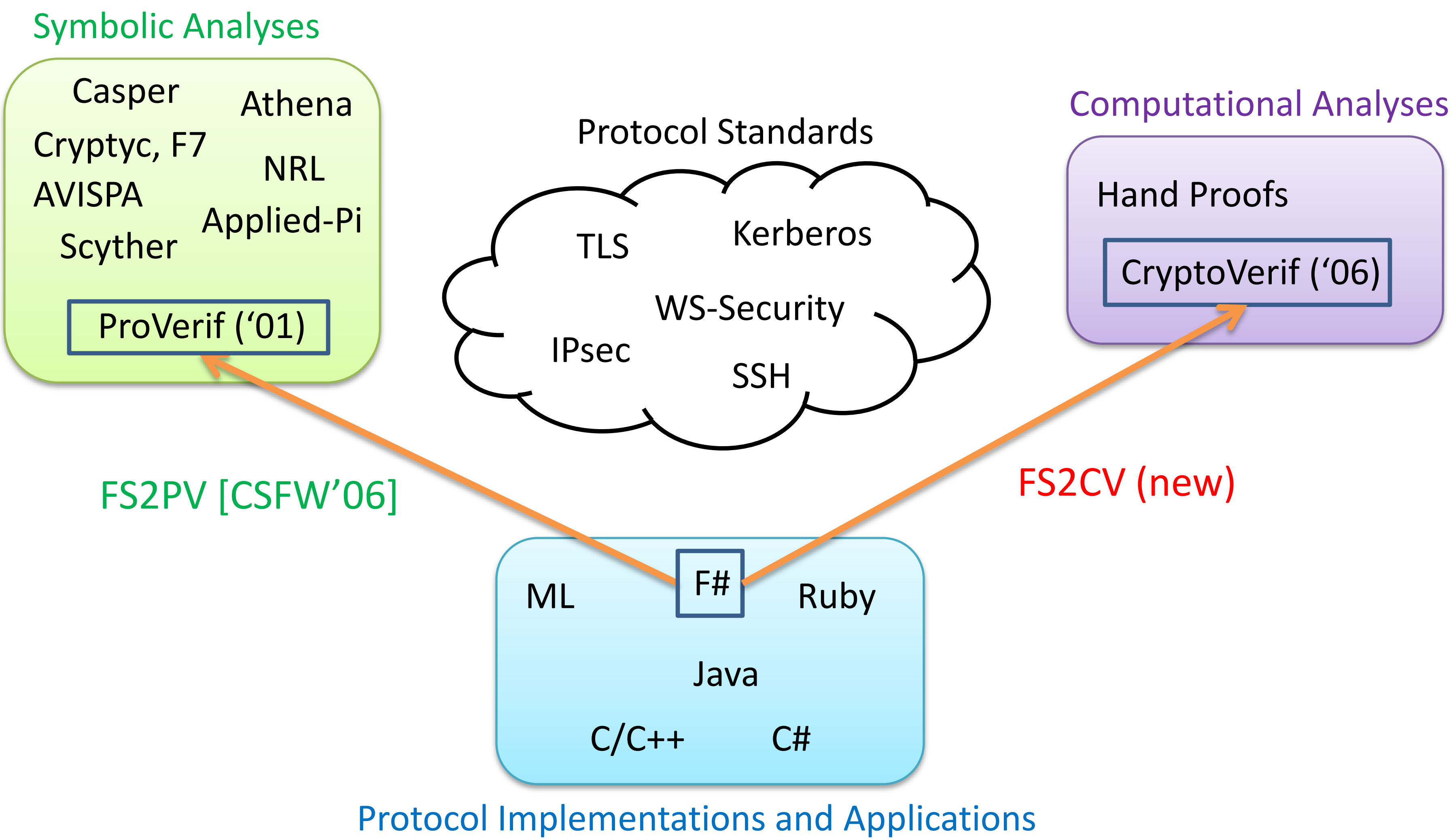
Cryptographic Verification of Protocol Implementations

Karthik Bhargavan, Ricardo Corin,
Cédric Fournet, *Eugen Zalinescu*

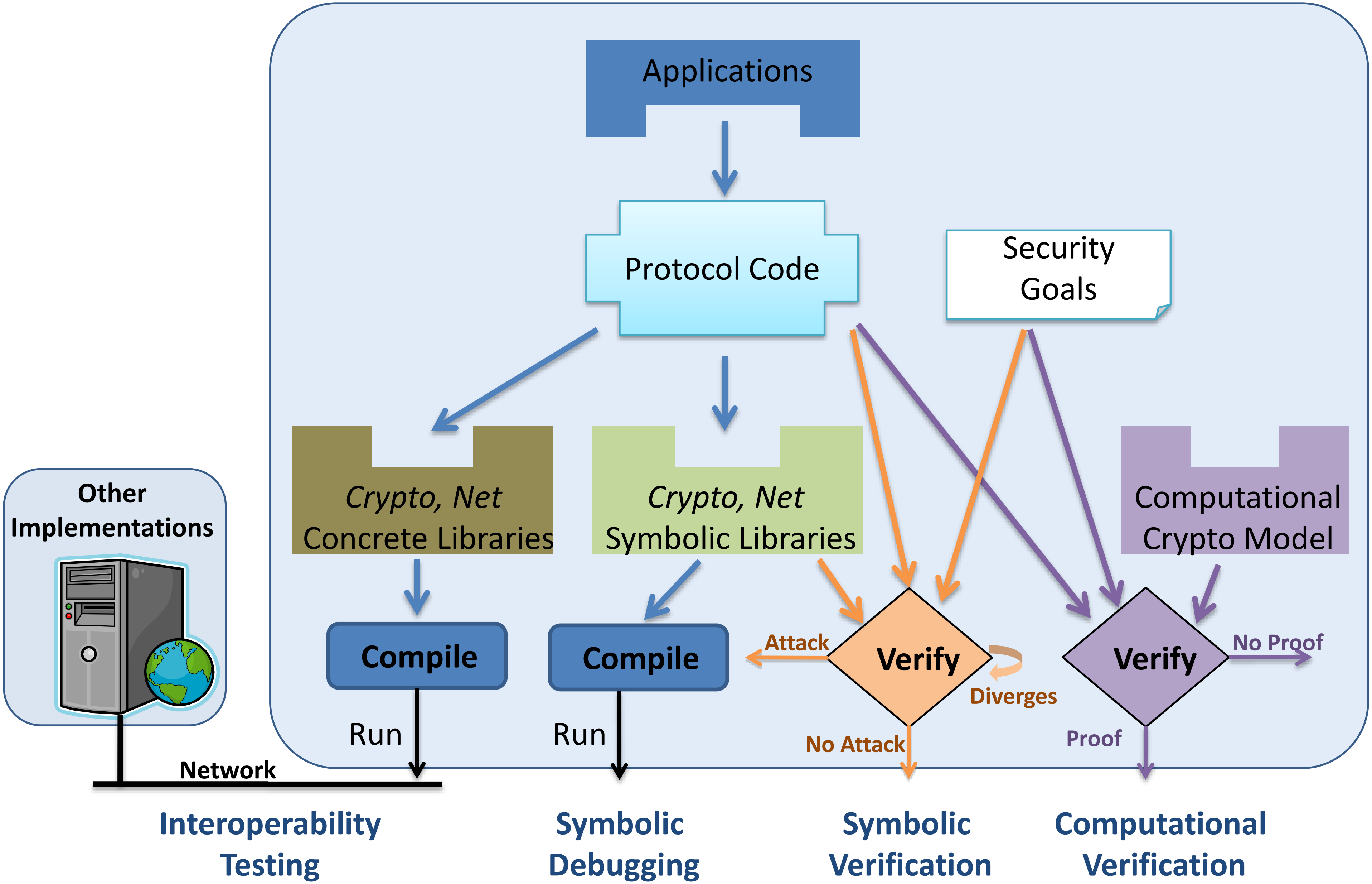
Microsoft Research-INRIA Joint Centre

CoSyPoofs Spring School
Atagawa Heights, 6 April 2009

Protocols and Analyses



Verifying Protocol Implementations



Computational Verification Method

We use Blanchet's CryptoVerif tool [S&P'06] to search for computational proofs using the game-hopping technique [Bellare Rogaway]

1. Manually code crypto assumptions (not in F#)
   - Must define types and assumptions for all cryptographic primitives used in the protocol (HMAC, AES, RSA,…) using probabilistic equivalences encoding indistinguishability
   - Crypto assumptions change rarely
2. Develop FS2CV, a new tool compiling F# code to CryptoVerif scripts
   - Networking and sampling functions translate to CryptoVerif primitives
   - Public functions translate to polynomially replicated processes
3. Run CryptoVerif on *generated script + crypto assumptions + security goals* to computationally verify these goals against PPT adversaries

From protocol code to CryptoVerif

The FS2CV compiler

- applies a series of code transformations
  - inlining of non-recursive functions
  - partial evaluation of functions and patterns
  - dead-code elimination
- converts all public functions to processes
- normalizes the result to fit in a restricted ML syntax (very close to CryptoVerif syntax)
- generates the CryptoVerif script, inlining
  - the protocol security goals
  - the abstract models for the core libraries

Models for core libraries

Core libraries form part of our trusted computing base.

We abstractly represent their properties through

- special source-level encodings
- computational assumptions in CryptoVerif

- for Net and Db library functions
  - we define *encodings* in terms of concurrency and communication constructs in our source language
- for other library functions
  - we treate them as *uninterpreted* functions (deterministic, polytime functions with no side effects)
  - for Crypto primitives we provide additional *security assumptions* as CryptoVerif equations, inequations, and equivalences

Computational Crypto Model

Interface

```
(* byte arrays *)
type bytes

(* symmetric keys *)
type symkey
…
(* generate fresh key *)
val mkKey: nonce -> symkey

(* symmetric encryption *)
val aesEncrypt:
  symkey -> bytes -> bytes

val aesDecrypt:
  symkey -> bytes -> bytes
```

Computational Model (CryptoVerif syntax)

```
type bytes = blocksize
type symkey [fixed].
type keyseed [large,fixed].

fun mkKey(keyseed):key.
fun aesEncrypt(symkey, blocksize): blocksize.
fun aesDecrypt(key, blocksize): blocksize.

forall m:blocksize, r:keyseed;
  aes_decrypt( mkKey(r),
    aes_encrypt(mkKey(r), m)) = m.

equiv
!N new r: keyseed;
 ((x:blocksize) N' -> aes_encrypt(mkKey(r),x),
  (m:blocksize) N' -> aes_decrypt(mkKey(r),m))
<= (N * Psymenc(time, N', Nsymdec)) =>  …
```

Syntax of source language

a, b, c	names
x, y, z	variables
f	constructor
h	uninterpreted function
D	domain

$M, N, K, F, Q, R ::=$	value
a, b, c	name
x	variable
$f(M_1, \ldots, M_n)$	constructor application
$\textbf{fun}\, x_1 \ldots x_n \rightarrow e$	function
$T ::=$	value pattern
a	name matching
$`x$	variable matching
x	variable binding
$f(T_1, \ldots, T_n)$	constructor matching

$e ::=$	expression
M	value
$M\ N_1\ \ldots\ N_n$	function application
$h\ M_1\ \ldots\ M_n$	uninterpreted function application
$\textbf{let}\ x = e\ \textbf{in}\ e'$	variable binding
$\textbf{match}\ M\ \textbf{with}\ T \rightarrow e\ \textbf{else}\ e'$	pattern match
$\textbf{rand}_D()$	random value sampling
$\textbf{log}\ M$	event logging
$\textbf{secret}_D\ M$	secrecy requirement
$(\nu c)e$	restriction (scope of c is e)
$e \upharpoonright e'$	fork
$M!N$	send message N on channel M
$M?T \rightarrow e$	receive message matching T off channel M; then continue with e
$*M?T \rightarrow e$	replicated receive on channel M

Small-step labeled reduction relation

VALUE
$$M \rightarrow_1 M$$

APPLY
$$(\mathbf{fun}\,\widetilde{x} \rightarrow e)\ \widetilde{M} \rightarrow_1 e[\widetilde{x} \mapsto \widetilde{M}]$$

EXTERNAL
$$\frac{(h\ \widetilde{M}) \Downarrow N}{h\ \widetilde{M} \rightarrow_1 N}$$

LET VAL
$$\mathbf{let}\ x = M\ \mathbf{in}\ e \rightarrow_1 e[x \mapsto M]$$

MATCH
$$\mathbf{match}\ T\theta\ \mathbf{with}\ T \rightarrow e\ \mathbf{else}\ e' \rightarrow_1 e\theta$$

MISMATCH
$$\frac{\forall \theta. M \neq T\theta}{\mathbf{match}\ M\ \mathbf{with}\ T \rightarrow e\ \mathbf{else}\ e' \rightarrow_1 e'}$$

RAND
$$\frac{a \in D}{\mathbf{rand}_D() \rightarrow_{\frac{1}{|D|}} a}$$

LET CTX
$$\frac{e \xrightarrow{\alpha}_p e' \qquad e \text{ not a value}}{\mathbf{let}\ x = e\ \mathbf{in}\ e'' \xrightarrow{\alpha}_p \mathbf{let}\ x = e'\ \mathbf{in}\ e''}$$

LOG
$$\mathbf{log}\ M \xrightarrow{M}_1 ()$$

SECRET
$$\mathbf{secret}_D\ M \rightarrow_1 ()$$

COMM
$$(c?T \rightarrow e) \upharpoonleft c!T\theta \xrightarrow{\overline{c}\langle T\theta\rangle}_p e\theta$$

COMM REPL
$$(*c?T \rightarrow e) \upharpoonleft c!T\theta \xrightarrow{\overline{c}\langle T\theta\rangle}_p (*c?T \rightarrow e) \upharpoonleft e\theta$$

NEW CTX
$$\frac{e \xrightarrow{\alpha}_p e' \qquad c \notin fn(\alpha)}{(\nu c)e \xrightarrow{\alpha}_p (\nu c)e'}$$

PAR CTX
$$\frac{e \xrightarrow{\alpha}_p e' \qquad e \text{ not a value}}{e \upharpoonleft e'' \xrightarrow{\alpha}_p e' \upharpoonleft e'';\ \ e'' \upharpoonleft e \xrightarrow{\alpha}_p e'' \upharpoonleft e'}$$

STRUCT
$$\frac{e \equiv e_1 \qquad e_1 \xrightarrow{\alpha}_p e_1' \qquad e_1' \equiv e'}{e \xrightarrow{\alpha}_p e'}$$

A Low-Level Abstract Machine

- A *restricted syntax* (close to CryptoVerif ’s), defining
  - input (waiting) expressions
  - output (active) expressions
- An *abstract machine semantics* (which mimics CryptoVerif’s)
  - defines reductions $\rightsquigarrow$ between runtime configurations

Theorem (*Operational correspondence*) For all output expressions A, values M,

$$\Pr[A \rightarrow^* M] = \Pr[\mathrm{Cfg}(A) \rightsquigarrow^* M]$$

Code transformations

- inlining
- func. app. partial evaluation
- match expr. partial evaluation

$$[\![\mathbf{let}\ x = M\ \mathbf{in}\ e]\!]_1 \triangleq e[x \mapsto M]$$
$$[\![(\mathbf{fun}\ \widetilde{x} \to e)\ \widetilde{M}]\!]_2 \triangleq e[\widetilde{x} \mapsto \widetilde{M}]$$
$$[\![\mathbf{match}\ T\theta\ \mathbf{with}\ T \to e\ \mathbf{else}\ e']\!]_3 \triangleq e\theta$$

Lemma. Transforms 1-3 preserve probabilities of traces.

- other transforms (including functions as processes [Milner'92])

We say that e is *compiled* if transforms don't apply anymore on e.

Theorem. For any expression e_P, any opponent O,
if e'_P is compiled from e_P then

1. e'_P is in the restricted syntax
2. there exists an opponent O' such that for all M

$$\Pr[O[e_P] \to^* M] = \Pr[e'_P | O' \to^* M]$$

Security properties in ML

- *Correspondence properties*
  - defined using CryptoVerif query language
- *Secrecy*
  - defined as the equivalence between two expressions:
    - one outputting the value of the secret
    - one outputting a fresh random value

Theorem. Security theorems proved by CryptoVerif on the compiled scripts apply to the source programs.

Password-based Authentication Protocol (example with compromise and key databases)

A -> B : m,{[m]$_{pwd(A,B)}$}$_{pk(B)}$

```
let pwdGen ab =
  let mkseed = new_mkeyseed () in
  let mk = mkgen(mkseed) in
  insert pwdDb ab (PwdEntry mk)

let leakedPwdGen ab pwd =
  log tr (PwdLeak(ab));
  insert pwdDb ab (LeakedPwdEntry pwd)

let getPwd ab =
  match select pwdDb ab with
   | Some (PwdEntry pwd) -> pwd
   | Some (LeakedPwdEntry pwd )-> pwd
```

```
let client a b m =
  let ab = concat a b in
  let pwd = getPwd ab in
  match select keyDb b with
  | Some (PkEntry (skB,pkB)) ->
      let conn = Net.connect b in
      log tr (Send(ab,text));
      Net.send conn (makeMsg m pkB pwd)

let server a b =
  …
  log tr (Accept text)
```

CryptoVerif query:

```
query m:bitstring,a:bitstring;
  event Accept(a,m) ==> Send(a,m) || PwdLeak(a).
```

Sample generated code

```
let client a b m =
  let ab = concat a b in
  let pwd = getPwd ab in
  match select keyDb b with
  | Some (PkEntry (skB,pkB)) ->
        let conn = connect b in
        log tr (Send(ab,m));
        let p = makeMsg m pkB pwd in
        send conn p


let makeMsg m pk pwd =
  let seed = new_seed () in
  let m' = mac (bs2bl m) pwd in
  let en = enca (m2bl m') pk seed in
  concat en text
```

```
!N in(c, (a:bitstring, b:bitstring, m:bitstring));
    let ab = concat(a,b) in
    let F11 = select(pwdDb,ab) in
    let Some(PwdEntry(pwd8)) = F11 in
    ( let Some(PkEntry(skB,pkB)) = select(keyDb,b) in
          event  Send(ab,m);
          new seed:seed;
          let F13 = bs2bl(m) in
          let m5 = mac(F13,pwd8) in
          let F14 = m2bl(m5) in
          let en7 = enca(F14,pkB,seed) in
          let p:bitstring = concat(en7,m) in
          out(c,p);
          0)
      else
          let Some(LeakedPwdEntry(pwd9)) = F11 in
          …
```

Case Study: TLS

Transport Layer Security Protocol (TLS 1.0)

- Widely-deployed industrial protocol
- Well-understood, with detailed specs
- Good benchmark for analysis techniques

Cryptographically Verified Implementations for TLS *(CCS'08)*:

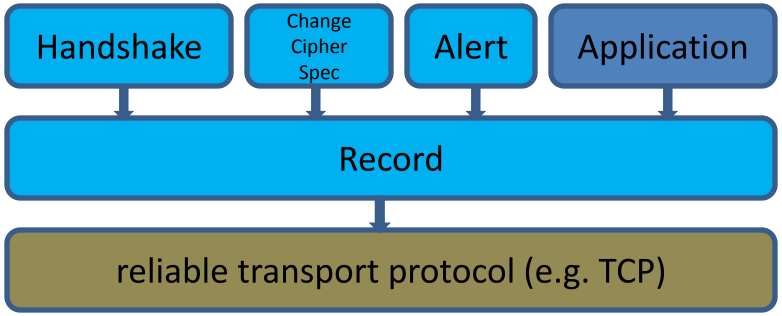
- A reference implementation in F#
- Symbolic verification (of full TLS)
- *Computational verification* (of parts of TLS)

Implementation (10 kLoC)

- a *subset of TLS* (server-only authentication, RSA mode only)
- tested on a few basic scenarios (e.g. *interoperable* HTTPS client & server)

TLS (Transport Layer Security)

- A long history:
  - 1994 – Netscape's Secure Sockets Layer (SSL)
  - 1994 – SSL2 (known attacks)
  - 1995 – SSL3 (fixed them)
  - 1999 – IETF's TLS1.0 (RFC2246, ≈SSL3)
  - 2006 – TLS1.1 (RFC4346)
  - 2008 – TLS1.2 (RFC5246)

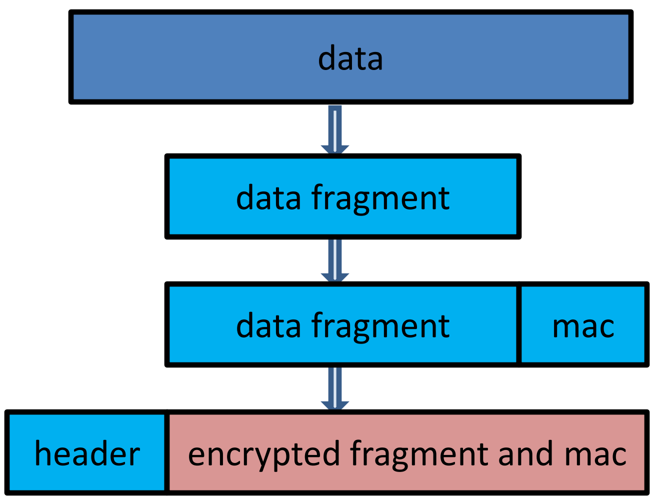


- Two-party protocol between a client and a server
- Provides a layer between TCP and Application (in the TCP/IP model)
  - Itself a layered protocol: Handshake over Record
- *Record* (sub)protocol
  - provides a private and reliable connection
- *Handshake* (sub)protocol
  - authenticates one or both parties, negotiates security parameters
  - establishes secret connection keys for the Record protocol
- *Resumption* (sub)protocol
  - abbreviated version of Handshake: generates connection keys from previous handshake

Record Module

Record protocol (informal narration)

$A \rightarrow B : \{m,[m]_{ak}\}_{ek}$



Record.fs (implementation excerpt)

```
let recv (connid:ConnectionId) =
  let conn = getConnection connid in
  let conn, input = recvRecord conn in
  let conn, msg = verifyPayload conn CT_application_data input in
  let id,entity = connid in log tr (Recv (id,entity,msg));
  storeConnection connid conn;
  msg

let verifyPayload (conn:Connection) (ct:ContentType) (input:bytes) =
  let (bct, bver, blen, ciphertext) = parseRecord input in
  let rct, rver, rlen = getAbstractValues bct bver blen in
  let ver = conn.crt_version in
  if rver = ver then
   let connst = conn.read in
   let connst, plaintext = decrypt ver connst ciphertext  in
   let payload, recvmac = parsePlaintext ver connst plaintext in
   let len = bytes_of_int 2 (length payload) in
   let bseq = bytes_of_seq connst.seq_num in
   let maced = append5 bseq bct bver len payload in
   let conn = updateConnection_read conn connst in
   checkContentType ct rct payload;
   if hmacVerify connst  maced recvmac  = true then
    (conn,payload)
   else failwith "bad record mac"
  else failwith "bad version
```

Retrieve connection

Security event: message accepted!

Message parsing

Decryption supports multiple ciphersuites

MAC verification

Security Properties (Record)

- Verify *Record* in isolation
  - Assume a database of pre-established connections: keys are generated from fresh cr, sr, and ms (using PRF)
- Connection keys may be leaked
- Crypto assumptions:
  - *ROM* for PRF: turns derived keys into random bitstrings
  - *UF-CMA* for HMAC: correlates valid macs with their possible origin(s)
  - *SPRP* (super pseudo-random permutation) for block ciphers (AES/DES): replaces encryptions and decryptions by random bitstrings

Message Authentication

In any polynomial run of the protocol, with overwhelming probability,
if the client receives message *p*, then the server has sent *p* or the connection is corrupted.

Payload Secrecy

In any polynomial run of the protocol, the sequence of *sent payload* values
is *indistinguishable* from a sequence of independent *random values*.

TLS Handshake Protocol

Goals:
- authenticate one or both parties
- reliably negotiate security parameters
- establish secret connection keys (for Record protocol)

Phases	*Client*		*Server*
Negotiation	ClientHello	→	
			ServerHello
Key agreement (RSA, no client auth)		←	Certificate ServerHelloDone
	ClientKeyExchange		
Confirmation	[ChangeCipherSpec] Finished	→	
		←	[ChangeCipherSpec] Finished
Application	Application Data	↔	Application Data

(Protected by Record Protocol)

Security Properties (Handshake secrecy)

- Verify most of the client role of the Handshake protocol
  - We assume pre-established parameters and a public/private keypair
  - The client sends ClientKeyExchange, generates connection keys, and sends Finished

- Crypto assumptions:
  - *IND-CCA2* for asymmetric encryption: indistinguishability against chosen-ciphertext attacks
  - *random oracle* for PRF (key derivation)

Secrecy of PMS Random (recall that pms = ver_max || random)

In any polynomial run of the protocol, the sequence of random values is *indistinguishable* from a sequence of independent fresh values.

Security Properties (Handshake auth.)

- A similar setting as for Handshake secrecy :
  - We assume pre-established pms
  - Parties send and receive Finished messages
  - The messages are sent over the Record layer in NULL mode (enc=mac=identity)

- Crypto assumptions:
  - *UF-CMA* for PRF when used to build the Finished messages

Agreement on MS

In any polynomial run of the protocol, with overwhelming probability, if the client receives the Finished message, then the server has sent it, and they agree on the value of ms.

Symbolic Verification of TLS

Approach

- write F# symbolic implementations for libraries
- declare capabilities for active attackers through library interfaces (in Dolev-Yao style)
- run FS2PV/ProVerif tool chain

Results

- proved secrecy & authentication goals for Handshake & Record (full TLS verification: 3.5h, 4.5GB)
- identified known pitfalls (e.g. version rollback)

Computational ≠ Symbolic

- Some properties hold symbolically but not computationally:
  - Computationally
    - hash functions yield no secrecy guarantees
    - encryption keys are secret only before they are used
  - For the Handshake protocol
    - symbolically, pms is secret,
    - computationally, only random is secret (where pms = ver_max || random)
- Symbolically, we verify the code for the full protocol + applications
  Computationally, we could only verify the code for "core fragments"
  - Our computational tools are still young
  - MAC-then-encrypt is computationally delicate

Experimental results

Protocol	LoC	Query	Nb. of games	Applied equivalences	Verification Time
Authenticated RPC	90	authentication	7	mac	0.2sec
Password-based Authentication	110	secrecy	16	db, db, enca	0.3sec
		authentication	16	db, db, mac	0.5sec
Otway-Rees	200	secrecy (5)	31	db, enc, enc	1m 34sec
		authentication (6)	43	db, enc, enc	2m 15sec
TLS	3000	Record auth.	15	prf, db, hmac	2.5sec
		Record secrecy	14	prf, db, enc	0.8sec
		Handshake auth.	8	prf_hmac	0.9sec
		Handshake secrecy	18	prf, enca	1.8sec

Recent related work

- P. Morrissey, N. Smart, B. Warinschi. A Modular Security Analysis of the TLS Handshake Protocol. *AsiaCrypt'08*.
- S. Gajek, M. Manulis, O. Pereira, A.-R. Sadeghi, J. Schwenk. Universally Composable Security Analysis of TLS. *ProvSec'08.*
- S. Chaki, A. Datta. ASPIER: An Automated Framework for Verifying Security Protocol Implementations. *CSF'08*.

Summary

- first computational verification results for protocol *implementations*
- can use F# as a front-end for CryptoVerif
- strong security for a functional implementation of TLS 1.0 [CCS'08]
  - against realistic active adversaries
  - both symbolically and computationally

Future work

- analyze computationally full TLS, consider more examples
- optimizations for FS2CV, explore automatic code simplifications
- handle production code?

FS2CV project

http://www.msr-inria.inria.fr/projects/sec/fs2cv/

- Cryptographically Verified Implementations for TLS (*CCS'08* paper + slides + long version)

- tls.tgz (TLS symbolic & computational verification)
- fs2cv-examples.tgz (Authenticated RPC, Password-based Auth, Otway-Rees)
  - Comments and bug reports welcomed!

- (very) soon: tech-report on F# prob. semantics & FS2CV correctness

Thank you! Questions?

Source level encodings of Net and Db

Net library

```
val connect: string -> conn
val listen: string -> conn
val sendrecv: conn -> bitstring -> (bitstring -> unit) -> unit
val recvsend: conn -> (bitstring -> bitstring) -> unit
```

$\text{chan}\ () \triangleq (\nu a)a$

$\text{sendrecv}\ (Q,R)\ M\ (\textbf{fun}\ T \to e) \triangleq (R?T \to e) \upharpoonright Q!M$

$\text{recvsend}\ (Q,R)\ (\textbf{fun}\ T \to e) \triangleq Q?T \to R!e$

$\text{listen}\ (Q,R)\ (\textbf{fun}\ T \to e) \triangleq (Q?T \to R!e) \upharpoonright ()$

$\text{listenN}\ (Q,R)\ (\textbf{fun}\ T \to e) \triangleq (*Q?T \to R!e) \upharpoonright ()$

Db library

```
val newDb: guid -> db
val insert: db -> bitstring -> bitstring -> unit
val select: db -> bitstring -> bitstring option
```

$\textbf{let}\ \text{newDb} = \textbf{fun}\ () \to (\text{chan}(), \text{chan}())$

$\textbf{let}\ \text{insert} = \textbf{fun}\ (q,r)\ k\ m \to \text{listenN}\ (q,r)\ (\textbf{fun}\ \text{`}k \to (k,m))$

$\textbf{let}\ \text{select} = \textbf{fun}\ (q,r)\ k \to \text{sendrecv}\ (q,r)\ k\ (\textbf{fun}\ (\text{`}k,x) \to x)$

CV equivalence encoding databases

```
fun newdb(guid):db[compos].
fun insert(db,key,value):unit[compos].
fun select(db,key):option[compos].

equiv
  !N  new db: guid; (
      (k:key,v:value) N1 -> insert(newdb(db),k,v),
                (k':key) N2 -> select(newdb(db),k') )

  <=(N * Pdb(N2))=>

  !N  new db: guid;  (
     (k:key,v:value) N1 -> un,
               (k':key) N2 -> find j <= N1 suchthat defined(k[j],v[j]) && k'=k[j]
                              then Some(v[j])
                              else None).
```

SPRP assumption

```
equiv

!N new r: keyseed;
    ((x:blocksize) N1  -> symenc(x, kgen(r)),
     (m:blocksize) N2 ->  symdec(m, kgen(r)))

<= (N * Psymenc(time, N1, N2)) =>

!N new r: keyseed;
    ((x:blocksize) N1 ->
        find j<=N1 suchthat defined(x[j],r2[j]) && otheruses(r2[j]) && x = x[j] then r2[j]
        orfind k<=N2 suchthat defined(r4[k],m[k]) && otheruses (r4[k]) && x = r4[k] then m[k]
        else new r2: blocksize; r2,
     (m:blocksize) N2 ->
        find j<=N1 suchthat defined(x[j],r2[j]) && otheruses(r2[j]) && m = r2[j] then x[j]
        orfind k<=N2 suchthat defined(r4[k],m[k]) && otheruses(r4[k]) && m = m[k] then r4[k]
        else new r4: blocksize; r4).
```

Related Work

Early, informal analyses

- around SSL1, PCT
- Schneier & Wagner "*Analysis of the SSL3.0 protocol*", USENIX'96

Symbolic Verifications

- Mitchell, Schmatikov, Stern "*Finite state analysis of SSL 3.0*", USENIX'98
- Paulson "*Inductive Analysis of the Internet protocol TLS*", ACM TISS, '99.
- Kamil, Lowe "*Analysing TLS in the Strand Spaces Model*", Research Report 2008

Related Work (cont.)

Attacks

- On SSL2 (rollback attacks, same keys for enc&auth)
- On SSL3
  - RSA and CBC padding attacks (Bleichenbacher; Paterson)
  - Timing attacks (Klima, Pokorny, Rosa)

Computational Analyses

- Jonsson, Kaliski, "*On the Security of RSA Encryption in TLS*", CRYPTO'02
- Morrisay, Smart, Warinschi, "*A modular security analysis of SSL/TLS*", AsiaCrypt'08

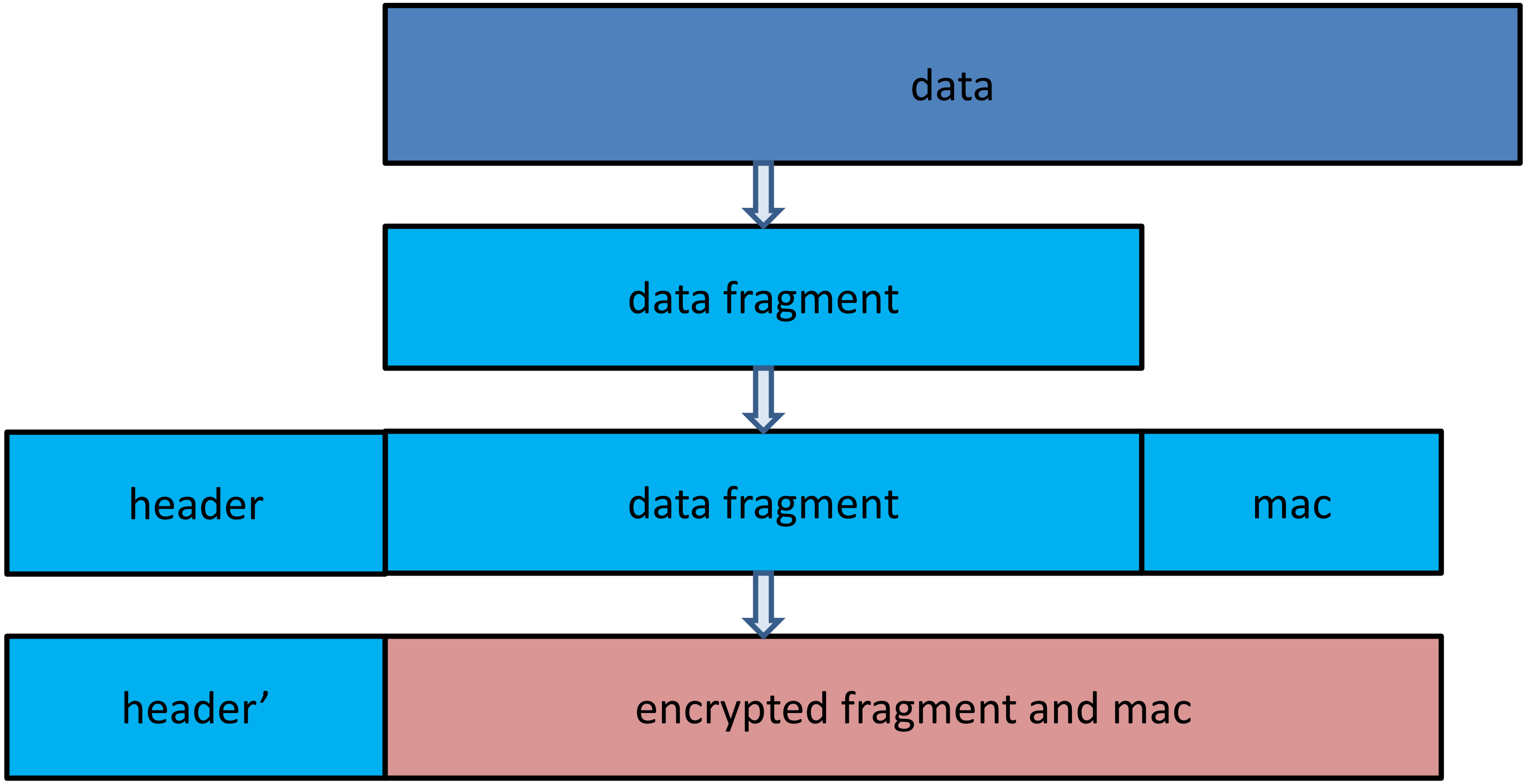
Related Work (cont.)

- verification of security protocol code
  - Goubault-Larrecq & Parrennes "*Cryptographic Protocol Analysis on Real C Code*", VMCAI'05 (on the Needham-Schroeder protocol)
  - Chaki, Datta "*Automated verification of security protocol implementation*", tech. report '08 (on OpenSSL)

TLS Record Protocol

Goal: “private and reliable connection” [RFC]

Relies on pre-established connection keys



$$mac = HMAC_k(header \,||\, fragment)$$

Abbreviated Handshake (or Resumption) Protocol

- Reestablish connection keys within same session
- Keys computed using old ms (as pms), new cr, sr
- Avoids costly key agreement phase

Phases	Client		Server
Negotiation	ClientHello	→	
		←	ServerHello
Confirmation	[ChangeCipherSpec] Finished	→	
		←	[ChangeCipherSpec] Finished
Application	Application Data	↔	Application Data

(Protected by Record Protocol)

```
let verifyPMS pms ver =
  let bver = bytes_of_ProtocolVersion ver in
  let prefix, random = parsePMS pms in
  if prefix = bver then ()
  else failwith "client_version and PMS version do not correspond”
```

Symbolic Implementation of Crypto

Crypto.fsi (interface)

```
type bytes                    (* byte arrays *)
type symkey                   (* symmetric keys *)

…

val mkNonce: unit -> bytes  (* generate nonce *)
val mkKey: nonce -> symkey (* make  key *)

val sha1: bytes -> byte

val aes_encrypt: symkey -> bytes -> bytes
val aes_decrypt: symkey -> bytes -> bytes

…
```

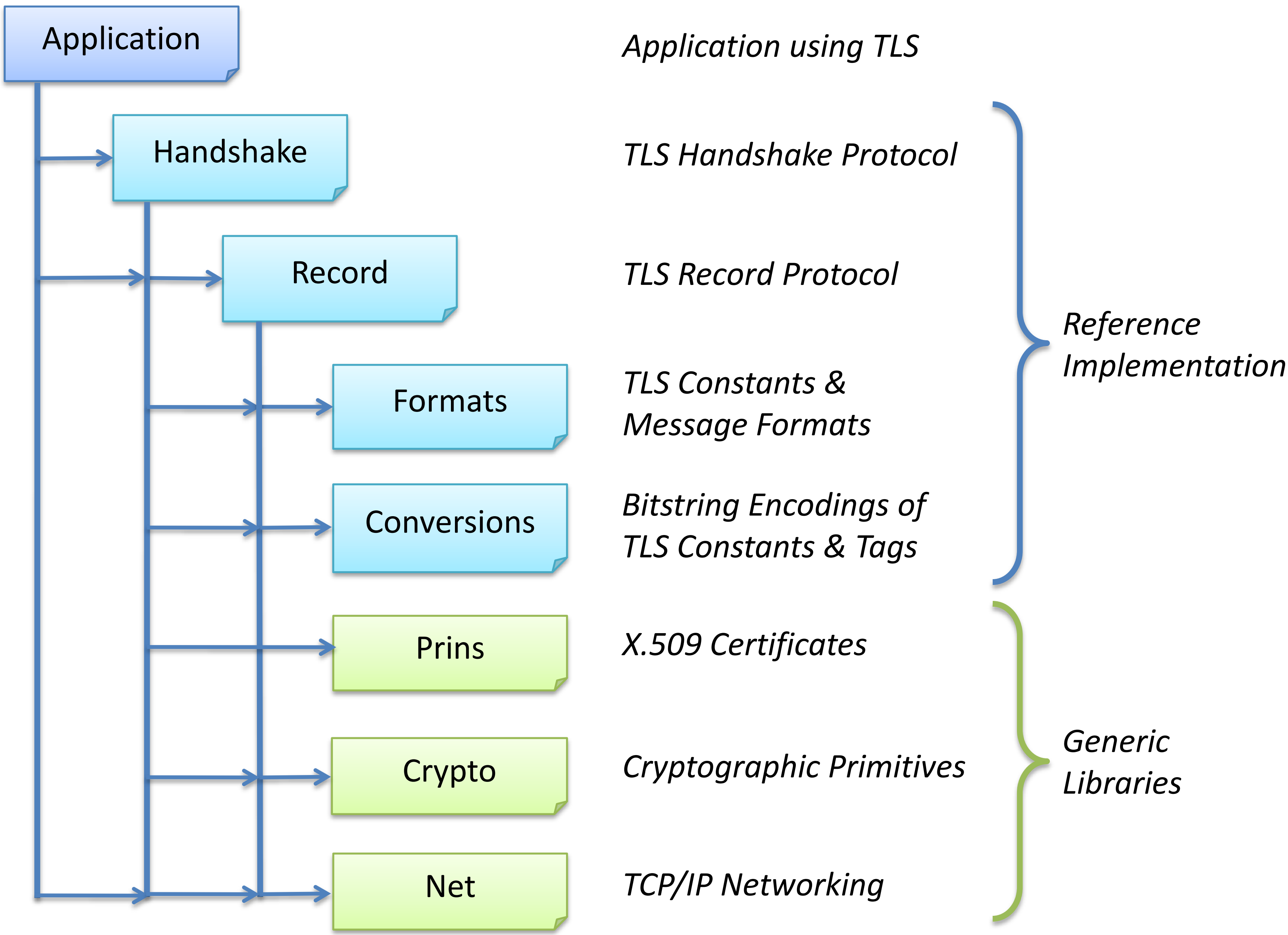
Crypto.fs (symbolic implementation)

```
type bytes =
  | Name of Pi.name
  | Hash of bytes
  | SymEncrypt of bytes * bytes
  | …
type symkey = Sym of bytes

let mkNonce () = Pi.name "nonce"
let mkKey () = Sym(mkNonce())

let sha1 b = Hash(b)

let aes_encrypt (Sym(k)) x = SymEncrypt(k,x)
let aes_decrypt (Sym(k)) (SymEncrypt(k',x)) =
  if k = k' then x else raise Fail
```


Application
Handshake
Record
Formats
Conversions
Prins
Crypto
Net
Application using TLS
TLS Handshake Protocol
TLS Record Protocol
TLS Constants & Message Formats
Bitstring Encodings of TLS Constants & Tags
X.509 Certificates
Cryptographic Primitives
TCP/IP Networking
Reference Implementation
Generic Libraries