MPRI 2-30: Automated Verification of Cryptographic Protocol Implementations

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

Spring, 2014
TLS Handshakes
The Handshake

- Two linked sub-protocols
  - Negotiates protocol version, handshake method and algorithms, authenticated encryption method and algorithms
  - Authenticates peers from their certificates
  - Derives connection keys

- Full handshake takes up to 3 rounds with 11 messages

- Abbreviated handshake often possible
Full handshake

ClientHello

-----> ServerHello

Certificate*
ServerKeyExchange*
CertificateRequest*
ServerHelloDone

Certificate*
ClientKeyExchange
CertificateVerify*
ChangeCipherSpec
Finished  

<--------

Application Data

<--------

Application Data

<--------

ChangeCipherSpec
Finished
The key exchange messages are used to compute shared pre-master-secret (PMS) then master-secret (MS) for the session.

The MS and (Cr,Sr) are used to (1) derive fresh connection keys and (2) authenticate the handshake digests in Finished messages.
Provided the client and server already share a session sId, they can use its pre-established master secret and (Cr, Sr) to derive fresh connection keys.

This saves one round trip & any public-key cryptography.

Otherwise the server continues with a full handshake (picking some fresh sId).
The client samples a fresh pms (mostly) at random

Here * stands for “if Client auth”, at the server initiative (prescribing “for signing” as X.509 attributes)
DHE full handshake

ClientHello [Cr] -------> ServerHello [Sr]
Certificate chain [vk | fixed p g g^y]
ServerKeyExchange
[p g g^y sig(Cr Sr p g g^y)_vk]
CertificateRequest*
[for signing | fixed]
ServerHelloDone

<--------
Certificate* chain [for signing | fixed p g g^x]
ClientKeyExchange [fixed ? empty : g^x]
CertificateVerify* [sig(Clog')] ChangeCipherSpec
Finished MAC(Clog) -------> ChangeCipherSpec
Finished MAC(Slog)
Application Data <-------->
Application Data

E stands for server-ephemeral; client-ephemeral ⇔ “for signing”
Here * stands for “if Client auth”, at the server initiative (prescribing “for
signing”, “fixed”, or both, as X.509 attributes)
..._DSA and ..._RSA only affect X.509 certs
EC... should only affects the crypto parameters
Diffie-Hellman Key Exchange

- A fundamental cryptographic algorithms [1976]

Alice

let x = sample q
let X = g^x

agree on public parameters:
p prime, g generator of Z/pZ^*, q = |Z/pZ*|

let y = sample q
let Y = g^y

exchange X & Y

let Z = Y^x

let Z = X^y

now sharing Z = g^{x*y}
we can derive keys as PRF(Z,...)

- Secure against passive adversaries;
  otherwise we must authenticate X and Y
- Many variants: STS, ISO, MQV, SIGMA
- Many implementations: SSH, IPsec, Kerberos
Decisional Diffie-Hellman

• The Decisional Diffie-Hellman assumption: the probability of distinguishing between the exponentials of a DH exchange and its idealized variant is negligible

agree on public parameters: 
\[ p \text{ prime, } g \text{ generator of } \mathbb{Z}/p\mathbb{Z}^*, \quad q = |\mathbb{Z}/p\mathbb{Z}^*| \]

Concrete

\[
\text{let } x = \text{sample } q \\
\text{let } y = \text{sample } q \\
( g^x, g^y, g^{x\cdot y} )
\]

Ideal

\[
\text{let } x = \text{sample } q \\
\text{let } y = \text{sample } q \\
\text{let } z = \text{sample } q \\
( g^x, g^y, g^z )
\]

• Application: El Gamal encryption is CPA

\[
\text{let GEN()} = \\
\text{let } x = \text{sample } q \\
( x, g^x )
\]

\[
\text{let ENC* } X \ m = \\
\text{let } y = \text{sample } q \\
\text{let } z = \text{sample } q \\
( g^y, g^z \cdot m )
\]
The Handshake, ideally

• Our interface abstracts over many details of the Handshake
  – Handshake messages and their formats
  – Certificate formats and public key infrastructure
  – Database of past sessions, available for abbreviated handshakes

• A key index is *safe* when
  – Its ciphersuite is cryptographically strong; and
  – Its peer authentication materials are trustworthy
e.g. the private key for the peer certificate
  is used only by compliant handshake sessions

• For instances with safe indexes, the (typed) idealized handshake
  – Generates fresh abstract keys instead of calling the concrete KDF
  – Drops “Complete” notifications not preceded by a Finished
    with matching parameters sent by a compliant peer instance.
Internal interface for Handshake & CCS protocols

- New keys are delivered before handshake completion
- Negotiated parameters can be read off the state
- Refinements imply precise matching conversations

```haskell
type (;r:role, o:config) state // for each local instance of the protocol
type (;ki:KeyInfo) fragment // content type for the Handshake protocol
type (;ki:KeyInfo) ccs // content type for the Handshake protocol

// Control Interface
val init: r:role -> o:config -> (;r ,o) state
val resume: si:SessionInfo -> o:config -> (;Client, o) state
val rehandshake: (;Client, idle) state -> o:config -> (;Client, o) state
val rekey: (;Client, idle) state -> o:config -> (;Client, o) state
val request: (;Server, idle) state -> o:config -> (;Server, o) state

// Network Interface (output)
type (;r:role, o:config, ki:KeyInfo) outgoing =
  OutFragment of (;r, o) state * (;ki) fragment option
  OutCCS of s:;(r, o) state * (;ki) ccs * (;OutKi(s)) ccs_data
  OutComplete of s:;(r, o) state {Complete(r, o, s)}
  ...
val nextFragment: r:role -> o:config -> ki:KeyInfo ->
  (;r, o) state -> (;r, o, ki) outgoing

// Network Interface (input)
type (;r:role, o:config) incoming =
  InTLSVersion of (;r, o) state * ProtocolVersion
  InComplete of s:;(r, _) state {Complete(r, o, s)}
  ...
val recvFragment: r:role -> o:config -> ki:KeyInfo ->
  (;r, o) state -> (;ki) fragment -> (;r, o) incoming
val recvCCS: r:role -> o:config -> ki:KeyInfo ->
  (;r, o) state -> (;ki) ccs -> s:;(r, o) state * (;InKi(s)) ccs_data
```

• New keys are delivered before handshake completion
• Negotiated parameters can be read off the state
• Refinements imply precise matching conversations
Snap Start & False Start

Two proposed extensions trading speed for security:

- No need to negotiate if both client & server are jointly configured
- The server cannot enforce its own security policy:
  - a client contacts a secure server; the server policy demands AES
  - an active adversary suggests using RC4 instead (accepted by the client, but not the server)
  - the client starts sending secrets shared with the server
  - the adversary logs all traffic, decrypts in a few years’ time
Main TLS API
The TLS API & ideal functionality

- Our API is similar but more informative than mainstream APIs
  - We run on the caller’s thread, letting the application do the scheduling & multiplexing
  - We give more control to the application code, and reflect more information from the underlying TLS state (lengths, fragmentation, authorization queries)
    - More precise security theorems
    - More flexibility for experiments & testing

- We can implement safe & simple APIs on top of it
- Sample applications using our API
  - Secure RPCs (with one connection per call)
  - Password-based client authentication
  - Basic HTTPS clients and servers (for interoperability testing)
Each application provides its own plaintext module for data streams:

- Typing ensures secrecy and authenticity at safe indexes.
- Parameters select ciphersuites and certificates.
- Results provide detailed information on the protocol state.

```scala
// our main TLS API (outline)

// for each local instance of the protocol

// creating new client and server instances
val connect: TcpStream -> params -> (;Client) nullCn Result
val accept: TcpStream -> params -> (;Server) nullCn Result

// triggering new handshakes, and closing connections
val rehandshake: c:cn{Role(c)=Client} -> cn Result
val request: c:cn{Role(c)=Server} -> cn Result
val shutdown: c:cn -> TcpStream Result

// writing data

// reading data
```
Main result: concrete TLS and ideal TLS are indistinguishable.

Our typed ideal API for TLS thus yields application security by typing.

<table>
<thead>
<tr>
<th>Component</th>
<th>F# (LOC)</th>
<th>F7 (LOC)</th>
<th>F7 (S)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Base</td>
<td>945</td>
<td>581</td>
<td>11</td>
</tr>
<tr>
<td>TLS Record</td>
<td>826</td>
<td>511</td>
<td>77</td>
</tr>
<tr>
<td>Handshake/CCS</td>
<td>2 400</td>
<td>777</td>
<td>413</td>
</tr>
<tr>
<td>Alert Protocol</td>
<td>184</td>
<td>119</td>
<td>105</td>
</tr>
<tr>
<td>AppData Protocol</td>
<td>139</td>
<td>113</td>
<td>34</td>
</tr>
<tr>
<td>TLS API</td>
<td>640</td>
<td>426</td>
<td>309</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>5 134</strong></td>
<td><strong>2 527</strong></td>
<td><strong>949</strong></td>
</tr>
</tbody>
</table>
Main result:
concrete TLS and
ideal TLS are
indistinguishable

Our typed ideal API
for TLS thus yields
application security
by typing

<table>
<thead>
<tr>
<th></th>
<th>LOC</th>
<th>F7 (LOC)</th>
<th>F7 (S)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Base</td>
<td>945</td>
<td>581</td>
<td>11</td>
</tr>
<tr>
<td>TLS Record</td>
<td>826</td>
<td>511</td>
<td>77</td>
</tr>
<tr>
<td>Handshake/CCS</td>
<td>2400</td>
<td>777</td>
<td>413</td>
</tr>
<tr>
<td>Alert Protocol</td>
<td>184</td>
<td>119</td>
<td>105</td>
</tr>
<tr>
<td>AppData Protocol</td>
<td>139</td>
<td>113</td>
<td>34</td>
</tr>
<tr>
<td>TLS API</td>
<td>640</td>
<td>426</td>
<td>309</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td>5134</td>
<td>2527</td>
<td>949</td>
</tr>
</tbody>
</table>
“Untyped” ideal API for TLS

Cryptographic Provider
- Cryptographic assumptions

Bytes, Network lib.fs

Ideal/Concrete stream.fs

our verified modular TLS implementation
- TLS.fs

any typed F# program

TLS proxy with bytes-only API
- TLS_Adv.fs

any typed F# program

plain typed interface (attacker model)

Indistinguishability for all PPT adversaries
Security for RPC over TLS

- Cryptographic Provider
  - Cryptographic assumptions
- our verified modular TLS implementation
- Bytes, Network lib.fs
- RPC Payload plain.fs
- TLS.fs
- any typed F# program
- active adversaries
- application code
- RPC rpc.fs
- RpcAdv.fs
- any typed F# program
- any typed F# program
We run clients against an OpenSSL 1.0.1e server for various ciphersuites

- How many handshakes per second?
- How much data transferred per second?

<table>
<thead>
<tr>
<th>KEX</th>
<th>Ciphersuite</th>
<th>MAC</th>
<th>F# (BC)</th>
<th>OpenSSL</th>
<th>Oracle</th>
<th>JSSE</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>HS/s</td>
<td>MiB/s</td>
<td>HS/s</td>
<td>MiB/s</td>
</tr>
<tr>
<td>RSA</td>
<td>RC4</td>
<td>MD5</td>
<td>305.25</td>
<td>30.17</td>
<td>292.04</td>
<td>226.51</td>
</tr>
<tr>
<td>RSA</td>
<td>RC4</td>
<td>SHA</td>
<td>291.37</td>
<td>27.85</td>
<td>288.74</td>
<td>232.42</td>
</tr>
<tr>
<td>RSA</td>
<td>3DES</td>
<td>SHA</td>
<td>267.09</td>
<td>8.40</td>
<td>283.04</td>
<td>22.95</td>
</tr>
<tr>
<td>RSA</td>
<td>AES128</td>
<td>SHA</td>
<td>278.71</td>
<td>18.54</td>
<td>285.35</td>
<td>234.41</td>
</tr>
<tr>
<td>RSA</td>
<td>AES128</td>
<td>SHA256</td>
<td>278.71</td>
<td>16.50</td>
<td>281.92</td>
<td>128.33</td>
</tr>
<tr>
<td>RSA</td>
<td>AES256</td>
<td>SHA</td>
<td>291.37</td>
<td>16.86</td>
<td>282.89</td>
<td>204.47</td>
</tr>
<tr>
<td>RSA</td>
<td>AES256</td>
<td>SHA256</td>
<td>267.09</td>
<td>15.16</td>
<td>307.72</td>
<td>119.42</td>
</tr>
<tr>
<td>DHE</td>
<td>3DES</td>
<td>SHA</td>
<td>20.16</td>
<td>8.37</td>
<td>58.07</td>
<td>22.99</td>
</tr>
<tr>
<td>DHE</td>
<td>AES128</td>
<td>SHA</td>
<td>20.41</td>
<td>18.59</td>
<td>57.06</td>
<td>244.30</td>
</tr>
<tr>
<td>DHE</td>
<td>AES128</td>
<td>SHA256</td>
<td>19.99</td>
<td>16.45</td>
<td>58.33</td>
<td>128.34</td>
</tr>
<tr>
<td>DHE</td>
<td>AES256</td>
<td>SHA</td>
<td>20.29</td>
<td>16.72</td>
<td>56.83</td>
<td>203.01</td>
</tr>
<tr>
<td>DHE</td>
<td>AES256</td>
<td>SHA256</td>
<td>20.16</td>
<td>14.86</td>
<td>59.52</td>
<td>120.96</td>
</tr>
</tbody>
</table>
An Implementation of TLS with Verified Cryptographic Security

Our ideal API provides strong, modular, usable, conditional application security by typing.

We trust

- **automated typechecking**: F7 and Z3
  - Now: mechanized type theory
  - Next: certified typechecker (F*, POPL’12) and SMT solver
- **cryptographic assumptions**, with handwritten proofs
  - Next: better concrete reductions, with tighter bounds
  - Next: mechanized proofs a la Certicrypt & Crypter
- **the F# compiler and runtime**: Windows and .NET
- **core cryptographic providers**
  - Next: correctness proofs for selected algorithms (elliptic curves)

We account for some side-channels, but not for timing analysis
Analyzing TLS Implementations with ProVerif/CryptoVerif
Protocols and Analyses

Symbolic Analyses
- Casper
- Cryptyc, F7
- AVISPA
- Scyther
- ProVerif ('01)

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

Computational Analyses
- Hand Proofs
- CryptoVerif ('06)

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

Symbolic Analyses
- Casper
- Cryptyc, F7
- AVISPA
- Scyther
- ProVerif ('01)

Computational Analyses
- CryptoVerif ('06)

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

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

Applications

Protocol Code

Crypto, Net
Concrete Libraries

Compile

Other Implementations

Network

Interoperability Testing

Run
Verifying Protocol Implementations

- Applications
- Protocol Code
  - Concrete Libraries: Crypto, Net
  - Symbolic Libraries: Crypto, Net
- Security Goals
- Other Implementations
- Network
  - Interoperability Testing
  - Symbolic Debugging
  - Symbolic Verification
  - Compile
  - Compile
  - Verify
    - Attack
    - No Attack
    - Diverges
Verifying Protocol Implementations

- Applications
- Protocol Code
- Security Goals
- Computational Crypto Model
- Crypto, Net Concrete Libraries
- Crypto, Net Symbolic Libraries
- Other Implementations
- Computational Crypto Model
- Network
- Interoperability Testing
- Symbolic Debugging
- Symbolic Verification
- Verify
- No Proof
- Attack
- Diverges
- No Attack
- Proof
Proof Technique

Compiling F# to the Pi Calculus
An Applied Pi Calculus

- Source language of the ProVerif theorem prover

Processes of the $\pi$ calculus:

\[
P, Q, R ::= \quad \text{process} \\
\text{out } M(N) \quad \text{asynchronous output of } N \text{ on channel } M \\
\text{in } M(x); P \quad \text{input of } x \text{ from channel } M \text{ (} x \text{ has scope } P \text{)} \\
!\text{in } M(x); P \quad \text{replicated input} \\
\text{new } x; P \quad \text{fresh generation of name } x \text{ (} x \text{ has scope } P \text{)} \\
P \parallel Q \quad \text{parallel composition of } P \text{ and } Q \\
0 \quad \text{inactivity} \\
\text{let } \vec{x} = D \text{ in } P \text{ else } Q \quad \text{bind results of } D \text{ to } \vec{x} \text{ in } P, \text{ or else run } Q \\
\text{begin } L \quad \text{begin-event labelled } L \\
\text{end } L \quad \text{end-event labelled } L
\]
Functions as Processes

Milner’s Call-By-Value Continuation-Passing Translation from \( \lambda \) to \( \pi \):

\[
\begin{align*}
[[x]]k & \triangleq \textbf{out } k(x) \\
[[\lambda x.e]]k & \triangleq \textbf{new } f; \ (\textbf{out } k(f) \ | \ \textbf{in } f(\langle x, k' \rangle); [[e]]k') \\
[[e_1 \ e_2]]k & \triangleq \textbf{new } k_1; ([[e_1]]k_1 \ | \ \textbf{in } k_1(f); \textbf{new } k_2; ([[e_2]]k_2 \ | \ \textbf{in } k_2(x); \textbf{out } f(\langle x, k \rangle)))
\end{align*}
\]

This is the core of model extraction, but additionally we perform transformations to speed up verification.
How to compile a function?

- Our tool specifically targets symbolic verification, with many optimization to help ProVerif converge
  - Complete inlining (anticipating resolution)
  - + Dead Code Elimination

- We select a translation for each function
  - Pure non-recursive functions are compiled to term reductions (as supported by ProVerif)
  - Pure recursive functions are compiled to predicate declarations (logic programming)
  - Functions with side-effects are compiled to Pi Processes

- The generated reduction, predicate, or process is declared public or private depending on whether it is in the interface
Compiling a Function

Consider the F# function

```fsharp
let mac nonce pwd text =
    Crypto.hmacsha1 nonce (concat (utf8 pwd) (utf8 text))
```

We can translate it as a process

```fsharp
!in(mac, (nonce,pwd,text,k));
  out(k,Hmacsha1(nonce,Concat(Utf8(pwd),Utf8(text))))
```

We actually translate `mac` into a ProVerif reduction rule:

```fsharp
reduc mac(nonce,pwd,text) =
    HmacSha1(nonce,Concat(Utf8(pwd),Utf8(text)))
```
Protocol Model in ProVerif

Crypto Model

Secrets and Channels

Pi Calculus Process for A

Pi Calculus Process for B

Full Protocol

data concat/2.
fun hmacsha1/2.
fun pk/1.
fun rsaencrypt/2.
reduce rsaencrypt(k,rsaencrypt(pk(k),m)) = m.

private free pwdA.
private free skB.
free netChan.
free text.

let alice =
  new nonce;
  event AliceSent(text);
  out (netChan, concat(hmacsha1(nonce,concat(pwdA,text)),
          concat(rsaencrypt(pk(skB),nonce),
          text))).

let bob =
in (netChan, x);
let concat(mac,concat(enc,text)) = x in
let nonce = rsaencrypt(skB,enc) in
if mac = hmacsha1(nonce,concat(pwdA,text)) then
  event BobAccepts(text)
else 0.

process
  out (netChan,pk(skB)); (alice | bob)
Soundness of our translation

**Theorem 1 (Reflection of Robust Safety)**

If $S_0 :: I_{pub}$ and $[S_0 :: I_{pub}]$ is robustly safe for $q$ (in the pi calculus) then $S_0$ is robustly safe for $q$ and $I_{pub}$ (in F#)

- $S_0$ is the series of modules that define our system;
- $I_{pub}$ is the list of values and functions of $S_0$ available to the attacker;
- $q$ is our target security query; and
- $[S_0 :: I_{pub}]$ is the ProVerif script compiled from $S_0$ and $I_{pub}$.

To verify that $S_0$ is robustly safe for $q$ and $I_{pub}$,
1. we run ProVerif on $[S_0 :: I_{pub}]$ with query $q$;
2. if ProVerif completes successfully, we apply Theorem 1.

The proof relies on an operational correspondence between reductions on F configurations and reductions in the pi calculus.
Protocol Verification with ProVerif

Message Authentication

Password Secrecy

```plaintext
Protocol Verification with ProVerif
```

```plaintext
Message Authentication

Password Secrecy
```
Protocol Verification with ProVerif

- If a property is false, ProVerif exhibits the counter-example as an attack
  - E.g. Suppose A does not include text in the HMACSHA1

![Attack Trace](image)
Limitations of fs2pv and ProVerif

- We cannot write higher-order functions
  - Upcoming version will allow higher-order functions as long as they do not appear in interfaces

- We cannot use some built-in types such as refs, arrays
  - We use channels instead to store/retrieve data
  - Upcoming version will allow limited references

- To use additional libraries, such as System.SQL or List, the user must code up symbolic implementations
  - We provide implementations for Crypto, Net, Prins, XML

- We avoid recursive and stateful functions
  - Recursive functions often lead to non-terminating analyses
  - Stateful functions often take a lot of memory to verify
Verifying TLS

Transport Layer Security Protocol (TLS 1.0)
– Widely-deployed industrial protocol
– Well-understood, with detailed specs
– Good benchmark for analysis techniques

This work:
• A reference implementation in F#
• Symbolic verification
• Computational verification
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
Our Reference Implementation in F#

We implement a *subset of TLS* as a modular library (9700 LOC)
- Supports SSL3.0, TLS1.0, TLS1.1 (with session resumption)
- Supports any ciphersuite using DES, AES, RC4, SHA1, MD5
- Server-only authentication, RSA mode only
- No compression, fragmentation, or alerts

We test it using three basic *sample applications*:
- An HTTPS client that can retrieve pages from any web server
  (tested against IIS, Apache, and our F# server)
- An HTTPS server that can serve pages to any web client
  (tested against IE, Firefox, Opera, and our F# client)
- A client-server application that performs password-based authentication over TLS
Implementation Structure

- 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
## APIs & Sample Client

### HTTPS_Client.fs (application code)

```ocaml
let client_request url =
  let netconn = Net.connect url in
let connid, sessionid = Handshake.connect netconn url in
let request = httpRequest url in
Record.send connid request;
let response = Record.recv connid in
response
```

### Record.fsi (interface)

```ocaml
type ConnectionId = bytes
val send: ConnectionId -> bytes -> unit
val recv: ConnectionId -> bytes

(* private *)
type Connection = {
  net_conn: Net.conn;
  crt_version: ProtocolVersion;
  write: ConnectionState;
  read: ConnectionState
}
val getConnection: ConnectionId -> Connection
```

### Handshake.fsi (interface)

```ocaml
type SessionId = bytes
val connect: Net.conn -> ServerName -> ConnectionId * SessionId
val accept: Net.conn -> CertName -> ConnectionId * SessionId
val resume: Net.conn -> SessionId -> ConnectionId * SessionId
val close: ConnectionId -> unit

(*private *)
type Session = { sid: SessionId;
  ms: bytes;
  serverCertificate: bytes;
}
val getSession: SessionId -> Session
```
Record Module

Record protocol
(informal narration)

A → B : {m, [m]_{ak}}_{ek}
Implementing TLS

• The RFCs do not specify an API
  – Implementations are free to choose the interface

• Certificate management and verification is tricky

• Testing standard conformance and interop hard
  – Multiple versions and ciphersuites
  – Poor testing tools
  – But results in good insights (e.g. understanding rollback issues) and ensures a very precise model
Symbolic Verification

- **Write F# symbolic implementation for libraries**
  - **Crypto**: cryptographic operations are algebraic constructors/destructors
  - **Net**: networking functions defined using Pi calculus channels
  - **Prins**: database of (server) principals, each with a certificate and private key

- **Declare capabilities for active attackers (in Dolev-Yao style)**
  - **Crypto, Net, Prins**: can perform crypto, control the network, compromise principals
  - **Handshake**: can open (and resume) connections with any clients and servers
  - **Record**: can send/recv application messages on any open connections

- **Run** `fs2pv` tool [BFGT’06] **to generate ProVerif script**
  - from F# code for TLS, for symbolic libraries + F# interface for attacker + security goals

- **Run** ProVerif [Blanchet’01] **to verify security goals against any attacker with access to this interface**
  - security guarantees for any number of clients and servers running any number of Handshake, Resumption, and Record instances
Symbolic Verification Goals

**Record Message Authentication**
If the client receives message $p$ over connection $c$,
then either the server has sent $p$ over $c$,
or the server has been compromised (and similarly in the other direction)

**Record Payload Secrecy**
If the attacker obtains a fresh message $p$ sent over connection $c$
then either the client or the server has been compromised.

**Handshake Authentication**
At the end of the handshake,
the client and the server agree on all negotiated parameters and keys,
provided that the client and the server are not compromised.

**Key Secrecy**
If the attacker obtains a connection key,
then either the client or the server has been compromised.

*and similarly when using Resumption*
in particular both sets of negotiated parameters are authenticated and correlated
Symbolic Implementation of Crypto

**Crypto.fsi (interface)**

```fsharp
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)**

```fsharp
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
```
Symbolic Verification Results

- All properties are automatically proved
  - But after a lot of hand-tuning on the source code (otherwise ProVerif runs out of memory or does not finish)
  - Final ProVerif script of *Handshake+Resumption+Record* still large (2100LOC)
  - Proving Record/Handshake separately is much easier (but less precise)

- Experimental details:

<table>
<thead>
<tr>
<th>Part of protocol verified</th>
<th># of queries</th>
<th>PV running time</th>
<th>Memory used</th>
</tr>
</thead>
<tbody>
<tr>
<td>Handshake (auth. queries)</td>
<td>2</td>
<td>16sec</td>
<td>60MB</td>
</tr>
<tr>
<td>Handshake (secr. queries)</td>
<td>2</td>
<td>10sec</td>
<td>80MB</td>
</tr>
<tr>
<td>Handshake + Resumption (resumption auth. queries)</td>
<td>2</td>
<td>4min</td>
<td>460MB</td>
</tr>
<tr>
<td>Handshake + Resumption + Record (record auth. queries)</td>
<td>2</td>
<td>6min</td>
<td>700MB</td>
</tr>
<tr>
<td>Handshake + Resumption + Record</td>
<td>8</td>
<td>2hours</td>
<td>1.7GB</td>
</tr>
</tbody>
</table>
Symbolically Identified Pitfalls

- An early version of our implementation failed to correlate certificates; this leads to a counter-example of Handshake authentication
  - Common mistake: Ruby/Cisco/OpenSSL

- There is a potential SSL2 version rollback attack during resumption
  - The version rollback protection in ClientKeyExchange is absent during Resumption
  - Experimentally, we could not downgrade to SSL 2.0 with e.g. IIS, Apache.

```
Client      Server
ClientHello ServerHello
ClientKeyExchange Certificate
ClientKeyExchange Certificate
[ChangeCipherSpec] Finished
[ChangeCipherSpec] Finished
```

\[ ClientHello : (\text{ver\_max, cipher\_suites, \ldots}) \]

\[ ClientKeyExchange : \{\text{pms}\}_\text{pk(server)} \]

with \( \text{pms} = \text{ver\_max} || \text{random} \)
Computational Verification

• Computational models are more realistic than symbolic models (bitstrings instead of terms, PPT adversaries), but harder to work with

• Use Blanchet’s recent CryptoVerif \[\text{Blanchet-S&P’06}\] as a backend
  – CV script = PPT processes + crypto assumptions + security goals
  – Automatic computational proofs using the game-hopping technique

• Develop a new tool compiling F# code to CryptoVerif scripts
  – Networking and sampling functions translate to CryptoVerif primitives
  – Public functions translate to polynomially replicated processes

• Manually code crypto assumptions in CryptoVerif syntax
  – 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

• Run CryptoVerif on generated script + crypto assumptions + security goals to computationally verify these goals against PPT adversaries
Security Properties (Record)

• **Verify Record** in isolation
  – Assume a pre-established connection
  – Any (polynomial) number of clients and servers share the connection

• **Crypto assumptions:**
  – **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 $p$.

**Payload Secrecy**
In any polynomial run of the protocol, the sequence of sent payload values is indistinguishable from a sequence of independent random values.
## Computational Crypto Model

### Crypto.fsi (interface)

- `type bytes (* byte arrays *)`
- `type symkey (* symmetric keys *)`
- `type keyseed = bytes (* seed for symkey *)`

...  

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

...  

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

...  

### Crypto.cv (CryptoVerif script)

- `(* type bytes = blocksize *)`
- `type symkey [fixed].`
- `type keyseed [large, fixed].`

- `fun aes_encrypt(symkey, blocksize): blocksize.`
- `fun mkKey(keyseed): key.`
- `fun aes_decrypt(key, blocksize): blocksize.`

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

- `(* SPRP assumption for AES Encryption *)`
- `equiv`  
  `!N new r: keyseed;`  
  `((x:blocksize) Nsymenc -> aes_encrypt(mkKey(r), x),`  
  `(m:blocksize) Nsymdec -> aes_decrypt(mkKey(r), m))`  
  `<= (N * Psymenc(time, Nsymenc, Nsymdec)) => ...`
SPRP assumption

equiv

\!N \textbf{new} r: \text{keyseed};
\textstyle ((x:\text{blocksize}) \text{N}_1 \rightarrow \text{symenc}(x, \text{kgen}(r)),
\textstyle (m:\text{blocksize}) \text{N}_2 \rightarrow \text{symdec}(m, \text{kgen}(r)))

\leq \textstyle (N \cdot \text{Psymenc}(\text{time}, \text{N}_1, \text{N}_2)) \rightarrow

\!N \textbf{new} r: \text{keyseed};
\textstyle ((x:\text{blocksize}) \text{N}_1 \rightarrow
\textstyle \quad \textbf{find} \; j \leq \text{N}_1 \; \textbf{suchthat} \; \text{defined}(x[j], r_2[j]) \; \&\& \; \text{otheruses}(r_2[j]) \; \&\& \; x = x[j] \; \textbf{then} \; r_2[j]
\textstyle \quad \textbf{orfind} \; k \leq \text{N}_2 \; \textbf{suchthat} \; \text{defined}(r_4[k], m[k]) \; \&\& \; \text{otheruses}(r_4[k]) \; \&\& \; x = r_4[k] \; \textbf{then} \; m[k]
\textstyle \quad \textbf{else new} \; r_2: \text{blocksize}; \; r_2,
\textstyle (m:\text{blocksize}) \text{N}_2 \rightarrow
\textstyle \quad \textbf{find} \; j \leq \text{N}_1 \; \textbf{suchthat} \; \text{defined}(x[j], r_2[j]) \; \&\& \; \text{otheruses}(r_2[j]) \; \&\& \; m = r_2[j] \; \textbf{then} \; x[j]
\textstyle \quad \textbf{orfind} \; k \leq \text{N}_2 \; \textbf{suchthat} \; \text{defined}(r_4[k], m[k]) \; \&\& \; \text{otheruses}(r_4[k]) \; \&\& \; m = m[k] \; \textbf{then} \; r_4[k]
\textstyle \quad \textbf{else new} \; r_4: \text{blocksize}; \; r_4).
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:
  - \textit{IND-CCA2} for asymmetric encryption: indistinguishability against chosen-ciphertext attacks
  - \textit{random oracle} for PRF (key derivation): turns derived keys into random bitstrings

\textbf{Secrecy of PMS Random} (recall that $\text{pms} = \text{ver\_max} || \text{random}$)

In any polynomial run of the protocol, the sequence of random values is \textit{indistinguishable} from a sequence of independent fresh values.
Security Properties (Handshake auth.)

• A similar setting as for Handshake secrecy:
  – we assume pre-established parameters and a public/private keypair
  – parties generate connection keys, and send and receive Finished messages
  – the messages are sent over the Record layer in NULL mode (enc=mac=identity)

• Crypto assumptions:
  – random oracle for PRF when used for key derivation
  – UF-CMA for PRF when used to build the Finished messages

Agreement on PMS

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 pms.
Computational vs Symbolic Results

• Some properties hold symbolically but not computationally:
  – Computationally, mac functions give no secrecy guarantees
  – For the Handshake protocol, symbolically, \textit{pms} is (syntactically) secret,
    but computationally, only \textit{random} is secret
    (where \textit{pms} = \textit{ver_max} || \textit{random})
  – Encryption keys are secret computationally only before they are used
  – \texttt{let Concat(x,y) = m} prevents a type-flaw symbolically, but not computationally

• Symbolically, we were able to analyze the full protocol, but not (yet) computationally
  – \textit{mac-then-encrypt} is difficult in CryptoVerif
  – CryptoVerif interactive mode hard to use with generated code
Previous Analyses of TLS

- 1996 - Schneier & Wagner “Analysis of the SSL3.0 protocol”, informal, full protocol
- 1999 - Paulson “Inductive Analysis of the Internet protocol TLS”, theorem-proving, handshake protocol
- 2001 - Krawczyk “The Order of Encryption and Authentication for Protecting Communications (or: How Secure Is SSL?)”, computational analysis, record protocol
- 2001 - Yasinac, Childs "Analyzing Internet Security Protocols", automatic symbolic analysis, handshake
- 2005 - Ogata, Futatsugi "Equational Approach to Formal Analysis of TLS“, symbolic analysis, handshake
- 2005 - He, Sundararajan, Datta, Derek, Mitchell, "A modular correctness proof of IEEE802.11i and TLS" manual symbolic analysis, handshake protocol
- 2008 - Kamil, Lowe “Analysing TLS in the Strand Spaces Model”, manual symbolic analysis, full handshake and record protocols
- 2008 - Chaki, Datta “Automated verification of security protocol implementation”, automatic (with Copper) symbolic analysis of OpenSSL code
- ..... (padding attacks, side-channel attacks) .....
Comparing Verification Methods
## 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>4</td>
<td>6.6s</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>5</td>
<td>2.6s</td>
</tr>
<tr>
<td>Password-X.509 Authenticated XML RPC</td>
<td>session</td>
<td>token</td>
<td>15</td>
<td>44m</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></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>

Verifying miTLS

<table>
<thead>
<tr>
<th>Component</th>
<th>F# (LOC)</th>
<th>F7 (LOC)</th>
<th>F7 (S)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Base</td>
<td>945</td>
<td>581</td>
<td>11</td>
</tr>
<tr>
<td>TLS Record</td>
<td>826</td>
<td>511</td>
<td>77</td>
</tr>
<tr>
<td>Handshake/CCS</td>
<td>2400</td>
<td>777</td>
<td>413</td>
</tr>
<tr>
<td>Alert Protocol</td>
<td>184</td>
<td>119</td>
<td>105</td>
</tr>
<tr>
<td>AppData Protocol</td>
<td>139</td>
<td>113</td>
<td>34</td>
</tr>
<tr>
<td>TLS API</td>
<td>640</td>
<td>426</td>
<td>309</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>5134</strong></td>
<td><strong>2527</strong></td>
<td><strong>949</strong></td>
</tr>
</tbody>
</table>
## TLS Verification

<table>
<thead>
<tr>
<th>Verified Parts of TLS</th>
<th>Security Goals</th>
<th>F# Code</th>
<th>Queries</th>
<th>Time</th>
<th>Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>Full Handshake</td>
<td>Authentication</td>
<td>1418</td>
<td>2</td>
<td>27 s</td>
<td>60 MB</td>
</tr>
<tr>
<td>Full Handshake</td>
<td>Secrecy</td>
<td>1418</td>
<td>2</td>
<td>25 s</td>
<td>80 MB</td>
</tr>
<tr>
<td>Full Handshake &amp;</td>
<td>Authentication</td>
<td>2194</td>
<td>2</td>
<td>8 min</td>
<td>460 MB</td>
</tr>
<tr>
<td>Resumption</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Full Handshake &amp;</td>
<td>Authentication</td>
<td>3344</td>
<td>2</td>
<td>11 min</td>
<td>700 MB</td>
</tr>
<tr>
<td>Resumption &amp; Record</td>
<td>(Record Only)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Full Handshake &amp;</td>
<td>Authentication &amp;</td>
<td>3344</td>
<td>10</td>
<td>3.5 h</td>
<td>4.5 GB</td>
</tr>
<tr>
<td>Resumption &amp; Record</td>
<td>Secrecy</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Full TLS &amp; Password-based Application</td>
<td>User Authentication</td>
<td>3855</td>
<td>1</td>
<td>1.5 h</td>
<td>1.2 GB</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Verification Result</th>
<th>F# Code</th>
<th>Crypto Assumptions</th>
<th>Games</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Record Authentication (Theorem 7)</td>
<td>1967 lines</td>
<td>18 lines</td>
<td>15</td>
<td>1.9 s</td>
</tr>
<tr>
<td>Record Secrecy (Theorem 8)</td>
<td>1967 lines</td>
<td>25 lines</td>
<td>14</td>
<td>0.3 s</td>
</tr>
<tr>
<td>PMS Random Secrecy (Theorem 9)</td>
<td>2497 lines</td>
<td>33 lines</td>
<td>18</td>
<td>1.1 s</td>
</tr>
<tr>
<td>MS Authentication (Theorem 10)</td>
<td>2497 lines</td>
<td>23 lines</td>
<td>8</td>
<td>24 s</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Component</th>
<th>F# (LOC)</th>
<th>F7 (LOC)</th>
<th>F7 (S)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Base</td>
<td>945</td>
<td>581</td>
<td>11</td>
</tr>
<tr>
<td>TLS Record</td>
<td>826</td>
<td>511</td>
<td>77</td>
</tr>
<tr>
<td>Handshake/CCS</td>
<td>2,400</td>
<td>777</td>
<td>413</td>
</tr>
<tr>
<td>Alert Protocol</td>
<td>184</td>
<td>119</td>
<td>105</td>
</tr>
<tr>
<td>AppData Protocol</td>
<td>139</td>
<td>113</td>
<td>34</td>
</tr>
<tr>
<td>TLS API</td>
<td>640</td>
<td>426</td>
<td>309</td>
</tr>
<tr>
<td>Total</td>
<td>5,134</td>
<td>2,527</td>
<td>949</td>
</tr>
</tbody>
</table>
Model Extraction vs. Program Analysis

- ProVerif automatically verifies medium-size protocols
  - Can prove authentication queries and syntactic secrecy
  - New work on strong secrecy, equivalence-based reasoning
    See: [http://proverif.inria.fr](http://proverif.inria.fr)
  - Other new tools can better handle stateful protocols, Diffie-Hellman key exchanges
    See e.g.: Tamarin, StatVerif

- RCF/F7/F* can verify larger programs but require type annotations
  - Can prove authentication and computational indistinguishability
  - New improvements in type inference, stateful programs
    See: [http://research.microsoft.com/fstar](http://research.microsoft.com/fstar)
  - Other similar tools can verify Java, C programs
    See: CVJ, Csec
Symbolic vs. Computational Models

• Computational proofs require manual intervention
  – Tools can be proof assistants
  – Results relate well to cryptographic literature
  – Many new tools: EasyCrypt, CertiCrypt, RF*

• Symbolic proofs are automatic
  – Prove weaker properties
  – Against weaker adversaries
  – Useful for finding attacks
  – Results relate to programming language semantics
  – Stable, well-documented tools: ProVerif, Tamarin, F7, F*
Examples
Authenticated Encryption

• Define cryptographic interfaces for enc and MAC
  – Both symbolic and computational

• Use them to implement authenticated encryption
  – Encrypt-Then-MAC (IPsec-style)
  – MAC-And-Encrypt (SSH style)

• Prove that your code satisfies the security goals of authenticated encryption
  – Use typing with RCF, both symbolic and computational
module MAC

type text = bytes

type mac = bytes

type key

val GEN : unit -> key

val LEAK : key -> bytes

val COERCE: bytes-> key

val MAC : key -> text -> mac

val VERIFY: key -> text -> mac -> bool
Basic Interfaces: ENC

module ENC

type plain

type cipher = bytes

type key

val GEN : unit -> key
val LEAK : key -> bytes
val COERCE: bytes-> key

val ENC : key -> plain -> cipher
val DEC : key -> cipher -> plain
Authenticated Encryption

• Provides both confidentiality and integrity
  – Encryption turns secret data into public data
  – Data returned by decryption is guaranteed to be the same as the data that was encrypted

• Consider two implementations:
  – Enc-Then-Mac (IPsec).
    Encryption returns: \( \text{MAC } k \ (\text{ENC } k' \ t) \)
  – Mac-And-Enc (SSH)
    Encryption returns: \( (\text{MAC } k \ t, \text{ENC } k' \ t) \)
Tasks (to submit)

1. Define RCF interfaces for ENC & MAC
2. Define symbolic code for ENC & MAC
3. Prove that the implementations meet the interface (by typing in RCF)
4. Define an RCF interface for AuthEnc
   - Similar to ENC, but with stronger guarantees
5. Define concrete code for AuthEnc
6. Prove that this code meets the interface
   - Can you weaken the crypto assumptions?
   - Does your proof give computational security?