MPRI 2-30:
Automated Verification of Cryptographic Protocol Implementations

K Bhargavan
(Slides from A.D. Gordon and C. Fournet)

Spring, 2014
Outline of Lectures

• Lecture 1 (Jan 22, Today)
  – Intro to Verified Protocol Implementations
  – RCF type system

• Lecture 2 (Jan 29)
  – F7 typechecker for RCF
  – Symbolic verification with F7

• (Skip Feb 5)

• Lecture 3 (Feb 12)
  – Computational RCF
  – Using F7 for computational verification

• Lecture 4 (Feb 19)
  – Advanced verification techniques
Protocols and Implementations

• Distributed software applications that provide abstract security functionality by using cryptographic schemes
  – Transport Layer Security (TLS)
    • secure duplex communication channel
    • used for secure web surfing (https://…)
  – SOAP Message Security (WS-Security)
    • secure XML-based RPC
    • used for accessing web services APIs (Amazon S3)

• Building blocks for secure web applications
  – Part of our trusted computing base
  – Can we guarantee that they work correctly?
Popular Protocols Still Have Attacks

• An adversary can cause honest protocol participants to violate their abstract security goals
• Flaws in design and in implementation
  – Protocols too large to analyze by hand
• Many advances in analysis of protocol models
  – Models miss many low-level implementation details
• A need for automated tools to verify protocol code
Example: Transport Layer Security

- Widely deployed protocol on clients and servers
  - HTTPs, 802.1x (EAP), FTPS, VPN, SMTP, XMPP, ...

- 18 years of attacks, fixes, and extensions
  - 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)

- Many implementations
  - OpenSSL, NSS, SChannel, GnuTLS, JSSE, PolarSSL, Bouncy Castle

- Many papers on its crypto, security & verification
  - Few analyses of TLS implementations, but many attacks
Between TCP and Application

- **Record**: private reliable connection
- **Handshake**: ciphersuite negotiation, key exchange
- **Change Cipher Spec**: signalling new keys
- **Alert**: errors and warnings
TLS Record Protocol

Typically uses a MAC-Encode-Encrypt scheme
- payload + additional data is MACed
- payload + MAC is padded
- the result is encrypted

Widely analyzed, many proofs of precise models
- yet attacks are found on implementations
- models often do not account for error handling
TLS Handshake Protocol

1. \(a \rightarrow b\): ClientHello(maxver, crandom, ciphersuites)
2. \(b \rightarrow a\): ServerHello(version, srandom, sid, ciphersuite)
3. \(b \rightarrow a\): ServerCertificate(x509("b", pk_b))
4. \(b \rightarrow a\): ServerHelloDone
   \(a\) generates a fresh pms
   \(ms = PRF(pms, "master secret", crandom | srandom)\)
   \((m_{ka}, m_{kb}, e_{ka}, e_{kb}) = PRF(ms, "key expansion", srandom | crandom)\)
5. \(a \rightarrow b\): ClientKeyExchange(rsa pk_b (maxver | pms))

   \[a \rightarrow b\]: ChangeCipherSpec
   cverify = PRF(ms, "client finished", hash (1 | 2 | 3 | 4 | 5))
6. \(a \rightarrow b\): Record(ek_{ab}, mk_{ab}, 0, h, ClientFinished(cverify))
   \[b \rightarrow a\]: ChangeCipherSpec
   sverify = PRF(ms, "server finished", hash (1 | 2 | 3 | 4 | 5 | 6))
7. \(b \rightarrow a\): Record(ek_{ba}, mk_{ba}, 0, h', ServerFinished(sverify))

- Negotiates session params, sets up keys for the Record protocol
- Many symbolic analyses, some computational analyses
  - None account for rekeying, renegotiation, multiple versions, ciphersuites
The Devil in Implementation Details

• Message processing in Record and Handshake
  – multiple ciphersuites: block ciphers, stream ciphers, MAC only
  – multiple protocol versions: SSL 3.0 – TLS 1.2

• Composite state machine for Handshake
  – interleaving: Handshake, Alert, and Application messages
  – session evolution: initial, resumption, rekeying, renegotiation

• A secure socket API for applications
  – what to reveal to the application, what to hide?
  – socket state: when can data be safely sent or received
  – error handling: notification of failures and warnings

<table>
<thead>
<tr>
<th>TLS Library</th>
<th>Crypto Use</th>
<th>API</th>
<th>State Machine</th>
<th>Formats</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>OpenSSL 1.0.0g (C)</td>
<td>9 (6460)</td>
<td>15 (6452)</td>
<td>22 (26965)</td>
<td>9 (9652)</td>
<td>55 (49529)</td>
</tr>
<tr>
<td>Sun SSL JDK 1.7 (Java)</td>
<td>22 (6870)</td>
<td>39 (7840)</td>
<td>24 (16799)</td>
<td>13 (4039)</td>
<td>96 (35548)</td>
</tr>
<tr>
<td>MSR-INRIA TLS (F#)</td>
<td>42 (1733)</td>
<td>21 (1797)</td>
<td>10 (3460)</td>
<td>33 (2676)</td>
<td>106 (8863)</td>
</tr>
</tbody>
</table>
Bugs in Message Processing

OpenSSL 0.9.8 Server Certificate Verification

• When processing a server certificate, in one case, a signature verification failure was incorrectly assumed to be a success

```
if (EVP_VerifyFinal(&md_ctx,p,(int)n,pkey)){
    /* good signature */
    ...}
```

• By sending a carefully crafted message, the adversary could bypass certificate checking
  – A standard programming error that becomes a security flaw
  – It only appears in certain ciphersuites and certain control flows
Bugs in State Machines

OpenSSL 1.0.0 Alert Fragment Handling

• An alert has two bytes \((t,d)\) and it can arrive at any time
  – \(t = 1\) for warning, \(t = 2\) for fatal error

• Suppose the attacker sends a one-byte alert fragment \(t = 1\)
  (during the initial handshake)
• Suppose the honest peer later tries to send a fatal alert \(= (t=2,d)\)
• The receiver will append 1 to \((2,d)\) and parse the alert as \((1,2)\)

• Hence, an adversary can cause a fatal alert to be ignored
  – The attack only appears when we consider Alert fragmentation
    composed with the full TLS state machine
Bugs in the TLS API

Sun Java SSLSocket API Renegotiation Hiding

• Sun SSL provides a standard Socket interface
  
  ```java
  public abstract class SSLSocket extends Socket
  ```

• `SSLSocket.read` reads one data record, but also performs a handshake if necessary, silently modifying the underlying session, without notifying the application.

• Suppose an application calls `SSLSocket.read` to get \( d \)
• Suppose it then calls `SSLSocket.read` to get \( d' \)
• Can \( d \) and \( d' \) be safely concatenated?
  - What if the underlying session changed?
  - What if we are not using the safe renegotiation extension?
Towards Verified Implementations

- Our goal is to verify the security goals of running protocol code, under precise cryptographic assumptions, and against realistic attackers.

- Can we make our symbolic and cryptographic models precise enough to handle implementation details?
  - Maybe we can generate code from models? See Spi2Java (ProVerif to Java), CV2ML (CryptoVerif to ML).

- Can we treat implementations as executable models?
  - Directly verify running code, and all its details. *We will explore this approach in these lectures.*
Specs, code, and formal tools

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

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

General Verification
- SMT Solvers
- Theorem Provers
- Model Checkers

Symbolic Analyses
- Casper
- Cryptyc
- AVISPA
- ProVerif ('01)
- F7 ('08)
- F* ('11)

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

Computational Analyses
- NRL
- Athena
- Securify
- Scyther

ML, F#
Verification Architecture

Interoperability Testing

One Source
Many Tasks

Networking

Compile
Crypto, Net
Concrete Libraries

Compile
Crypto, Net
Abstract Libraries

Verify
Symbolic/Computational Verification

Verify
Typechecking
(Symbolic/Computational)

Security Goals

Typed Library Interfaces

Protocol Code

Compile

Other Implementations

Interoperability Testing

Compile

Symbolic Debugging

Network
Writing and Verifying Protocols in F#
Source language: F#

- F#, a dialect of ML: [http://fsharp.org](http://fsharp.org)
  “Combining the strong typing, scripting and productivity of ML with the efficiency, stability, libraries, cross-language working and tools of .NET.”

- Very similar to OCaml, but with .NET libraries

- Clean strongly-typed semantics
  - Modular programming based on strong interfaces
  - Algebraic data types with pattern matching useful for symbolic cryptography, message formats
Core F# (+ RCF constructs)

- A concurrent call-by-value lambda-calculus
- Fresh (secret) names
- Channel-based communication
- Security goals as logical assumes and asserts
An F# Library of Protocol Primitives

We identify a set of platform libraries used in protocol code
- **Data**: string and byte array conversion functions
- **Net**: communication over TCP
- **Crypto**: cryptographic primitives
- **Principals**: certificate and password store
- **Xml**: manipulating XML documents
- **Db**: managing private databases

Each library is implemented as a wrapper around the corresponding .NET class
(System.Net.Sockets.TcpClient,
System.Security.Cryptography.HMACSHA1)
Example: Authenticated RPC

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

```plaintext
let client (a: str) (b: str) (k: keyab) (s: str) =
    assume (Request(a,b,s));
let c = Net.connect p in
let mac = hmacsha1 k (request s) in
Net.send c (concat (utf8 s) mac);
let (pload', mac') = iconcat (Net.recv c) in
let t = iutf8 pload' in
hmacsha1Verify k (response s t) mac';
assert(RecvResponse(a,b,s,t))

let server(a: str) (b: str) (k: keyab) : unit =
let c = Net.listen p in
let (pload, mac) = iconcat (Net.recv c) in
let s = iutf8 pload in
hmacsha1Verify k (request s) mac;
assert(RecvRequest(a,b,s));
let t = service s in
assume (Response(a,b,s,t));
let mac' = hmacsha1 k (response s t) in
Net.send c (concat (utf8 t) mac')
```
Authenticated RPC Goals

1. A request $s$ should be accepted by $b$ only if it was a request sent by $a$

2. A response $t$ should be accepted by $a$ only if it was sent by $b$ in response to $s$

\[
\forall a,b,s. \text{RecvRequest}(a,b,s) \iff (\text{Request}(a,b,s) \lor \text{Bad}(a) \lor \text{Bad}(b))
\]

\[
\forall a,b,s,t. \text{RecvResponse}(a,b,s,t) \iff (\text{Request}(a,b,s) \land \text{Response}(a,b,s,t)) \lor \text{Bad}(a) \lor \text{Bad}(b)
\]
Verification Techniques

• Model Extraction
  – Translate F# code to ProVerif
    Needs symbolic abstraction of libraries
    (tool: fs2pv)
  – Translate F# code to CryptoVerif
    Needs probabilistic semantics of ML,
    and computational abstraction of libraries
    (tool: fs2cv)

• Typechecking
  – Define secure type system for F#
    Needs semantics, abstractions,
    plus manual annotations, constraint solver
    (tool: F7)
A Symbolic Model of the Library

- An abstract model of Data and Crypto in F
- Byte arrays represented as an algebraic datatype
- Key generation as fresh name generation
- Encryption and MAC as (private) constructors
- Decryption as destructor
Other Symbolic Models of Crypto

• Using seals [Morris’73] or abstract types [Fournet’11]
  – each key contains a reference to a table that maps plaintexts to ciphertexts
  – encrypt adds plaintext to the table and encrypts 0 instead
  – decrypt looks up the table; if no value is found it still returns a value
  – under certain conditions the encryption API is computationally sound
Other Symbolic Models of Crypto

\[
\text{type } \alpha \text{ SealRef } = ((\alpha \times \text{Un}) \text{ list}) \text{ ref}
\]

\[
\text{let seal: } (\alpha \text{ SealRef } \rightarrow \alpha \rightarrow \text{Un}) = \text{fun } s \text{ m } \rightarrow
\]
\[
\begin{align*}
& \text{let state } = \text{deref } s \text{ in match first (left m) state with} \\
& \quad | \text{Some(a)} \rightarrow a \\
& \quad | \text{None } \rightarrow \text{let } a : \text{Un } = \text{Pi.name()} \text{ in } s := ((m,a)\text{::state}); a
\end{align*}
\]

\[
\text{let unseal: } (\alpha \text{ SealRef } \rightarrow \text{Un } \rightarrow \alpha ) = \text{fun } s \text{ a } \rightarrow
\]
\[
\begin{align*}
& \text{let state } = \text{deref } s \text{ in match first (right a) state with} \\
& \quad | \text{Some(m)} \rightarrow m \\
& \quad | \text{None } \rightarrow \text{failwith } "\text{not a sealed value}" \\
\end{align*}
\]

\[
\text{type } \alpha \text{ Seal } = (\alpha \rightarrow \text{Un}) \times (\text{Un } \rightarrow \alpha)
\]

\[
\text{let mkSeal } (n:\text{string}) : \alpha \text{ Seal } =
\]
\[
\begin{align*}
& \quad \text{let } s:\alpha \text{ SealRef } = \text{ref } [] \text{ in } (\text{seal } s, \text{unseal } s)
\end{align*}
\]
Symbolic Verification Goal

Given a protocol implementation P.ml with public interface P.mli

Suppose we have symbolic models for our libraries

LIBS=Data.ml, Crypto.ml, Net.ml, Principals.ml

with public interfaces

Data.mli, Crypto.mli, Net.mli, Principals.mli

Our goal is that the program LIBS P.ml is robustly safe.

That is, for all opponents O.ml that respect the public interfaces of LIBS and P.mli, LIBS P.ml O.ml is safe
Symbolic Verification by Model Extraction
The Applied Pi Calculus [Abadi, Fournet]

- A process calculus with an equational theory
- Popular formalism for cryptographic protocols
- Processes can be verified using ProVerif [Blanchet]

\[
P, Q, R ::= \\
\quad \text{\texttt{in}}(M, x); P \\
\quad \text{\texttt{out}}(M, N); P \\
\quad \text{\texttt{new}} a; P \\
\quad !P \\
\quad P | Q \\
\quad 0 \\
\quad \text{event } M \\
\quad \text{let } x_1, \ldots, x_n \text{ such that } M = N \text{ in } P \text{ else } Q \\
\quad \text{let } x = g(M_1, \ldots, M_n) \text{ in } P \text{ else } Q
\]
Extracting Applied Pi Models from F

• We define a translation from F programs to applied pi calculus scripts
  – Follows Milner’s “Functions as processes”
  – Includes optimizations for eliminating irrelevant code

• We prove that the translation is **sound**: if the translation is robustly safe, so is the source.

```
let f x = match x with F(y)→y | G(z)→z

new f;
(!in(internal(f), (arg,kR));
  let y suchthat F(y) = arg in out(kR,y)
  else let z suchthat G(z) = arg in out(kR,z) else 0
| !in(f,(arg,kR)); out(internal(f),(arg,kR))
| ...)
```
Translating Authenticated RPC

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

```
let client (a: str) (b: str) (k: keyab) (s: str) in
  assume (Request(a,b,s));
  let c = Net.connect p in
  let mac = hmacsha1 k (request s) in
  Net.send c (concat (utf8 s) mac);
  let (pload',mac') = icomcat (Net.receive p) in
  hmacsha1Verify k (response s t) mac'
  assert(RecvResponse(a,b,s,t))
```

```python
(Client clientAB, (s,K));

event RPC.Request(SaS,SbS,s);
  let mac = Crypto.hmacsha1(key,
    RPC.request(s)) in
  let req = Data.concat(Data.utf8(s),mac) in
  out(Net.pubChan, req);

in(Net.pubChan, resp);
  let (pload,mac) = Data.icomcat(resp) in
  let t = Data.utf8(pload) in
  if Crypto.hmacsha1Verify(key,
    RPC.response(s,t),mac) = True()
  then
    event Expect(RPC.RecvResponse(SaS,SbS,s,t));
    out(K, ())
  else 0)
```
Translating Crypto

```plaintext
type bytes =
    | Name of Pi.name
    | MAC of bytes * bytes
    | Encrypt of bytes * bytes

let hmacsha1 k t = MAC(k,t)
let hmacsha1Verify k x e = hmacsha1 k x = e

private fun Data.MAC/2.
reduc Crypto.hmacsha1(key,text) =
    Data.MAC(key,text).
reduc Crypto.hmacsha1Verify(key,text,
    Data.MAC(key,text)) = True().
```
Security Verification using ProVerif

To use ProVerif for security verification:

1. Write symbolic models of libraries and their public interfaces
   Crypto.ml, Crypto.mli, Net.ml, Net.mli …
2. Annotate the program with security goals
   P.ml
3. Define the public (attacker) F# interface
   P-pub.mli
4. Translate all modules and their interfaces to applied pi
   fs2pv Crypto.ml Crypto.mli … P.ml P-pub.mli > Script.pv
5. Run ProVerif on Script.pv.
   ProVerif may verify the script, or produce a counterexample, or it may not terminate

Security Theorem:

If the script Script.pv is verified by ProVerif then the program
Crypto.ml Crypto.mli … P.ml P.ml is robustly safe.
(by soundness of the translation and the correctness of ProVerif)
Symbolic Verification by Typechecking
An enhanced type system that tracks logical invariants:

- **Refinement Types**: \( x : ty \{ \phi \} \)
  - \( x : \text{int} \{ x > 0 \} \)
  - \( \text{pwd} : \text{string} \{ \text{Password("Alice"}, \text{pwd}) \} \)

- **Dependent function types**: \( x : ty \rightarrow ty' \)
  - \( \text{lookup} : u : \text{string} \{ \text{User}(u) \} \rightarrow p : \text{string} \{ \text{Password}(u, p) \} \)
  - Pre-condition: \( \text{User}(u) \), post-condition: \( \text{Password}(u, p) \)

- **Refinement subtyping**: \( ty <: ty' \)
  - \( x : ty \{ \phi \} <: ty \)

- **Typechecking guarantees** (statically) that no `assert` will fail at run-time
  - That is, each `assert` is entailed by its preceding assumes
  - It relies on an external SMT solver for logical proof obligations
Summary: Safety by Typing

**Core Syntax of Types:**

\[
T, U, V ::= \text{type} \\
\quad \text{unit} \quad \text{unit type} \\
\quad x : T \to U \quad \text{dependent function type (scope of } x \text{ is } U) \\
\quad x : T \times U \quad \text{dependent pair type (scope of } x \text{ is } U) \\
\quad T + U \quad \text{disjoint sum type} \\
\quad \text{rec } \alpha.T \quad \text{iso-recursion type (scope of } \alpha \text{ is } T) \\
\quad \alpha \quad \text{type variable (abstract or iso-recursion)} \\
\quad x : T\{C\} \quad \text{refinement type (scope of } x \text{ is } C)
\]

*Type environments, } E, \text{ are sequences } \mu_1, \ldots, \mu_n, \text{ where each } \mu_i \text{ is a subtype assumption } \alpha \ll : \alpha', \text{ an abstract type, or an entry } x : T.\]*

The two main judgments are:

- \text{subtyping: } E \vdash T \ll : U \\
- \text{type assignment: } E \vdash A : T

**Theorem 1 (CSF’08)** If } \emptyset \vdash A : T, \text{ then for all executions of program } A, \text{ all asserted formulas logically follow from previously-assumed formulas.}
Security Verification using F7

To use F7 for security verification:

1. Annotate libraries with security assumptions
   Crypto.ml7, Net.ml7, …

2. Annotate the program with security goals
   P.ml7, P.ml

3. Define the public (attacker) F# interface
   P-pub.mli

4. Verify that the program typechecks against both the public interface and the program interface
   Crypto.ml7, Net.ml7, … |- P.ml : P.ml7 <: P-pub.mli

Security Theorem (by type safety):

If the libraries Lib.ml satisfy Crypto.ml7, Net.ml7, … then the program Lib.ml P.ml is robustly safe.
Refinement Types for Crypto

- Every MAC key has a usage precondition
- \textbf{MACSays}(k,b) says that mackey k may be used to MAC b
- Each protocol defines MACSays for its own keys
- Secret plaintexts may not be encrypted with compromised keys
- Decryption does not authenticate data

\begin{verbatim}
module Crypto
val hmac_keygen: unit \rightarrow mackey
val hmacsha1: k:mackey \rightarrow
  b:bytes \{ MACSays(k,b) \} \rightarrow
  h:bytes \{ IsMAC(h,k,b) \}
val hmacsha1Verify: k:mackey \rightarrow
  b:bytes \rightarrow h:bytes \rightarrow
  r:bool \{ r=true \Rightarrow MACSays(k,b) \}
val aes_keygen: unit \rightarrow symkey
val aes_encrypt: k:symkey \rightarrow
  b:bytes \{ Pub(b) \Rightarrow Pub.k(k) \} \rightarrow
  e:bytes \{ IsEncryption(e,k,b) \}
val aes_decrypt: k:symkey \rightarrow e:bytes \rightarrow
  b:bytes \{ \forall p. IsEncryption(e,k,p) \Rightarrow b = p \}
\end{verbatim}
Verifying Authenticated RPC

```plaintext
type \( (a: \text{str}, b: \text{str}) \rightarrow \text{k: mackey\{KeyAB}\{k, a, b\}} \)
val mkKeyAB: \( a: \text{str} \rightarrow b: \text{str} \rightarrow (a, b) \rightarrow \text{keyab} \)
val client: \( a: \text{str} \rightarrow b: \text{str} \rightarrow k:(a, b) \rightarrow \text{payload} \rightarrow \text{unit} \)
val server: \( a: \text{str} \rightarrow b: \text{str} \rightarrow k:(a, b) \rightarrow \text{keyab} \rightarrow \text{unit} \)

\text{assume} \quad \forall a, b, k, m. \text{KeyAB}\{k, a, b\} \Rightarrow
( \text{MACSays}\{k, m\} \iff (\exists s. \text{Requested}\{m, s\} \land \text{Request}\{a, b, s\}) \lor
(\exists s, t. \text{Responded}\{m, s, t\} \land \text{Response}\{a, b, s, t\}) \lor
(\text{Bad}\{a\} \lor \text{Bad}\{b\})) \)

let client (a: str) (b: str) (k: keyab) (s: str) =
  assume (Request(a, b, s));
let c = Net.connect p in
let mac = hmacsha1 k (request s) in
Net.send c (concat (utf8 s) mac);
let (pload', mac') = iconcat (Net.recv c) in
let t = iutf8 pload' in
hmacsha1Verify k (response s t) mac';
assert(RecvResponse(a, b, s, t))
```
Results
## Protocol Implementations in F#

<table>
<thead>
<tr>
<th>Protocol</th>
<th>Modules</th>
<th>LOCs</th>
<th>Messages</th>
<th>Crypto Ops</th>
</tr>
</thead>
<tbody>
<tr>
<td>Trusted Library</td>
<td>6</td>
<td>1456</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Password-based MAC</td>
<td>1</td>
<td>38</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>Authenticated RPC</td>
<td>1</td>
<td>91</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>Otway-Rees</td>
<td>1</td>
<td>148</td>
<td>4</td>
<td>8</td>
</tr>
<tr>
<td>Transport Layer Security</td>
<td>9</td>
<td>4498</td>
<td>12</td>
<td>15</td>
</tr>
<tr>
<td>Web Services Security Library</td>
<td>6</td>
<td>1925</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>X.509-based XML Signature</td>
<td>1</td>
<td>85</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>Password-X.509 Authenticated XML RPC</td>
<td>1</td>
<td>149</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>X.509-based Authenticated XML RPC</td>
<td>1</td>
<td>117</td>
<td>2</td>
<td>9</td>
</tr>
<tr>
<td>Windows Cardspace</td>
<td>9</td>
<td>1429</td>
<td>4</td>
<td>37</td>
</tr>
</tbody>
</table>
## Verification with ProVerif

<table>
<thead>
<tr>
<th>Protocol</th>
<th>Authentication</th>
<th>Secrecy</th>
<th>Queries</th>
<th>Verification Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Password-based MAC</td>
<td>message</td>
<td>password</td>
<td>5</td>
<td>0.8s</td>
</tr>
<tr>
<td>Authenticated RPC</td>
<td>session</td>
<td>key</td>
<td>16</td>
<td>1m50s</td>
</tr>
<tr>
<td>Otway-Rees</td>
<td>session</td>
<td>key</td>
<td>16</td>
<td>1m50s</td>
</tr>
<tr>
<td>X.509-based XML Signature</td>
<td>message</td>
<td>key</td>
<td>18</td>
<td>51m</td>
</tr>
<tr>
<td>Password-X.509 Authenticated XML RPC</td>
<td>session</td>
<td>message</td>
<td>18</td>
<td>51m</td>
</tr>
<tr>
<td>X.509-based Authenticated XML RPC</td>
<td>session</td>
<td>message</td>
<td>18</td>
<td>51m</td>
</tr>
<tr>
<td>Self-Issued-X.509 Cardspace</td>
<td>session</td>
<td>token</td>
<td>10</td>
<td>38s</td>
</tr>
<tr>
<td>Password-X.509 Cardspace</td>
<td>session</td>
<td>token</td>
<td>13</td>
<td>20m53s</td>
</tr>
<tr>
<td>Password-TLS Cardspace</td>
<td>session</td>
<td>token</td>
<td>13</td>
<td>24m40s</td>
</tr>
<tr>
<td>X.509-X.509 Cardspace</td>
<td>session</td>
<td>token</td>
<td>13</td>
<td>66m21s</td>
</tr>
<tr>
<td>TLS Handshake</td>
<td>session</td>
<td>key</td>
<td>4</td>
<td>52s</td>
</tr>
<tr>
<td>TLS Handshake + Resumption</td>
<td>session</td>
<td>key</td>
<td>2</td>
<td>8m</td>
</tr>
<tr>
<td>TLS Record</td>
<td>message</td>
<td>message</td>
<td>2</td>
<td>11m</td>
</tr>
<tr>
<td>Full TLS</td>
<td>session</td>
<td>key,message</td>
<td>10</td>
<td>3.5h</td>
</tr>
<tr>
<td>Full TLS + Password-based Client</td>
<td>user</td>
<td>key,message</td>
<td>1</td>
<td>1.5h</td>
</tr>
</tbody>
</table>
Verification with F7

<table>
<thead>
<tr>
<th>Protocol</th>
<th>Modules</th>
<th>F# Interfaces</th>
<th>F7 Interfaces</th>
<th>F7 Time</th>
<th>Fs2PV Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Trusted Libraries</td>
<td>6</td>
<td>395</td>
<td>1167</td>
<td>29s</td>
<td>-</td>
</tr>
<tr>
<td>Authenticated RPC</td>
<td>1</td>
<td>21</td>
<td>103</td>
<td>10s</td>
<td>6.6s</td>
</tr>
<tr>
<td>Cryptographic Patterns</td>
<td>1</td>
<td>46</td>
<td>260</td>
<td>17.1s</td>
<td>-</td>
</tr>
<tr>
<td>Otway-Rees</td>
<td>1</td>
<td>21</td>
<td>255</td>
<td>1m29s</td>
<td>1m50s</td>
</tr>
<tr>
<td>Web Services Security Library</td>
<td>6</td>
<td>257</td>
<td>475</td>
<td>48.8s</td>
<td>-</td>
</tr>
<tr>
<td>X.509-based XML Signature</td>
<td>1</td>
<td>8</td>
<td>22</td>
<td>10.8s</td>
<td>2.6s</td>
</tr>
<tr>
<td>X.509-based XML RPC</td>
<td>1</td>
<td>8</td>
<td>53</td>
<td>19.8s</td>
<td>51m</td>
</tr>
<tr>
<td>X.509-X.509 Cardspace</td>
<td>1</td>
<td>25</td>
<td>309</td>
<td>6m3s</td>
<td>66m21s</td>
</tr>
</tbody>
</table>

- Other verifications not covered here:
  - Secure Multi-Party Sessions (2k lines, 2m)
  - TLS Transport Protocol (computational) (>5k lines, 13m)
Relevant Publications

Summary

• Protocol models often miss important security-critical details of implementations

• Several ways to address this gap
  – Generate code from models (e.g. CV2ML)
  – Extract models from code (e.g. fs2pv)
  – Directly analyze protocol code (e.g. F7)

• Our main focus will be on symbolic and computational verification by typechecking with F7
RCF: a concurrent $\lambda$-calculus with refinement types

Slides adapted from A. D. Gordon’s
see also the RCF tutorial
Tools, Reading

- F# compiler: http://fsharp.org/
- F7 typechecker: http://research.microsoft.com/f7

Lecture notes on the course website:
- RCF: a compact definition of the type system, plus exercises
- *Principles and Applications of Refinement Types*, a tutorial on the RCF type system.
- *Cryptographic Verification by Typing for a Sample Protocol Implementation*, a tutorial on F7

Research papers: http://research.microsoft.com/f7
A formal core for ML (outline)

• An assembly of standard parts, generalizing some ad hoc constructions in language-based security
  – FPC (Plotkin 1985, Gunter 1992) – core of ML and Haskell
  – Concurrency in style of the pi-calculus (Milner, Parrow, Walker 1989) but for a lambda-calculus (like 80s languages PFL, Poly/ML, CML)
  – Symbolic crypto is derivable e.g. by coding up seals (Morris 1973, Sumii and Pierce 2002), not primitive as in the applied pi calculus
  – Security specs via assume/assert (Floyd, Hoare, Dijkstra 1970s), generalizing eg correspondences (Woo and Lam 1992)
  – To check assertions statically, rely on dependent functions and pairs with subtyping (Cardelli 1988) and refinement types (Pfenning 1992, ...) aka predicate subtyping (as in PVS, and more recently Russell)
  – Public/tainted kinds to track data that may flow to or from the opponent, as in Cryptyc (Gordon, Jeffrey 2002)
• For experiment, there is a downloadable implementation F7
RCF Part 1:
SYNTAX AND SEMANTICS
The Core Language (FPC):

\[ x, y, z \]
\[ h ::= \]
\[ \text{inl} \quad \text{left constructor of sum type} \]
\[ \text{inr} \quad \text{right constructor of sum type} \]
\[ \text{fold} \quad \text{constructor of iso-recursive type} \]
\[ M, N ::= \]
\[ x \quad \text{variable} \]
\[ () \quad \text{unit} \]
\[ \text{fun} x \to A \quad \text{function (scope of } x \text{ is } A) \]
\[ (M, N) \quad \text{pair} \]
\[ h M \quad \text{construction} \]
\[ A, B ::= \]
\[ M \quad \text{value} \]
\[ M N \quad \text{application} \]
\[ M = N \quad \text{syntactic equality} \]
\[ \text{let } x = A \text{ in } B \quad \text{let (scope of } x \text{ is } B) \]
\[ \text{let } (x, y) = M \text{ in } A \quad \text{pair split (scope of } x, y \text{ is } A) \]
\[ \text{match } M \text{ with } h x \to A \text{ else } B \quad \text{constructor match (scope of } x \text{ is } A) \]
The Reduction Relation: $A \rightarrow A'$

$\text{(fun } x \rightarrow A)\ N \rightarrow A\{N/x\}$

$\text{(let } (x_1, x_2) = (N_1, N_2) \text{ in } A) \rightarrow A\{N_1/x_1\}\{N_2/x_2\}$

$\text{(match } M \text{ with } h\ x \rightarrow A \text{ else } B) \rightarrow \begin{cases} A\{N/x\} & \text{if } M = h\ N \text{ for some } N \\ B & \text{otherwise} \end{cases}$

$M = N \rightarrow \begin{cases} \text{inl()} & \text{if } M = N \\ \text{inr()} & \text{otherwise} \end{cases}$

$\text{let } f = \text{fun } x \rightarrow x + 1 \text{ in } (f\ 7) \rightarrow (\text{fun } x \rightarrow x + 1)\ 7 \rightarrow 7 + 1 \rightarrow 8$

$A \rightarrow A' \Rightarrow \text{let } x = A \text{ in } B \rightarrow \text{let } x = A' \text{ in } B$
Example: Booleans and Conditional Branching:

\[
\begin{align*}
\text{false} & \triangleq \text{inl} () \\
\text{true} & \triangleq \text{inr} () \\
\text{if } A \text{ then } B \text{ else } B' & \triangleq \\
& \quad \text{let } x = A \text{ in match } x \text{ with inr}(_-) \rightarrow B \text{ else match } x \text{ with inl}(_-) \rightarrow B'
\end{align*}
\]
Example: Booleans and Conditional Branching:

\[
\begin{align*}
\text{false} & \triangleq \text{inl}() \\
\text{true} & \triangleq \text{inr}() \\
\text{if } A \text{ then } B \text{ else } B' & \triangleq \\
& \text{let } x = A \text{ in match } x \text{ with } \text{inr}(\_ \_ \_) \rightarrow B \text{ else match } x \text{ with } \text{inl}(\_ \_ \_) \rightarrow B'
\end{align*}
\]

Exercise: Expressiveness

1. Derive arithmetic, that is, value zero, functions succ, pred, and iszero.

2. Derive list processing, that is, value nil, functions cons, hd, tl, and null.

3. Write down an expression \( \Omega \) that diverges, that is, \( \Omega \rightarrow A_1 \rightarrow A_2 \rightarrow \ldots \).

4. Derive a fixpoint function \( \text{fix} \) so that we can define recursive function definitions as follows: \( \text{let rec } fx = A \triangleq \text{let } f = \text{fix} \ (\text{fun } f \rightarrow \text{fun } x \rightarrow A) \).
COMMUNICATIONS &
CONCURRENCY
Communications and Concurrency:

\[ A, B ::= \]

\[ \ldots \]

\[ (\forall a)A \]

\[ a!M \]

\[ a? \]

\[ A \parallel B \]

\[ a!M \parallel a? \rightarrow M \]

\[ A \rightarrow A' \quad \text{if} \quad A \Rightarrow B, B \Rightarrow B', B' \Rightarrow A' \]

- as before
- local channel
- transmission of \( M \) on channel \( a \)
- receive message off channel
- parallel composition

Communications step

Reductions step are “up to structural rearrangements”
Towards Concurrency: The Heating Relation $A \Rightarrow A'$

Axioms $A \equiv A'$ are read as both $A \Rightarrow A'$ and $A' \Rightarrow A$.

$A \Rightarrow A$

$A \Rightarrow A''$ if $A \Rightarrow A'$ and $A' \Rightarrow A''$

$A \Rightarrow A' \Rightarrow \text{let } x = A \text{ in } B \Rightarrow \text{let } x = A' \text{ in } B$

$A \rightarrow A'$ if $A \Rightarrow B, B \rightarrow B', B' \Rightarrow A'$

Heatings is an auxiliary relation; its purpose is to enable reductions, and to place every expression in a normal form, known as a *structure*. This style of operational semantics is called “chemical abstract machine” [Berry, Boudol].

Reductions step are “up to structural rearrangements”
Parallel Composition:

\[ A, B ::= \]

\[ \ldots \]

\[ A \triangleright B \]

expression as before fork

\[ () \triangleright A \equiv A \]

\[ (A \triangleright A') \triangleright (A'') \equiv A \triangleright (A' \triangleright A'') \]

\[ (A \triangleright A') \triangleright (A'') \Rightarrow (A' \triangleright A) \triangleright (A'') \]

let \( x = (A \triangleright A') \) in \( B \equiv A \triangleright (\text{let } x = A' \text{ in } B) \)

\[ A \Rightarrow A' \Rightarrow (A \triangleright B) \Rightarrow (A' \triangleright B) \]

\[ A \Rightarrow A' \Rightarrow (B \triangleright A) \Rightarrow (B \triangleright A') \]

\[ A \rightarrow A' \Rightarrow (A \triangleright B) \rightarrow (A' \triangleright B) \]

\[ B \rightarrow B' \Rightarrow (A \triangleright B) \rightarrow (A \triangleright B') \]

Exercise: Which parameter is passed to the function \( F \) by the following expression:

let \( x = (1 \triangleright (2 \triangleright 3)) \) in \( Fx \)
Channel-Based Communications:

\[ A, B ::= \]

\[ \ldots \]

\[ a!M \]

\[ a? \]

expression as before

transmission of \( M \) on channel \( a \)

receive message off channel

\[ a!M \Rightarrow a!M \triangleright () \]

\[ a!M \triangleright a? \rightarrow M \]

\[ \begin{align*}
  a!0 \triangleright a!1 \triangleright (\text{let } x = a? \text{ in } (a!(x+2) \triangleright x)) \\
  \equiv a!1 \triangleright \text{let } x = (a!0 \triangleright a?) \text{ in } (a!(x+2) \triangleright x) \\
  \rightarrow a!1 \triangleright \text{let } x = 0 \text{ in } (a!(x+2) \triangleright x) \\
  \rightarrow a!1 \triangleright a!2 \triangleright 0
\end{align*} \]

\[ \begin{align*}
  a!0 \triangleright a!1 \triangleright (\text{let } x = a? \text{ in } (a!(x+2) \triangleright x)) \\
  \equiv \rightarrow \rightarrow a!0 \triangleright a!3 \triangleright 1
\end{align*} \]
Name Generation:

\[ A, B ::= \]

\[ \ldots \]

\[ (\forall a)A \]

expression as before fork

\[ A \Rightarrow A' \Rightarrow (\forall a)A \Rightarrow (\forall a)A' \]

\[ a \notin \text{fn}(A') \Rightarrow A' \triangleright ((\forall a)A) \Rightarrow (\forall a)(A' \triangleright A) \]

\[ a \notin \text{fn}(A') \Rightarrow ((\forall a)A) \triangleright A' \Rightarrow (\forall a)(A \triangleright A') \]

\[ a \notin \text{fn}(B) \Rightarrow \textbf{let } x = (\forall a)A \textbf{ in } B \Rightarrow (\forall a)\textbf{let } x = A \textbf{ in } B \]

\[ A \rightarrow A' \Rightarrow (\forall a)A \rightarrow (\forall a)A' \]
Example: Concurrent ML:

\[(T)\text{chan} \triangleq (T \to \text{unit}) \star (\text{unit} \to T)\]

\[\text{chan} \triangleq \text{fun } x \to (\forall a)(\text{fun } x \to a!x, \text{fun } _- \to a?)\]

\[\text{send} \triangleq \text{fun } c \ x \to \text{let } (s, r) = c \ in \ s \ x \quad \text{send } x \text{ on } c\]

\[\text{recv} \triangleq \text{fun } c \to \text{let } (s, r) = c \ in \ r () \quad \text{block for } x \text{ on } c\]

\[\text{fork} \triangleq \text{fun } f \to (f() \uparrow ()) \quad \text{run } f \text{ in parallel}\]

Example: Mutable State:

\[(T)\text{ref} \triangleq (T)\text{chan}\]

\[\text{ref } M \triangleq \text{let } r = \text{chan } "x" \text{ in send } r\ M; r \quad \text{new reference to } M\]

\[\text{deref } M \triangleq \text{let } x = \text{recv } M \text{ in send } M\ x; x \quad \text{dereference } M\]

\[M := N \triangleq \text{let } x = \text{recv } M \text{ in send } M\ N \quad \text{update } M \text{ with } N\]
FUNCTIONAL PROGRAMMING AND CONCURRENCY
LOGICAL SPECIFICATIONS
Specifications: **Assume and Assert**

- Suppose there is a global set of formulas, 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, we say the assertion fails.
  - The log is only for specification purposes; it does not affect execution.

- Our use of first-order logic predicates in C generalizes conventional assertions (like `assert` i>0 in eg JML, Spec#)
  - Such predicates usefully represent security-related concepts like roles, permissions, events, compromises.
A General Class of Logics:

\[ C ::= p(M_1, \ldots, M_n) \mid M = M' \mid C \land C' \mid C \lor C' \mid \neg C \mid C \Rightarrow C' \mid \forall x.C \mid \exists x.C \]

\[ \{C_1, \ldots, C_n\} \vdash C \]

deducibility relation

Assume and Assert:

\[ A, B ::= \]

expression

\[ \ldots \]

as before

\[ \text{assume } C \]

assumption of formula \( C \)

\[ \text{assert } C \]

assertion of formula \( C \)

\[ \text{assume } C \Rightarrow \text{assume } C \uparrow () \]

\[ \text{assert } C \rightarrow () \]
Semantics: expression safety

- We use a standard small-step reduction semantics; runtime configurations are expressions of the form

$$S ::= (\forall a_1)\ldots(\forall a_\ell) \left( \prod_{i=1}^{m} \text{assume } C_i \right) \left( \prod_{j=1}^{n} c_j!M_j \right) \left( \prod_{k=1}^{o} L_k\{e_k\} \right)$$

  active assumptions pending messages running threads

- An expression is safe when, for all runs of $A$, all assertions succeed
Are these expressions safe?

\textbf{assert} \ (p \land q \Rightarrow q) \\
\textbf{assert} \ (p \lor q \Rightarrow q) \\
\textbf{assume} \ (p \Rightarrow q); \ \textbf{assert} \ (p \lor q \Rightarrow q) \\

\textbf{let} \ x = 0 \ \textbf{in} \ \textbf{assert} \ (x = 1) \\
a!0 \vdash \ a!1 \vdash \ (\textbf{let} \ x = a? \ \textbf{in} \ \textbf{assert} \ (x=0 \lor x=1)) \\
a!0 \vdash \ a!1 \vdash \ (\textbf{let} \ x = a? \ \textbf{in} \ \textbf{assert} \ x=1) \\
a!0 \vdash \ a!1 \vdash \ (\textbf{let} \ x = a? \ \textbf{in} \ \textbf{if} \ x > 0 \ \textbf{then} \ \textbf{assert} \ x=1) \\
\ldots
Structures and Static Safety:

\[ e ::= M | MN | M = N | \text{let } (x, y) = M \text{ in } B | \text{match } M \text{ with } h x \to A \text{ else } B | M? | \text{assert } C \]

\[ \Pi_{i \in 1..n} A_i \overset{\Delta}{=} () \vdash A_1 \vdash \ldots \vdash A_n \]

\[ \mathcal{L} ::= \{} | (\text{let } x = \mathcal{L} \text{ in } B) \]

\[ S ::= (\forall a_1) \ldots (\forall a_\ell) \left( \left( \prod_{i \in 1..m} \text{assume } C_i \right) \vdash \left( \prod_{j \in 1..n} c_j \text{!} M_j \right) \vdash \left( \prod_{k \in 1..o} \mathcal{L}_k \{e_k\} \right) \right) \]

Let structure \( S \) be \textit{statically safe} if and only if,
for all \( k \in 1..o \) and \( C \), if \( e_k = \text{assert } C \) then \( \{C_1, \ldots, C_m\} \vdash C \).

\textbf{Lemma} For every expression \( A \), there is a structure \( S \) such that \( A \Rightarrow S \).

Expression Safety:

Let expression \( A \) be \textit{safe} if and only if,
for all \( A' \) and \( S \), if \( A \rightarrow^* A' \) and \( A' \Rightarrow S \), then \( S \) is statically safe.
PROGRAMMING EXAMPLE:

ACCESS CONTROL IN PARTIALLY-TRUSTED CODE
Example: access control for files

- **Untrusted** code may call a **trusted** library
- Trusted code expresses security policy with assumes and asserts

```ocaml

let read file = assert(CanRead(file)); ...
let delete file = assert(CanWrite(file)); ...

let pwd = "C:/etc/password"
let tmp = "C:/temp/tempfile"

assume CanWrite(tmp)
assume \( \forall x. \text{CanWrite}(x) \rightarrow \text{CanRead}(x) \)

let untrusted() =
  let v1 = read tmp in // ok, by policy
  let v2 = read pwd in // assertion fails

```

Typechecking failed at aclfs.fs(39,9)–(39,12)
Error: Cannot establish formula CanRead(pwd)

- Each policy violation causes an assertion failure
- We **statically** prevent any assertion failures by typing
Logging dynamic events

- Security policies often stated in terms of dynamic events such as role activations or data checks.

- We mark such events by adding formulas to the log with `assume`.

```plaintext
type facts = ... | PublicFile of string
let read file = assert(CanRead(file)); ...
let readme = "C:/public/README"

// Dynamic validation:
let publicfile f =
  if f = "C:/public/README" || ...
  then assume (PublicFile(f))
  else failwith "not a public file"

assume ∀x. PublicFile(x) → CanRead(x)

let untrusted() =
  let v2 = read readme in // assertion fails
  publicfile readme; // validate the filename
  let v3 = read readme in () // now, ok
```
Access control with refinement types

```
val read: file:string\{CanRead(file)\} → string
val delete: file:string\{CanDelete(file)\} → unit
val publicfile: file:string → unit\{PublicFile(file)\}
```

- Preconditions express access control requirements
- Postconditions express results of validation
- We typecheck partially trusted code to guarantee that all preconditions (and hence all asserts) hold at runtime
F7: refinement typechecking for F#

- We program in F#
- We specify in F7 We typecheck programs against interfaces
- F7 does some type inference & calls Z3, an SMT solver, on each logical proof obligation
- We thus develop crypto libraries and verify protocol implementations
Access control for files (demo)
RCF Part 2:

TYPES FOR SAFETY
Starting Point: The Type System for FPC:

\[
\begin{align*}
E \vdash \emptyset & \quad (x : T) \in E \\
E \vdash A : T & \quad E, x : T \vdash B : U \\
E \vdash x : T & \quad E \vdash \text{let } x = A \text{ in } B : U \\
E \vdash \emptyset & \quad E \vdash M : T & \quad E \vdash N : U \\
E \vdash () : \text{unit} & \quad E \vdash M = N : \text{unit + unit} \\
E, x : T \vdash A : U & \quad E \vdash M : (T \to U) & \quad E \vdash N : T \\
E \vdash \text{fun } x \to A : (T \to U) & \quad E \vdash MN : U \\
E \vdash M : T & \quad E \vdash N : U & \quad E \vdash M : (T \times U) & \quad E, x : T, y : U \vdash A : V \\
E \vdash (M, N) : (T \times U) & \quad E \vdash \text{let } (x, y) = M \text{ in } A : V \\
h : (T, U) & \quad E \vdash M : T & \quad E \vdash U & \quad E \vdash M : T & \quad h : (H, T) & \quad E, x : H \vdash A : U & \quad E \vdash B : U \\
E \vdash hM : U & \quad E \vdash \text{match } M \text{ with } h x \to A \text{ else } B : U \\
\text{inl} : (T, T + U) & \quad \text{inr} : (U, T + U) & \quad \text{fold} : (T \{\mu \alpha.T/\alpha\}, \mu \alpha.T)
\end{align*}
\]
Three Steps Toward Safety by Typing

1. We include **refinement types** \( \{ x : T \mid C \} \)
   whose values are those of \( T \) that satisfy \( C \)

2. To exploit refinements, we add a judgment \( E \vdash C \)
   meaning that \( C \) follows from the refinement types in \( E \)

3. To manage refinement formulas, we need (1) dependent
   versions of the function and pair types, and (2) subtyping
   - A value of \( x : T \to U \) is a function \( M \) such that if \( N \)
     has type \( T \), then \( M \ N \) has type \( U \{ N/x \} \).

   - A value of \( x : T \times U \) is a pair \((M,N)\) such that \( M \)
     has type \( T \) and \( N \) has type \( U \{ M/x \} \).

   - If \( A : T \) and \( T <: U \) then \( A : U \).
Syntax of RCF Types:

\[ H, T, U, V ::= \text{type} \]

- unit: unit type
- \( \Pi x : T. U \): dependent function type (scope of \( x \) is \( U \))
- \( \Sigma x : T. U \): dependent pair type (scope of \( x \) is \( U \))
- \( T + U \): disjoint sum type
- \( \mu \alpha.T \): iso-recursive type (scope of \( \alpha \) is \( T \))
- \( \alpha \): iso-recursive type variable
- \( \{ x : T | C \} \): refinement type (scope of \( x \) is \( C \))

\[ \{ C \} \overset{\Delta}{=} \{ - : \text{unit} | C \} \quad \text{ok-type} \]

\[ \text{bool} \overset{\Delta}{=} \text{unit} + \text{unit} \quad \text{Boolean type} \]
Starting Point: The Type System for FPC:

\[
\begin{align*}
E \vdash \diamond & \quad (x : T) \in E & E \vdash A : T & \quad E, x : T \vdash B : U \\
& \quad E \vdash x : T & E \vdash \text{let } x = A \text{ in } B : U \\
E \vdash \diamond & \quad E \vdash M : T & E \vdash N : U \\
& \quad E \vdash () : \text{unit} & E \vdash M = N : \text{unit} + \text{unit} \\
E, x : T \vdash A : U & \quad E \vdash M : (T \to U) & E \vdash N : T \\
& \quad E \vdash \text{fun} x \to A : (T \to U) & E \vdash M N : U \\
E \vdash M : T & E \vdash N : U & \quad E \vdash M : (T \times U) & E, x : T, y : U \vdash A : V \\
& \quad E \vdash (M, N) : (T \times U) & E \vdash \text{let } (x, y) = M \text{ in } A : V \\
h : (T, U) & \quad E \vdash M : T & E \vdash U \\
& \quad E \vdash M : T & E \vdash h : (H, T) & E, x : H \vdash A : U & E \vdash B : U \\
& \quad E \vdash h M : U & E \vdash \text{match } M \text{ with } h x \to A \text{ else } B : U \\
inl : (T, T + U) & \quad \text{inr} : (U, T + U) & \quad \text{fold} : (T\{\mu \alpha.T/\alpha\}, \mu \alpha.T)
\end{align*}
\]
Starting Point: The Type System for FPC:

\[
\begin{align*}
E \vdash (x : T) \in E & \quad E \vdash A : T \quad E, x : T \vdash B : U \\
E \vdash x : T & \quad E \vdash \text{let } x = A \text{ in } B : U \\
E \vdash () : \text{unit} & \quad E \vdash M = N : \text{unit + unit} \\
E, x : T \vdash A : U & \quad E \vdash M : (\Pi x : T. U) \quad E \vdash N : T \\
E \vdash \text{fun } x \rightarrow A : (\Pi x : T. U) & \quad E \vdash M N : U[^{\Pi x : T. U}] \\
E \vdash M : T & \quad E \vdash N : U[^{x}] \quad E \vdash M : (A \times U) \quad E, x : T, y : U \vdash A : V \\
E \vdash (M, N) : (\Pi x : T. U) & \quad E \vdash \text{let } (x, y) = M \text{ in } A : V \\
h : (T, U) & \quad E \vdash M : T \quad E \vdash U \\
E \vdash h M : U & \quad E \vdash M : T \quad h : (H, T) \quad E, x : H \vdash A : U \quad E \vdash B : U \\
E \vdash \text{match } M \text{ with } h x \rightarrow A \text{ else } B : U \\
\text{inl} : (T, T + U) & \quad \text{inr} : (U, T + U) \quad \text{fold} : (T\{\mu \alpha. T / \alpha\}, \mu \alpha. T)
\end{align*}
\]
Rules for Formula Derivation:

\[ \text{forms}(E) \triangleq \begin{cases} 
\{ C[y/x] \} \cup \text{forms}(y : T) & \text{if } E = (y : \{ x : T \mid C \}) \\
\text{forms}(E_1) \cup \text{forms}(E_2) & \text{if } E = (E_1, E_2) \\
\emptyset & \text{otherwise}
\end{cases} \]

\[ E \vdash \circ \quad \text{fnf}(C) \subseteq \text{dom}(E) \quad \text{forms}(E) \vdash C \]

\[ E \vdash C \]

The function \text{forms}(\cdot) extracts the logical content of a typing environment.
Assume and Assert

\[
E \vdash \diamond \quad \text{fnfv}(C) \subseteq \text{dom}(E)
\]

\[
E \vdash \text{assume } C : \{\_ : \text{unit} \mid C\}
\]

\[
E \vdash C
\]

\[
E \vdash \text{assert } C : \text{unit}
\]
Rules for Refinement Types:

\[ E \vdash \{ x : T \mid C \} \quad E \vdash T <: T' \]
\[ E \vdash \{ x : T \mid C \} <: T' \]

\[ E \vdash T <: T' \quad E, x : T \vdash C \]
\[ E \vdash T <: \{ x : T' \mid C \} \]

\[ E \vdash M : T \quad E \vdash C \{ M / x \} \]
\[ E \vdash M : \{ x : T \mid C \} \]

Exercise: Derive the following subtyping rules:

\[ E \vdash T <: T' \quad E, x : \{ x : T \mid C \} \vdash C' \]
\[ E \vdash C \Rightarrow C' \]
\[ E \vdash \{ C \} <: \{ C' \} \]
Standard Rules of Subtyping:

\[
\begin{align*}
E \vdash A : T & \quad E \vdash T <: T' \\
\quad \quad E \vdash A : T' \\
E \vdash \diamond & \quad E \vdash T' <: T \\
E, x : T' \vdash U <: U' & \quad E \vdash (\Pi x : T. U) <: (\Pi x : T'. U') \\
E \vdash \text{unit} <: \text{unit} & \quad E \vdash (T + U) <: (T' + U') \\
E \vdash T <: T' & \quad E, x : T \vdash U <: U' \\
\quad \quad E \vdash (\Sigma x : T. U) <: (\Sigma x : T'. U') \\
E \vdash (\alpha <: \alpha') \in E & \quad E, \alpha <: \alpha' \vdash T <: T' \\
\quad \quad \quad \quad \alpha \notin \text{fnfv}(T') \quad \alpha' \notin \text{fnfv}(T) \\
E \vdash \alpha <: \alpha' & \quad E \vdash (\mu \alpha. T) <: (\mu \alpha'. T')
\end{align*}
\]
Rules for Restriction, I/O, and Parallel Composition:

\[
\begin{align*}
E, a \uparrow T \vdash A : U & \quad a \not\in \text{fn}(U) & E \vdash M : T & (a \uparrow T) \in E & E \vdash \Diamond & (a \uparrow T) \in E \\
E \vdash (\forall a)A : U & & & E \vdash a!M : \text{unit} & E \vdash a? : T
\end{align*}
\]

\[E, _, \{\bar{A_2}\} \vdash A_1 : T_1 \quad E, _, \{\bar{A_1}\} \vdash A_2 : T_2\]
\[E \vdash (A_1 \vdash A_2) : T_2\]

\[
\begin{align*}
(\forall a)A &= (\exists a.\bar{A}) & A_1 \vdash A_2 &= (\bar{A_1} \land \bar{A_2}) \\
\text{let } x = A_1 \text{ in } A_2 &= A_1 & \text{assume } C &= C \\
\bar{A} &= \text{True} & \text{if } A \text{ matches no other rule}
\end{align*}
\]

Exercise: Find types to typecheck the following code:

\[
a!42 \vdash (\forall c)((\text{let } x = a? \text{ in assume } \text{Sent}(x) \vdash c!x) \vdash (\text{let } x = c? \text{ in assert } \text{Sent}(x)))
\]
Type Judgements & Type safety

\[ E ::= x_1 : T_1, \ldots, x_n : T_n \]  \text{ environment}

\[ E \vdash \emptyset \]  \text{ E is syntactically well-formed}
\[ E \vdash T \]  \text{ in E, type T is syntactically well-formed}
\[ E \vdash C \]  \text{ formula C is derivable from E}
\[ E \vdash T <: U \]  \text{ in E, type T is a subtype of type U}
\[ E \vdash A : T \]  \text{ in E, expression A has type T}

\textbf{Lemma} If \( \emptyset \vdash S : T \) then S is statically safe.
\textbf{Lemma} If \( E \vdash A : T \) and \( A \Rightarrow A' \) then \( E \vdash A' : T \).
\textbf{Lemma} If \( E \vdash A : T \) and \( A \rightarrow A' \) then \( E \vdash A' : T \).

\textbf{Theorem} If \( \emptyset \vdash A : T \) then A is safe.
(For any \( A' \) and \( S \) such that \( A \rightarrow^* A' \) and \( A' \Rightarrow S \) we need that \( S \) is statically safe.)
TYPE THEORIES BEHIND RCF
Summary on RCF

- RCF supports
  - functional programming a la ML
  - concurrency in the style of process calculus, and
  - refinement types allowing correctness properties to be stated in the style of dependent type theory.

- Implementations & examples at
  [http://research.microsoft.com/F7](http://research.microsoft.com/F7)
  [http://research.microsoft.com/Fstar](http://research.microsoft.com/Fstar)

- Related language design and implementation: Aura, Fable, F7, F5, Fine, F*…
sample protocol

AUTHENTICATED RPC
Sample protocol: an authenticated RPC

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

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

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 $a$. 
Logical Specification

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

Events record the main steps of the protocol:
- $\text{Request}(a,b,s)$ before $a$ sends message 1;
- $\text{Response}(a,b,s,t)$ before $b$ sends message 2;
- $\text{KeyAB}(k,a,b)$ before issuing a key $k$ associated with $a$ and $b$;
- $\text{Bad}(a)$ before leaking any key associated with $a$.

Authentication goals are stated in terms of events:
- $\text{RecvRequest}(a,b,s)$ after $b$ accepts message 1;
- $\text{RecvResponse}(a,b,s,t)$ after $a$ accepts message 2;

where the predicates $\text{RecvRequest}$ and $\text{RecvResponse}$ are defined by

$\forall a,b,s. \text{RecvRequest}(a,b,s) \iff (\text{Request}(a,b,s) \lor \text{Bad}(a) \lor \text{Bad}(b))$

$\forall a,b,s,t. \text{RecvResponse}(a,b,s,t) \iff$

$(\text{Request}(a,b,s) \land \text{Response}(a,b,s,t)) \lor \text{Bad}(a) \lor \text{Bad}(b)$
F# Implementation

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

Our F# implementation of the protocol:

```fsharp
let mkKeyAB a b = let k = hmac.keygen() in assume (KeyAB(k,a,b)); k
let request s = concat (utf8(str "Request")) (utf8 s)
let response s t = concat (utf8(str "Response")) (concat (utf8 s) (utf8 t))

let client (a:str) (b:str) (k:keyab) (s:str) =
    assume (Request(a,b,s));
    let c = Net.connect p in
    let mac = hmacsha1 k (request s) in Net.send c (concat (utf8 s) mac);
    let (pload,mac) = iconcat (Net.recv c) in
    let t = iutf8 pload in
    hmacsha1Verify k (response s t) mac';
    assert(RecvResponse(a,b,s,t))

let server(a:str) (b:str) (k:keyab) : unit =
    let c = Net.listen p in
    let (pload,mac) = iconcat (Net.recv c) in
    let s = iutf8 pload in
    hmacsha1Verify k (request s) mac;
    assert(RecvRequest(a,b,s));
    let t = service s in
    assume (Response(a,b,s,t));
    let mac' = hmacsha1 k (response s t) in
    Net.send c (concat (utf8 t) mac')
```
The messages exchanged over TCP are:

```plaintext
Connecting to localhost:8080
Sending {BgAyICsgMj9mhJa7iDACW3Rrk...} (28 bytes)
Listening at ::1:8080
Received Request 2 + 2?
Sending {AQA0NccjcuL/WOaYS0GGtOtPm...} (23 bytes)
Received Response 4
```
1. $a \rightarrow b : \text{utf8} s \mid (\text{hmacsha1 } k_{ab}(\text{request } s))$
2. $b \rightarrow a : \text{utf8} t \mid (\text{hmacsha1 } k_{ab}(\text{response } s \ t))$
SYMBOLIC VERIFICATION:
LOGICAL INVARIANTS
FOR CRYPTOGRAPHY
Invariants for Cryptographic Structures

(1) We model cryptographic structures as elements of a symbolic algebra, e.g. $MAC(k,M)$.

(2) We use a “Public” predicate and events keep track of protocols.
   - $Pub(x)$ holds when the value $x$ is known to the adversary.
   - $Request(a,b,x)$ holds when $a$ intends to send message $x$ to $b$.

(3) We define logical invariants on cryptographic structures.
   - $Bytes(x)$ holds when the value $x$ appears in the protocol run.
   - $KeyAB(k_{ab},a,b)$ holds when key $k_{ab}$ is shared between $a$ and $b$.
   - After verifying the MAC (if no principals are compromised),
     $KeyAB(k_{ab},a,b) \land Bytes(hash \ k_{ab} \ x) \implies Request(a,b,x)$.

(4) We verify that the protocol code maintains these invariants (by typing)
   - $KeyAB(k_{ab},a,b) \land Request(a,b,x)$ is a precondition for computing $hash \ k_{ab} \ x$
Modelling Opponents as F# Programs

We program a protocol-specific interface for the opponent:

\[
\text{let } \text{setup } (a: \text{str}) \ (b: \text{str}) = \\
\text{let } k = \text{mkKeyAB } a \ b \ \text{in} \\
(\text{fun } s \rightarrow \text{client } a \ b \ k \ s), \\
(\text{fun } \_ \rightarrow \text{server } a \ b \ k), \\
(\text{fun } \_ \rightarrow \text{assume } (\text{Bad}(a)); \ k), \\
(\text{fun } \_ \rightarrow \text{assume } (\text{Bad}(b)); \ k)
\]

**Opponent Interface (excerpts):**

```fsharp
val \text{send}: \text{conn } \rightarrow \text{bytespub } \rightarrow \text{unit} \\
val \text{recv}: \text{conn } \rightarrow \text{bytespub} \\
val \text{hmacsha1}: \text{keypub } \rightarrow \text{bytespub } \rightarrow \text{bytespub} \\
val \text{hmacsha1Verify}: \text{keypub } \rightarrow \text{bytespub } \rightarrow \text{bytespub } \rightarrow \text{unit} \\
val \text{setup}: \text{strpub } \rightarrow \text{strpub} \rightarrow \\
\quad (\text{strpub } \rightarrow \text{unit}) \ast (\text{unit } \rightarrow \text{unit}) \ast (\text{unit } \rightarrow \text{keypub}) \ast (\text{unit } \rightarrow \text{keypub})
```
Symbolic Security Theorem

An expression is *semantically safe* when every executed assertion logically follows from previously-executed assumptions.

Let $I_L$ be the opponent interface for our library.
Let $I_R$ be the opponent interface for our protocol (the *setup* function).
Let $X$ be composed of library and protocol code.

**Theorem 1 (Authentication for the RPC Protocol)**

*For any opponent $O$, if $I_L, I_R ⊨ O : \text{unit}$, then $X[O]$ is semantically safe.*
Symbolic proof (typechecking)

To apply the authentication theorem, we typecheck our protocol code against the library interface.

For MACs, this interface is

**Refinement Types for MACs in the Crypto library:**

```plaintext
private val hmac_keygen: unit → k:key{MKey(k)}
val hmac_sha1:
  k:key →
  b:bytes{ (MKey(k) ∧ MACSays(k, b)) ∨ (Pub(k) ∧ Pub(b)) } →
  h:bytes{ IsMAC(h, k, b) ∧ (Pub(b) ⇒ Pub(h)) } }
val hmac_sha1_verify:
  k:key{MKey(k) ∨ Pub(k)} → b:bytes → h:bytes → unit{IsMAC(h, k, b)}
```

∀h,k,b. IsMAC(h,k,b) ∧ Bytes(h) ⇒ MACSays(k,b) ∨ Pub(k)
Security Proof: MACs

To apply the authentication theorem, we typecheck our protocol code against the library interface.

For MACs, this interface is

Refinement Types for MACs in the **Crypto** library:

```plaintext
private val hmac_keygen: unit → k:key{MKey(k)}
val hmacsha1:
  k:key →
  b:bytes{ (MKey(k) ∧ MACSays(k,b)) ∨ (Pub(k) ∧ Pub(b)) } →
  h:bytes{ IsMAC(h,k,b) ∧ (Pub(b) ⇒ Pub(h)) }
val hmacsha1Verify:
  k:key{MKey(k) ∨ Pub(k)} → b:bytes → h:bytes → unit{IsMAC(h,k,b)}
```

(C1. By expanding the definition of IsMAC)
∀h,k,b. IsMAC(h,k,b) ∧ Bytes(h) ⇒ MACSays(k,b) ∨ Pub(k)
(C2. MAC keys are public iff they may be used with any logical payload)
∀k. MKey(k) ⇒ (Pub(k) ⇔ ∀m. MACSays(k,m))
Our F# implementation of the protocol:

```fsharp
let mkKeyAB a b = let k = hmac.keygen() in assume (KeyAB(k,a,b)); k
let request s = concat (utf8(str "Request")) (utf8 s)
let response s t = concat (utf8(str "Response")) (concat (utf8 s) (utf8 t))

let client (a:str) (b:str) (k:keyab) (s:str) =
    assume (Request(a,b,s));
    let c = Net.connect p in
    let mac = hmacsha1 k (request s) in
    Net.send c (concat (utf8 s) mac);
    let (pload,mac) = iconcat (Net.recv c) in
    let t = iutf8 pload in
    hmacsha1Verify k (request s) mac;
    assert(RecvRequest(a,b,s,t));
    let mac' = hmacsha1 k (response s t) in
    Net.send c (concat (utf8 t) mac')

let server(a:str) (b:str) (k:keyab) : unit =
    let c = Net.listen p in
    let (pload,mac) = iconcat (Net.recv c) in
    let s = iutf8 pload in
    hmacsha1Verify k (request s) mac;
    assert(RecvRequest(a,b,s,t));
    let t = service s in
    assume (Response(a,b,s,t));
    let mac' = hmacsha1 k (response s t) in
    Net.send c (concat (utf8 t) mac')
```
**Message formats**

*Requested* and *Responded* are (typechecked) postconditions of *request* and *response*.

Typechecking involves verifying that they are injective and have disjoint ranges. (Verification is triggered by asserting the formulas below, so that Z3 proves them.)

**Properties of the Formatting Functions *request* and *response***:

(request and response have disjoint ranges)
\[
\forall v,v',s,s',t'. (\text{Requested}(v,s) \land \text{Responded}(v',s',t')) \Rightarrow (v \neq v')
\]

(request is injective)
\[
\forall v,v',s,s'. (\text{Requested}(v,s) \land \text{Requested}(v',s') \land v = v') \Rightarrow (s = s')
\]

(response is injective)
\[
\forall v,v',s,s',t,t'.
\ (\text{Responded}(v,s,t) \land \text{Responded}(v',s',t') \land v = v') \Rightarrow (s = s' \land t = t')
\]

For typechecking the rest of the protocol, we use only these formulas: the security of our protocol does not depend a specific format.
Security proof: protocol invariants

Formulas Assumed for Typechecking the RPC protocol:

\[(\text{KeyAB MACSays})\]
\[\forall a,b,k,m. \text{KeyAB}(k,a,b) \Rightarrow (\text{MACSays}(k,m) \Leftrightarrow (\exists s. \text{Requested}(m,s) \land \text{Request}(a,b,s)) \lor (\exists s,t. \text{Responded}(m,s,t) \land \text{Response}(a,b,s,t)) \lor (\text{Bad}(a) \lor \text{Bad}(b))))\]

\[(\text{KeyAB Injective})\]
\[\forall k,a,b,a',b'. \text{KeyAB}(k,a,b) \land \text{KeyAB}(k,a',b') \Rightarrow (a=a') \land (b=b')\]

\[(\text{KeyAB Pub Bad})\]
\[\forall a,b,k. \text{KeyAB}(k,a,b) \land \text{Pub}(k) \Rightarrow \text{Bad}(a) \lor \text{Bad}(b)\]

\[(\text{KeyAB MACSays})\] is a definition for the library predicate \text{MACSays}. It states the intended usage of keys in this protocol.

\[(\text{KeyAB Injective})\] is a theorem: each key is used by a single pair of principals.

\[(\text{KeyAB Pub Bad})\] is a theorem: each key is secret until one of its owners is compromised.
Security proof: protocol invariants

Formulas Assumed for Typechecking the RPC protocol:

(\text{KeyAB MACSays})
\forall a, b, k, m. \text{KeyAB}(k, a, b) \Rightarrow (\text{MACSays}(k, m) \Leftrightarrow
( (\exists s. \text{Requested}(m, s) \land \text{Request}(a, b, s)) \lor
(\exists s, t. \text{Responded}(m, s, t) \land \text{Response}(a, b, s, t)) \lor
(\text{Bad}(a) \lor \text{Bad}(b))))

(\text{KeyAB Injective})
\forall k, a, b, a', b'. \text{KeyAB}(k, a, b) \land \text{KeyAB}(k, a', b') \Rightarrow (a=a') \land (b=b')

(\text{KeyAB Pub Bad})
\forall a, b, k. \text{KeyAB}(k, a, b) \land \text{Pub}(k) \Rightarrow \text{Bad}(a) \lor \text{Bad}(b)

Using these assumptions, F7 typechecks our protocol code. This automatically completes our protocol verification.
SYMBOLIC THEORY

SEMANTIC SAFETY BY TYPING
Syntactic vs Semantic Safety

- Two variants of run-time safety: “all asserted formulas follow from previously-assumed formulas”
  - Either by **deducibility**, enforced by typing (the typing environment contains less assumptions than those that will be present at run-time)
  - Or in **interpretations** satisfying all assumptions

- We distinguish different kinds of logical properties
  - Inductive definitions (Horn clauses)
    \[ \forall x, y. \ Pub(x) \land \ Pub(y) \Rightarrow \ Pub(pair(x,y)) \]
  - Logical theorems additional properties that hold in our model
    \[ \forall x, y. \ Pub(pair(x,y)) \Rightarrow \ Pub(x) \]
  - Operational theorems additional properties that hold at run-time
    \[ \forall k, a, b. \ PubKey(k,a) \land PubKey(k,b) \Rightarrow a = b \]

- We are interested in **least models** for inductive definitions (not all models)
- After proving our theorems (by hand, or using other tools), we can **assume** them so that they can be used for typechecking
Refined Modules

- Defining cryptographic structures and proving theorems is hard... Can we do it once for all?
- A “refined module” is a package that provides
  - An F7 interface, including inductive definitions & theorems
  - A well-typed implementation

**Theorem:** refined modules with disjoint supports can be composed into semantically safe protocols

- We show that our crypto libraries are refined modules (defining e.g. Pub)
- To verify a protocol that use them, it suffices to show that the protocol itself is a refined module, assuming all the definitions and theorems of the libraries.
APPLICATIONS

SOME Refined MODULES
Some Refined Modules

- **Crypto**: a library for basic cryptographic operations
  - Public-key encryption and signing (RSA-based)
  - Symmetric key encryption and MACs
  - Key derivation from seed + nonce, from passwords
  - Certificates (x.509)

- **Principals**: a library for managing keys, associating keys with principals, and modelling compromise
  - Between Crypto and protocol code, defining user predicates on behalf of protocol code
  - Higher-level interface to cryptography
  - Principals are units of compromise (not individual keys)

- **XML**: a library for XML formats and WS* security
**Cryptographic Patterns**

**Patterns** is a refined module that shows how to derive authenticated encryption, for each of the three standard composition methods for encryption and MACs.

**Encrypt-then-MAC (as in IPSEC in tunnel mode):**

\[ a \rightarrow b: \quad e \ | \ \text{hmacsha1} \ k_{ab}^m e \ \text{where} \ e = \text{aes} \ k_{ab}^e t \]

**MAC-then-Encrypt (as in SSL/TLS):**

\[ a \rightarrow b: \quad \text{aes} \ k_{ab}^e (t \ | \ \text{hmacsha1} \ k_{ab}^m t) \]

**MAC-and-Encrypt (as in SSH):**

\[ a \rightarrow b: \quad \text{aes} \ k_{ab}^e t \ | \ \text{hmacsha1} \ k_{ab}^m t \]
Cryptographic Patterns:  
Hybrid Encryption

Hybrid encryption implements public-key encryption for large plaintexts:

1. generate a fresh symmetric key;
2. use it to encrypt the plaintext;
3. encrypt the key using the public key of the intended receiver.

**Hybrid Encryption:**

\[ a \rightarrow b: \text{rsa-oaep} \: pk_b \: k_{ab} | \text{aes} \: k_{ab} \: t \]

We combine authenticated asymmetric encryption (RSA-OAEP) with unauthenticated symmetric encryption, and provide unauthenticated asymmetric encryption.

Verification relies on the single usage of the symmetric key.
Hybrid Encryption API:

\[
\text{val } \text{hybrid\_keygen: unit } \rightarrow \text{(pk: key } \ast \text{ sk: key)} \\
\{ \text{HyPubKey}(pk) \land \text{HyPrivKey}(sk) \land \text{PubPrivKeyPair}(pk,sk) \} \\
\text{val } \text{hybridEncrypt: k: key } \rightarrow \text{ b: bytes} \\
\{ (\text{HyPubKey}(k) \land \text{CanHyEncrypt}(k,b)) \lor (\text{Pub}(k) \land \text{Pub}(b)) \} \rightarrow \text{ e: bytes } \{ \text{IsHyEncryption}(e,k,b) \} \\
\text{val } \text{hybridDecrypt: sk: key } \rightarrow \text{ e: bytes } \{ \text{HyPrivKey}(sk) \lor (\text{Pub}(sk) \land \text{Pub}(e)) \} \rightarrow \text{ b: bytes } \{ (\forall pk,x. (\text{PubPrivKeyPair}(pk,sk) \land \text{IsHyEncryption}(e,pk,x)) \Rightarrow x = b) \land (\text{Pub}(sk) \Rightarrow \text{Pub}(b)) \} \\
\]

Verification relies on the single usage of the symmetric key.

Hybrid Encryption Key Usage:

\( \text{(HyPubKey CanAsymEncrypt)} \) \\
\( \forall pk,kb. \text{HyPubKey}(pk) \land \text{HySymKey}(\text{SymKey}(kb),pk) \Rightarrow \text{CanAsymEncrypt}(pk,kb) \)

\( \text{(HySymKey CanSymEncrypt)} \) \\
\( \forall pk,k,b. \text{HySymKey}(k,pk) \land \text{CanHyEncrypt}(pk,b) \Rightarrow \text{CanSymEncrypt}(k,b) \)