = ref []
let mac t = log := t::!log; MAC k t
let verify t m =
    let v = VERIFY k t m in assert(not v || List.mem t !log); v
```

• Used to code the game capturing Resistance against existential Chosen Message forgery Attacks (CMA)

• The adversary may adaptively call the two oracles `mac` and `verify`: the adversary wins if it can forge a valid `mac`: the `assert` claims that this does not happen
CMA module

let $k = GEN()$
let log = ref []
let mac $t = log := t::!log; MAC k t$
let verify $t m =$
  let $v = VERIFY k t m$ in assert(not $v \parallel List.mem t !log); v$

• Used to code the game capturing *Resistance against existential Chosen Message forgery Attacks* (CMA)

• The adversary may adaptively call the two oracles $mac$ and $verify$: the adversary wins if it can forge a valid $mac$; the assert claims that this does not happen

**Definition 1**: $C$ is CMA-secure when, for all p.p.t. expressions $A$ with no `assume` or `assert` such that $mac: \text{text} \rightarrow \text{mac}$, $verify: \text{text} \rightarrow \text{mac} \rightarrow \text{bool} \vdash A: \text{unit}$, the expression $C \cdot \text{CMA} \cdot A$ is asymptotically safe.
Asymptotic safety for MAC

Theorem 2 (Asymptotic Safety for MAC). Let $C$ be p.p.t. CMA-secure and such that $\vdash C \sim I^C$. Let $A$ be p.p.t. such that $I \vdash A : \text{unit}$. The expression $C \cdot A$ is asymptotically safe.

- Ok, but how do we prove it? We still have the same problem: we type the code and the attacker against the ideal interface but we use an implementation that exports $I^C$.
MACs: interfaces and implementations

- a plain F# interface
- ... and its refinements

- Concrete Mac
- Mac
- Ideal Mac
- Ideal Mac
- RPC

- Mac
- CMA
- some concrete implementation
- some error correcting wrapper
- some sample protocol

- LINK
- LINK

RPC
MACs: proof idea

Concrete Mac  Ideal Mac  RPC  Adversary

is always safe (by typing)

Concrete Mac  RPC  Adversary

is indistinguishable from

is safe too, with overwhelming probability
We introduce an ideal functionality that "corrects" any unsafe verification of $C$.

```ocaml
let ks = ref [] (* state for all keys generated so far *)
let GEN () =
  let k = list_length !ks
  let kv = Mac.GEN() in
  ks := (k,(kv,empty_log k,false_flag k))::!ks; k
let MAC k t =
  let (kv,log,leaked) = assoc k !ks in
  log := t:: !log; Mac.MAC kv t
let VERIFY k t m =
  let (kv,log,leaked) = assoc k !ks in
  Mac.VERIFY kv t m && (mem k t !log || !leaked)
let LEAK k =
  let (kv,log,leaked) = assoc k !ks in
  leaked := true; Mac.LEAK kv
```
We introduce an ideal functionality that “corrects” any unsafe verification of $C$

```ocaml
let ks = ref [] (* state for all keys generated so far *)
let GEN () =
  let k = list_length !ks
  let kv = Mac.GEN() in
  ks := (k,(kv,empty_log k,false_flag k))::!ks; k
let MAC k t =
  let (kv,log,leaked) = assoc k !ks in
  log := t:: !log; Mac.MAC kv t
let VERIFY k t m =
  let (kv,log,leaked) = assoc k !ks in
  Mac.VERIFY kv t m && (mem k t !log || !leaked)
let LEAK k =
  let (kv,log,leaked) = assoc k !ks in
  leaked := true; Mac.LEAK kv
```
Ideal Functionality ($F$)

- We introduce an ideal functionality that “corrects” any unsafe verification of $C$

```ocaml
let ks = ref [] (* state for all keys generated so far *)
let GEN () =
  let k = list_length !ks
  let kv = Mac.GEN() in
  ks := (k,(kv,empty_log k,false_flag k))::!ks; k
let MAC k t =
  let (kv,log,leaked) = assoc k !ks in
  log := t:: !log; Mac.MAC kv t
let VERIFY k t m =
  let (kv,log,leaked) = assoc k !ks in
  Mac.VERIFY kv t m && (mem k t !log || !leaked)
let LEAK k =
  let (kv,log,leaked) = assoc k !ks in
  leaked := true; Mac.LEAK kv
```

- Keys represented by key indexes
- Corrects unsafe verifications
Asymptotic safety for MAC

**Lemma 1** (Typing). \( I^C \vdash F \leadsto I \).

**Theorem 3** (Ideal Functionality for MAC). Let \( C \) be p.p.t. CMA-secure such that \( \vdash C \leadsto I^C \). Let \( A \) be p.p.t. such that \( I \vdash A \). We have \( C \cdot A \approx_\varepsilon C \cdot F \cdot A \).

**Theorem 2** (Asymptotic Safety for MAC). Let \( C \) be p.p.t. CMA-secure and such that \( \vdash C \leadsto I^C \). Let \( A \) be p.p.t. such that \( I \vdash A : \text{unit} \). The expression \( C \cdot A \) is asymptotically safe.
Asymptotic safety for MAC

**Lemma 1 (Typing).** $I^C \vdash F \sim I$.

**Theorem 3 (Ideal Functionality for MAC).** Let $C$ be p.p.t. CMA-secure such that $\vdash C \sim I^C$. Let $A$ be p.p.t. such that $I \vdash A$. We have $C \cdot A \approx I C F A$.

**Theorem 2 (Asymptotic Safety for MAC).** Let $C$ be p.p.t. CMA-secure and such that $\vdash C \sim I^C$. Let $A$ be p.p.t. such that $I \vdash A : unit$. The expression $C \cdot A$ is asymptotically safe.

We type-check $C \cdot F \cdot A$ and get security for $C \cdot A$.
Computational Verification: Indistinguishability and encryption

Secrecy by typing
Modules for secret plain texts
Ideal functionalities for CPA and CCA2 encryption
Secrecy by Typing

- There exist two different ways for formalising the notion of (symbolic) secrecy
  - syntactic secrecy (the adversary cannot read a certain secret)
  - observational equivalence-based secrecy (the adversary cannot distinguish which between two secrets is used)
- The latter is the symbolic counterpart of the concept of indistinguishability used in cryptography
Syntactic secrecy

• Can be easily defined and enforced in F7

Robust Secrecy:
Let $A$ be an expression with free variable $s$. The expression $A$ preserves the secrecy of $s$ unless $C$ iff the expression $\text{let } s = (\text{fun } x \rightarrow \text{assert } C) \text{ in } A$ is robustly safe.

\textbf{Theorem 3 (Robust Secrecy).} If $s : \{C\} \rightarrow \text{unit} \vdash A : \text{Un}$, then $A$ preserves the secrecy of $s$ unless $C$. 
Observational equivalence-based secrecy

- Secrecy is expressed as observational equivalences between systems that differ on their secrets
  - two expressions are equivalent when they return values with the same probabilities

- We can prove secrecy by typing, relying on parametricity over some abstract type
Secrecy by typing

**Definition 2.** For a fixed type variable $\alpha$, let secret types be of the form $T_\alpha ::= \alpha | T \to T_\alpha$ where $T$ ranges over base types. Let secret interfaces be of the form $I_\alpha = \alpha, x_1 : T_{\alpha,1}, \ldots, x_n : T_{\alpha,n}$ for some $n \geq 0$. Let $P_\alpha$ range over modules that define **let** $x_i = v_i$ for some pure total values $v_i$ for $i = 1..n$, such that $\vdash P_\alpha \triangleright I_\alpha$.

**Theorem 5 (Secrecy by Typing).** Let $A$ such that $I_\alpha \vdash A : bool$. For all $P^0_\alpha$ and $P^1_\alpha$, we have $P^0_\alpha \cdot A \approx P^1_\alpha \cdot A$.

\[
I_\alpha \vdash C \triangleright I_C \\
I_C \vdash A : bool
\]

imply

\[
P^0_\alpha \cdot C \cdot A \approx P^1_\alpha \cdot C \cdot A
\]
Secrecy by typing

**Definition 2.** For a fixed type variable $\alpha$, let secret types be of the form $T_\alpha ::= \alpha \mid T \to T_\alpha$ where $T$ ranges over base types. Let secret interfaces be of the form $I_\alpha = \alpha, x_1 : T_{\alpha,1}, \ldots, x_n : T_{\alpha,n}$ for some $n \geq 0$. Let $P_\alpha$ range over modules that define **let** $x_i = v_i$ for some pure total values $v_i$ for $i = 1..n$, such that $\vdash P_\alpha \rightsquigarrow I_\alpha$.

**Theorem 5 (Secrecy by Typing).** Let $A$ such that $I_\alpha \vdash A : bool$. For all $P_\alpha^0$ and $P_\alpha^1$, we have $P_\alpha^0 \cdot A \approx P_\alpha^1 \cdot A$.

$\vdash I_\alpha \rightsquigarrow I_C$  
$\vdash I_C \rightsquigarrow I_A$  
imply  $P_\alpha^0 \cdot C \cdot A \approx P_\alpha^1 \cdot C \cdot A$
Secrecy by typing

**Definition 2.** For a fixed type variable \( \alpha \), let secret types be of the form \( T_\alpha ::= \alpha \mid T \to T_\alpha \) where \( T \) ranges over base types. Let secret interfaces be of the form \( I_\alpha = \alpha, x_1 : T_{\alpha,1}, \ldots, x_n : T_{\alpha,n} \) for some \( n \geq 0 \). Let \( P_\alpha \) range over modules that define \textbf{let} \( x_i = v_i \) for some pure total values \( v_i \) for \( i = 1..n \), such that \( \vdash P_\alpha \rightsquigarrow I_\alpha \).

**Theorem 5 (Secrecy by Typing).** Let \( A \) such that \( I_\alpha \vdash A : bool \). For all \( P_\alpha^0 \) and \( P_\alpha^1 \), we have \( P_\alpha^0 \cdot A \approx P_\alpha^1 \cdot A \).

\[
I_\alpha \vdash C \rightsquigarrow I_C \\
I_C \vdash A : bool
\]
imply
\[
P_\alpha^0 \cdot C \cdot A \approx P_\alpha^1 \cdot C \cdot A
\]
DEFINITION 2. For a fixed type variable $\alpha$, let secret types be of the form $T_\alpha ::= \alpha \mid T \rightarrow T_\alpha$ where $T$ ranges over base types. Let secret interfaces be of the form $I_\alpha = \alpha, x_1 : T_{\alpha,1}, \ldots, x_n : T_{\alpha,n}$ for some $n \geq 0$. Let $P_\alpha$ range over modules that define $\text{let } x_i = v_i$ for some pure total values $v_i$ for $i = 1..n$, such that $\vdash P_\alpha \leadsto I_\alpha$.

THEOREM 5 (Secrecy by Typing). Let $A$ such that $I_\alpha \vdash A : \text{bool}$. For all $P^0_\alpha$ and $P^1_\alpha$, we have $P^0_\alpha \cdot A \approx P^1_\alpha \cdot A$.

$\vdash I_\alpha \mid C \leadsto I_C$ imply $P^0_\alpha \cdot C \cdot A \approx P^1_\alpha \cdot C \cdot A$.
Concrete ($I^C_{\text{PLAIN}}$) and ideal ($I_{\text{PLAIN}}$) interfaces for Plain

- $I_{\text{PLAIN}}$ is the same, except for plain that is abstract
- Obviously $\vdash P \leadsto I^C_{\text{PLAIN}}$ implies $\vdash P \leadsto I_{\text{PLAIN}}$

```latex
val plainSize: int
type repr = b:bytes \{ Length(b) = plainSize \}
type plain = repr
val plain: repr \rightarrow plain
val respond: plain \rightarrow plain
```
Refined concrete interface \( I_{ENC}^C \)

val pksize: int
val ciphersize: int
type pkey = b:bytes \{ \text{Length}(b) = pksize \}
type cipher = b:bytes \{ \text{Length}(b) = ciphersize \}
type skey

\textbf{predicate} val GENerated: pkey \ast skey \rightarrow \text{bool}
\textbf{predicate} val ENCrypted: pkey \ast \text{repr} \ast cipher \rightarrow \text{bool}
val GEN: unit \rightarrow pk:pkey \ast sk:skey \{ \text{GENerated}(pk,sk) \}
val ENC: pk:pkey \rightarrow p:repr \rightarrow c:cipher \{ \text{ENCrypted}(pk,p,c) \}
val DEC: sk:skey \rightarrow c:cipher \rightarrow p:repr
   \{ \forall pk,p'. \text{GENerated}(pk,sk) \land \text{ENCrypted}(pk,p',c) \Rightarrow p = p' \}

• Captures functional correctness assumptions
Refined concrete interface \( I_{\text{ENC}}^C \)

\begin{verbatim}
val pksize: int
val ciphersize: int
type pkey = b:bytes{Length(b)=pksize}
type cipher = b:bytes {Length(b)=ciphersize}
type skey
predicate val GENerated: pkey * skey -> bool
predicate val ENCrypted: pkey * repr * cipher -> bool
val GEN: unit -> pk:pkey * sk:skey {GENerated(pk,sk)}
val ENC: pk:pkey -> repr -> cipher {ENCrypted(pk,repr,c)}
val DEC: sk:skey -> cipher -> repr
    {!pk,p'. GENerated(pk,sk) ∧ ENCrypted(pk,p',c) ⇒ p = p'}
\end{verbatim}

• Captures functional correctness assumptions

The plaintext types cannot be abstract, since plain texts have at least to be marshalled, their length to be checked, etc.
Ideal interface ($I_{ENC}$) for public-key encryption

- Secret keys are abstract, while public keys are concrete and can be passed to the attacker
- This interface is used for typing

```plaintext
predicate val PKey: bytes → bool
type pkey = pk:bytes{Length(pk)=pksize ∧ PKey(pk)}
type cipher = b:bytes {Length(b)=ciphersize}
type skey
val GEN: unit → pkey * skey
val ENC: pkey → plain → cipher
val DEC: skey → cipher → plain
```
CCA2 security

\[ I_{cca} \triangleq pk: \text{pkey}, \, enc: \text{plain} \rightarrow \text{cipher}, \, dec: \text{cipher} \rightarrow \text{plain} \]

let \( pk, \, sk = \text{GEN()} \)
let \( \text{log} = \text{empty\_log}\, pk \)
let \( \text{enc}\, x_0\, x_1 = \)
  let \( \text{x} = \text{select}\, x_0\, x_1\, \text{in} \)
  let \( \text{v} = \text{ENC}\, pk\, x\, \text{in} \)
  \( \text{log} := \text{cons}\, pk\, (v, x)\, !\text{log};\, v \)

let \( \text{dec}\, v = \)
  \( \text{match}\, \text{assoc}\, pk\, v\, !\text{log}\, \text{with} \)
  \( | \text{Some}(x) \rightarrow \text{zero} \)
  \( | \text{None} \rightarrow \text{DEC}\, sk\, v \)

- Indistinguishability under chosen-ciphertext attacks: the adversary is given access to \( pk, \, enc, \, dec \) and has to guess \( b \)

**Definition 3.** \( C_{\text{ENC}} \) is CCA2 secure when, for all p.p.t. modules \( P \) with \( P \sim I_{PLAIN}^C \) and \( A \) with \( I_{PLAIN}^C, I_{cca} \vdash A : \text{bool} \), we have \( P \cdot C_{\text{ENC}} \cdot CCA^0 \cdot A \approx_{\varepsilon} P \cdot C_{\text{ENC}} \cdot CCA^1 \cdot A. \)
CCA2 security

\[ I_{\text{cca}} \triangleq \begin{aligned} & \text{pk: pkey, enc: plain } \rightarrow \text{cipher, dec: cipher } \rightarrow \text{plain} \\ & \text{let } pk, sk = \text{GEN} () \\ & \text{let } log = \text{empty\_log} \text{ pk} \\ & \text{let } \text{enc} \ x_0 \ x_1 = \\ & \quad \text{let } x = \text{select} \ x_0 \ x_1 \text{ in} \\ & \quad \text{let } v = \text{ENC} \ pk \ x \text{ in} \\ & \quad \text{log} := \text{cons} \ pk \ (v,x) \ \text{!log;} \ v \\ & \text{let } \text{dec} \ v = \\ & \quad \text{match} \ \text{assoc} \ \text{pk} \ v \ \text{!log with} \\ & \quad \text{| Some}(x) \ \rightarrow \text{zero} \\ & \quad \text{| None } \rightarrow \text{DEC sk} \ v \\ \end{aligned} \]

- Indistinguishability under chosen-ciphertext attacks: the adversary is given access to \( pk, \ enc, \ dec \) and has to guess \( b \)

**Definition** 3. \( C_{\text{ENC}} \) is CCA2 secure when, for all p.p.t. modules \( P \) with \( \vdash P \rightleftharpoons I_{\text{PLAIN}}^C \) and \( A \) with \( I_{\text{PLAIN}}^C, I_{\text{cca}} \vdash A : \text{bool} \), we have \( P \cdot C_{\text{ENC}} \cdot CCA^0 \cdot A \approx_\varepsilon P \cdot C_{\text{ENC}} \cdot CCA^1 \cdot A. \)
Asymptotic secrecy

**Theorem 6 (Asymptotic Secrecy).** Let $C_{\text{ENC}}$ be a p.p.t. CCA2-secure module with $I^C_{\text{PLAIN}} \vdash C_{\text{ENC}} \sim I^C_{\text{ENC}}$. Let $A$ be a p.p.t. expression with $I_{\text{PLAIN}}, I_{\text{ENC}} \vdash A : \text{bool}$.

For any two pure p.p.t. implementations $P^b$ of $I_{\text{PLAIN}}$, we have $P^0 \cdot C_{\text{ENC}} \cdot A \approx_\varepsilon P^1 \cdot C_{\text{ENC}} \cdot A$.

• Nice, but how do we prove it? We are typing against $I_{\text{ENC}}$ but security is defined for $I^C_{\text{ENC}}$
Ideal functionality ($F_{ENC}$)

let $ks = \text{ref} \ [\] \ (* \ state \ for \ all \ keypairs \ generated \ so \ far \ *)$

let $zero = \text{zeroCreate plainsize}$

let $GEN () =$
  let $pk, sk = \text{PKEnc.GEN} ()$
  $ks := (pk,(sk, empty\_log \ pk))::!ks; (pk,SK(pk))$

let $ENC \ pk \ x =$
  let $(sk,log) = \text{assoc} \ pk \ !ks \ \text{in}$
  let $c = \text{PKEnc.ENC} \ pk \ \text{zero} \ \text{in} \ \log := \text{cons} \ pk \ (c,x) \ !log; c$

let $DEC \ (SK(pk)) \ c =$
  let $(sk,log) = \text{assoc} \ pk \ !ks \ \text{in}$
  match $\text{assocc} \ pk \ c \ !log$ with
  | $\text{Some}(x) \rightarrow x \ | \ \text{None} \rightarrow \text{plain} \ (\text{PKEnc.DEC} \ sk \ c)$$

$I_{\text{PLAIN}}, I_{\text{ENC}}^C \vdash F_{\text{ENC}} \leadsto I_{\text{ENC}}$

**THEOREM 7 (IDEAL FUNCTIONALITY).** Let $C_{ENC}$ be a p.p.t.
CCA2-secure module with $I_{\text{PLAIN}}^C \vdash C_{ENC} \leadsto I_{\text{ENC}}^C$, $P$ a p.p.t. mod-
ule with $\vdash P \leadsto I_{\text{PLAIN}}^C$, and $A$ a p.p.t. expression with $I_{\text{PLAIN}}^C, I_{ENC} \vdash A$. We have $P \cdot C_{ENC} \cdot A \approx_{\varepsilon} P \cdot C_{ENC} \cdot F_{ENC} \cdot A$. 
Ideal functionality ($F_{\text{ENC}}$)

let $ks = \text{ref} \; []$ (* state for all keypairs generated so far *)

let $zero = \text{zeroCreate plainsize}$

let $GEN () =$
  let $pk, \; sk = \text{PKEnc.GEN} ()$
  ks := (pk,(sk, empty_log pk)) ::! ks; (pk,SK(pk))

let $ENC \; pk \; x =$
  let $(sk,log) = \text{assoc} \; pk \; !ks \; \text{in}$
  let $c = \text{PKEnc.ENC} \; pk \; \text{zero} \; \text{in}$
  log := cons $pk \; (c,x) \; \!\log ; \; c$

let $DEC \; (SK(pk)) \; c =$
  let $(sk,log) = \text{assoc} \; pk \; !ks \; \text{in}$
  match assocc $pk \; c \; \!\log$ with
  $\mid \text{Some}(x) \rightarrow x \mid \text{None} \rightarrow \text{plain} (\text{PKEnc.DEC} \; sk \; c)$

$I_{\text{PLAIN}}, I_{\text{ENC}}^C \vdash F_{\text{ENC}} \sim \rightarrow I_{\text{ENC}}$

**Theorem 7 (Ideal Functionality).** Let $C_{\text{ENC}}$ be a p.p.t. CCA2-secure module with $I_{\text{PLAIN}}^C \vdash C_{\text{ENC}} \sim \rightarrow I_{\text{ENC}}^C$, $P$ a p.p.t. module with $P \sim \rightarrow I_{\text{PLAIN}}^C$, and $A$ a p.p.t. expression with $I_{\text{PLAIN}}^C, I_{\text{ENC}} \vdash A$. We have $P \cdot C_{\text{ENC}} \cdot A \approx_{\varepsilon} P \cdot C_{\text{ENC}} \cdot F_{\text{ENC}} \cdot A$. 

We always encrypt zero
Other primitives

• CPA Secure encryption (removed the decryption oracle)

\[
\text{predicate val } \text{ENCrypted: key } \times \text{ plain } \times \text{ cipher } \rightarrow \text{bool}
\]
\[
\text{val } \text{ENC: key } \rightarrow \text{ plain } \rightarrow \text{ cipher } \{ \text{ENCrypted}(k,p,c) \}
\]
\[
\text{val } \text{DEC: key } \rightarrow \text{ cipher } \{ \exists p. \text{ENCrypted}(k,p,c) \} \rightarrow \text{ plain}
\]

• Authenticated encryption

\[
\text{predicate val } \text{Msg: key } \times \text{ plain } \rightarrow \text{bool}
\]
\[
\text{val } \text{ENC: key } \rightarrow \text{ plain } \rightarrow \text{ cipher } \{ \text{Msg}(k,p) \} \rightarrow \text{ cipher}
\]
\[
\text{val } \text{DEC: key } \rightarrow \text{ cipher } \rightarrow (\text{ plain } \rightarrow \{ \text{Msg}(k,p) \}) \text{ option}
\]

• Hybrid encryption, encrypt-then-mac, …