DigiCosme 2013

Modular type-based cryptographic verification

Cédric Fournet et al.
fournet@microsoft.com
http://research.microsoft.com/~fournet
http://msr-inria.inria.fr/projects/sec
http://msr-inria.inria.fr/projects/sec/digicosme
Today: Authenticity by Typing

1. Verifying Protocols
2. Specifications, models, and implementations
3. RCF: syntax, semantics, and safety
4. Programming example: access control in F7 (demo)
5. RCF: refinement types
6. Programming example: a client-server protocol
7. Symbolic verification for a sample protocol in F7 (demo)
8. Computational verification for the same protocol (demo)

Tomorrow: Secrecy; Application: TLS
Context:
Verifying Protocols
Cryptographic protocols go wrong

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

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

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

They threw down a gauntlet.

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

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

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

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

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

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

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

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

Needham and Schroeder CACM (1978)

1978: N&S propose authentication protocols for “large networks of computers”
1981: Denning and Sacco find attack on N&S symmetric key protocol
1983: Doelev and Yao first formalize secrecy properties of NS threat model using formal algebra
1987: Burrows, Abadi, Needham invent authentication logic; incomplete, but useful
1994: Hickman, Elgamal invent SSL; holes in v1, v2, but v3 fixes these, very widely deployed
1994: Ylonen invents SSH; holes in v1, but v2 good, very widely deployed
1995: Lowe finds insider attack on N&S asymmetric protocol; rejuvenates interest in FMs
circa 2000: Several FMs for “D&Y problem”: tradeoff between accuracy and approximation
circa 2007: Many FMs developed; several deliver both accuracy and automation
• Authentication and secrecy properties for basic crypto protocols have been formalized and thoroughly studied.
• After intense effort on symbolic reasoning, several techniques and tools are available for automatically proving these properties – e.g. Athena, TAPS, ProVerif, CryptoVerif, FDR, AVISPA, etc.
• We can automatically verify most security properties for detailed models of crypto protocols – e.g. IPSEC, Kerberos, Web Services, Infocard, TLS.
2013: done?

- Best practice: apply formal methods and tools throughout the protocol design & review process
- Not so easy
  - Specifying a protocol is a lot of work
  - Practitioners don’t understand formal models
- Protocols go wrong because...
  - they are logically flawed, or
  - they wrongly use cryptography, or
  - they are wrongly implemented, or
  - they are used for the wrong purpose
- Some troublesome questions
  1. How to relate formal models to protocol implementations?
  2. How to relate crypto protocols to application security?
  3. How to relate symbolic models to computational cryptography?
Models: Formal vs Computational Cryptography

• Two approaches for verifying protocols and programs
  
  **Symbolic models** (Needham-Schroeder, Dolev-Yao, ... late 70’s)
  – Structural view of protocols, using formal languages and methods
  – Many automated verification tools, scales to large systems
    including full-fledged implementations of protocol standards
  
  **Computational models** (Yao, Goldwasser, Micali, Rivest, ... early 80’s)
  – Concrete, algorithmic view, using probabilistic polynomial-time machines
  – New formal tools: CryptoVerif, Certicrypt, Easycrypt

• Can we get the best of both worlds? Much ongoing work on computational soundness for symbolic cryptography
  (Abadi Rogaway, Backes Pfitzman Wagner, Warinschi,... mid 00’s)
  – It works... with many mismatches, restrictions, and technicalities
  – At best, one still needs to verify protocols symbolically

• Can we directly verify real-world protocols?
  These lectures: type-based verification, first symbolically, then computationally
Protocol
Implementations
Goal: Verify production code relying on Cryptography

- Communications Protocol (IPSEC, TLS)
- Cryptographic libraries (XML security, WS* standards, TCG)
- Security Components (InfoCard, DKM, TPM)
Specs, code, and formal tools

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

**Computational Analyses**
- CryptoVerif ('06)
- EasyCrypt ('11)
- F7 ('11)
- RF* ('13)

**Hand Proofs**
- NRL
- Athena
- Securify
- Scyther
- Cryptyc
- AVISPA
- ProVerif ('01)
- F7 ('08)
- F* ('11)

**Symbolic Analyses**
- Casper
- SMT Solvers
- Theorem Provers
- Model Checkers

**C/C++**
- ML, F#
- Ruby
- Java
- C#

**General Verification**
- Protocol Implementations and Applications
Models vs implementations

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

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

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

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

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

• Executable code has better tool support
  – Types, compilers, debuggers, libraries, verification tools
One source, three tasks

Source code with modules and strong interfaces

My code

Authz

Other Libraries

My protocol

Application

Symbolic Crypto

Concrete Crypto

Platform

Crypto Net

Some other implementation

ProVerif

fs2pv

Symbolic verification

Symbolic testing & debugging

Interoperability
1. Symbolic testing and debugging

- My code
- Authz
- Other Libraries

Coded in C#, F#...

We use idealized cryptographic primitives
Safety relies on typing

We model attackers as arbitrary code with access to selected libraries

Symbolic Crypto

Attacker (test)

Platform
Crypto Net
2. Formal verification

We only support a subset of F#

My code

Authz

Other Libraries

Application

We model attackers as arbitrary code with access to selected libraries

We translate to the pi calculus

Translated to

fs2pv

ProVerif

Pass: Ok for all attackers, or
No + potential attack trace

Formal verification considers ALL such attackers

My protocol

Symbolic

Crypto

Attacker

(unknown)
3. Concrete testing & interop

My code
My protocol
Application
Authz
Other Libraries
Concrete Crypto
Coded in C#, F#...

My protocol

We test that our code produces and consumes the same messages as another implementation

We only change our implementation of cryptographic primitives

Concrete Crypto

Attacker (test)
We can still run attacks to test other implementations

Platform
Crypto Net

Some Other Implementation

Interoperability
A large example: TLS in F# & F7

We develop a **reference implementation** for SSL 3.0—TLS 1.2 & extensions

1. **Standard compliance**: we closely follow the RFCs
   - concrete message formats
   - support for multiple ciphersuites, sessions and connections, re-handshakes and resumptions, alerts, message fragmentation,…
   - interop with other implementations such as web browsers and servers

2. **Verified security**: we structure our code to enable its automated modular verification, from its main API down to standard assumptions on its base cryptography (e.g. RSA)
   - formal computational security theorems for a 5000-line functionality (automation required)

3. **Experimental platform**: for testing corner cases, trying out attacks, analysing new extensions and patches, …
RCF: a concurrent $\lambda$-calculus with refinement types

Slides adapted from A. D. Gordon’s see also the RCF tutorial
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 \textit{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 \textit{seals} (Morris 1973, Sumii and Pierce 2002), not primitive as in the applied \textit{pi calculus}
  – Security specs via \textit{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 \textit{refinement types} (Pfenning 1992, ...) aka \textit{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):

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

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

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

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

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

$\text{let } x = M \in A \rightarrow A\{M/x\}$

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

\[\text{let } f = \text{fun } x \rightarrow x + 1 \in (f \ 7)\]
\[\rightarrow (\text{fun } x \rightarrow x + 1) \ 7\]
\[\rightarrow 7 + 1\]
\[\rightarrow 8\]
Example: Booleans and Conditional Branching:

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

\[ A, B ::= \]

\[
\quad \ldots
\]

\[
(\forall a)A \quad \text{as before}
\]

\[
a!M \quad \text{local channel}
\]

\[
a? \quad \text{transmission of } M \text{ on channel } a
\]

\[
a? \quad \text{receive message off channel}
\]

\[
A \uparrow B \quad \text{parallel composition}
\]

\[
a!M \uparrow a? \rightarrow M \quad \text{Reductions step are “up to structural rearrangements”}
\]

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

Heating 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]
Parallel Composition:

\[ A, B ::= \]

\[ \ldots \]

\[ A \uparrow B \quad \text{expression as before} \]

\[ (\_ \uparrow A \equiv A) \]

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

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

\textbf{let} \( x = (A \uparrow A') \textbf{in} B \equiv A \uparrow (\textbf{let} \ x = A' \textbf{in} B) \]

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

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

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

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

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

\textbf{let} \( x = (1 \uparrow (2 \uparrow 3)) \textbf{in} Fx \)
Channel-Based Communications:

\[ A, B ::= \]

\[ \ldots \]

\[ a!M \]  

\[ a? \]  

expression as before transmission of \( M \) on channel \( a \)
receive message off channel

\[ a!M \Rightarrow a!M \uparrow () \]
\[ a!M \uparrow a? \rightarrow M \]

\[ a!0 \uparrow a!1 \uparrow (\textbf{let } x = a? \textbf{ in } (a!(x+2) \uparrow x)) \]

\[ \equiv a!1 \uparrow \textbf{let } x = (a!0 \uparrow a?) \textbf{ in } (a!(x+2) \uparrow x) \]

\[ \rightarrow a!1 \uparrow \textbf{let } x = 0 \textbf{ in } (a!(x+2) \uparrow x) \]

\[ \rightarrow a!1 \uparrow a!2 \uparrow 0 \]

\[ a!0 \uparrow a!1 \uparrow (\textbf{let } x = a? \textbf{ in } (a!(x+2) \uparrow x)) \]

\[ \equiv \rightarrow\rightarrow a!0 \uparrow a!3 \uparrow 1 \]
Name Generation:

\[ A, B ::= \quad \text{expression as before} \]

\[ \quad \text{fork} \]

\[ (\nu a) A \]

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

\[ a \notin fn(A') \Rightarrow A' \vdash ((\nu a) A) \Rightarrow (\nu a)(A' \vdash A) \]

\[ a \notin fn(A') \Rightarrow ((\nu a) A) \vdash A' \Rightarrow (\nu a)(A \vdash A') \]

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

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

\[(T)\text{chan } \triangleq (T \rightarrow \text{unit}) \ast (\text{unit} \rightarrow T)\]

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

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

send \(x\) on \(c\)

\[\text{recv } \triangleq \text{fun } c \rightarrow \text{let } (s, r) = c \ \text{in } r ()\]

block for \(x\) on \(c\)

\[\text{fork } \triangleq \text{fun } f \rightarrow (f () \triangleright () )\]

run \(f\) in parallel

Example: Mutable State:

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

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

new reference to \(M\)

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

derereference \(M\)

\[M \ := N \triangleq \text{let } x = \text{recv } M \ \text{in } \text{send } M \ N\]

update \(M\) with \(N\)
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:

\[ A, B ::= \]

... as before

assume \( C \) assumption of formula \( C \)

assert \( C \) assertion of formula \( C \)

assume \( C \Rightarrow \) assume \( C \vdash () \)

assert \( C \rightarrow () \)
Semantics: expression safety

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

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

  active assumptions  pending messages  running threads

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

assert \((p \land q \Rightarrow q)\)

assert \((p \lor q \Rightarrow q)\)

assume \((p \Rightarrow q)\); assert \((p \lor q \Rightarrow q)\)

let \(x = 0\) in assert \((x = 1)\)

\(a!0 \vdash a!1 \vdash (\text{let } x = a? \text{ in assert } (x=0 \lor x=1))\)

\(a!0 \vdash a!1 \vdash (\text{let } x = a? \text{ in assert } x=1)\)

\(a!0 \vdash a!1 \vdash (\text{let } x = a? \text{ in if } x > 0 \text{ then assert } x=1)\)

...
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

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

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

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

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

```haskell
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
RCF Part 2:

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

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

\[ H, T, U, V ::= \text{type} \]
\[ \quad \text{unit} \quad \text{unit type} \]
\[ \quad \Pi x : T. U \quad \text{dependent function type (scope of } x \text{ is } U) \]
\[ \quad \Sigma x : T. U \quad \text{dependent pair type (scope of } x \text{ is } U) \]
\[ \quad T + U \quad \text{disjoint sum type} \]
\[ \mu \alpha.T \quad \text{iso-recursive type (scope of } \alpha \text{ is } T) \]
\[ \alpha \quad \text{iso-recursive type variable} \]
\[ \{ x : T \mid C \} \quad \text{refinement type (scope of } x \text{ is } C) \]

\[ \{C\} \triangleq \{ _{-} : \text{unit} \mid C \} \quad \text{ok-type} \]
\[ \text{bool} \triangleq \text{unit} + \text{unit} \quad \text{Boolean type} \]
Starting Point: The Type System for FPC:

\[
\begin{array}{c}
E \vdash \Box \quad (x : T) \in E \\
E \vdash A : T \\
E, x : T \vdash B : U \\
E \vdash \text{let } x = A \text{ in } B : U \\
E \vdash \Box \\
E \vdash M : T \\
E \vdash N : U \\
E \vdash () : \text{unit} \\
E \vdash M = N : \text{unit + unit} \\
\end{array}
\]

\[
\begin{array}{c}
E, x : T \vdash A : U \\
E \vdash M : (T \rightarrow U) \\
E \vdash N : T \\
E \vdash \text{fun } x \rightarrow A : (T \rightarrow U) \\
E \vdash M \; N : U \\
\end{array}
\]

\[
\begin{array}{c}
E \vdash M : T \\
E \vdash N : U \\
E \vdash (M,N) : (T \times U) \\
E \vdash M : (T \times U) \\
E, x : T, y : U \vdash A : V \\
E \vdash \text{let } (x,y) = M \text{ in } A : V \\
\end{array}
\]

\[
\begin{array}{c}
h : (T,U) \\
E \vdash M : T \\
E \vdash U \\
E \vdash M : T \\
h : (H,T) \\
E, x : H \vdash A : U \\
E \vdash B : U \\
E \vdash \text{match } M \text{ with } h\ x \rightarrow A \text{ else } B : U \\
\end{array}
\]

\[
\begin{array}{c}
\text{inl} : (T,T+U) \\
\text{inr} : (U,T+U) \\
\text{fold} : (T\{\mu \alpha . T / \alpha\}, \mu \alpha . T) \\
\end{array}
\]
Starting Point: The Type System for FPC:

\[
\begin{align*}
E \vdash \Diamond & \quad (x : T) \in E & E \vdash A : T & E, x : T \vdash B : U \\
E \vdash x : T & & E \vdash \text{let } x = A \text{ in } B : U
\end{align*}
\]

\[
\begin{align*}
E \vdash \Diamond & & E \vdash M : T & E \vdash N : U \\
E \vdash () : \text{unit} & & E \vdash M = N : \text{unit + unit}
\end{align*}
\]

\[
\begin{align*}
E, x : T \vdash A : U & & E \vdash M : (T \to U) & E \vdash N : T \\
E \vdash \text{fun } x \to A : (T \to U) & & E \vdash M N : U
\end{align*}
\]

\[
\begin{align*}
E \vdash M : T & E \vdash N : U & E \vdash M : (T \times U) & E, x : T, y : U \vdash A : V \\
E \vdash (M, N) : (T \times U) & & E \vdash \text{let } (x, y) = M \text{ in } A : V
\end{align*}
\]

\[
\begin{align*}
h : (T, U) & E \vdash M : T & E \vdash U \\
E \vdash h M : U
\end{align*}
\]

\[
\begin{align*}
E \vdash M : T & h : (H, T) & E, x : H \vdash A : U & E \vdash B : U \\
E \vdash \text{match } M \text{ with } h x \to A \text{ else } B : U
\end{align*}
\]

\[
\begin{align*}
inl : (T, T + U) & \quad \text{inr : } (U, T + U) & \text{fold : } (T \{\mu\alpha. T / \alpha\}, \mu\alpha. T)
\end{align*}
\]
Rules for Formula Derivation:

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

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

\[
E \vdash C
\]

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

\[
\begin{align*}
E \vdash \diamond \quad & fnfv(C) \subseteq dom(E) \\
E \vdash \textbf{assume} \; C : \{ \_ : \text{unit} \mid C \} \\
E \vdash C \\
E \vdash \textbf{assert} \; C : \text{unit}
\end{align*}
\]
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:

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

\[
\begin{align*}
E \vdash A : T & \quad E \vdash T <: T' \\
\hspace{1cm} & \quad E \vdash A : T' \\
E \vdash \Box & \quad E \vdash T' <: T & \quad E, x : T' \vdash U <: U' \\
E \vdash \text{unit} <: \text{unit} & \quad E \vdash (\Pi x : T. U) <: (\Pi x : T'. U') \\
E \vdash T <: T' & \quad E, x : T \vdash U <: U' & \quad E \vdash T <: T' & \quad E \vdash U <: U' \\
E \vdash (\Sigma x : T. U) <: (\Sigma x : T'. U') & \quad E \vdash (T + U) <: (T' + U') \\
E \vdash \Box & \quad (\alpha <: \alpha') \in E & \quad E, \alpha <: \alpha' \vdash T <: T' & \quad \alpha \notin \text{fnfv}(T') & \quad \alpha' \notin \text{fnfv}(T) \\
\hspace{1cm} & \quad E \vdash \alpha <: \alpha' & \quad E \vdash (\mu \alpha. T) <: (\mu \alpha'. T')
\end{align*}
\]
Rules for Restriction, I/O, and Parallel Composition:

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

\[
E, \_ : \{\overline{A_2}\} \vdash A_1 : T_1 \quad E, \_ : \{\overline{A_1}\} \vdash A_2 : T_2
\]

\[
E \vdash (A_1 \uparrow A_2) : T_2
\]

\[
(\forall a)A = (\exists a.\overline{A}) \\
\text{let } x = A_1 \text{ in } A_2 = \overline{A_1}
\]

\[
\overline{A_1 \uparrow A_2} = (\overline{A_1} \land \overline{A_2})
\]

\[
\text{assume } C = C
\]

\[
\overline{A} = \text{True} \quad \text{if } A \text{ matches no other rule}
\]

**Exercise:** Find types to typecheck the following code:

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

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

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

**Lemma** If \( \emptyset \vdash S : T \) then \( S \) is statically safe.

**Lemma** If \( E \vdash A : T \) and \( A \Rightarrow A' \) then \( E \vdash A' : T \).

**Lemma** If \( E \vdash A : T \) and \( A \rightarrow A' \) then \( E \vdash A' : T \).

**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.)
A CLIENT-SERVER PROTOCOL (USING PRIVATE CHANNELS)
Summary on RCF

• RCF supports
  – functional programming à 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/Fstar

• Related language design and implementation: Aura, Fable, F7, Fine, F*...
AUTHENTICATED RPC

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

Security depends on the following:

1. The function $hmacsha1$ 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

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

\[
\forall a,b,s,t. \text{RecvResponse}(a,b,s,t) \iff \\
(\text{Request}(a,b,s) \land \text{Response}(a,b,s,t)) \lor \text{Bad}(a) \lor \text{Bad}(b)
\]
Sample ideal functionality:

Keyed cryptographic hashes

```fsharp
module MAC

type text = bytes
val macsize

type key = bytes

val mac = bytes

val GEN : unit -> key

val MAC : key -> text -> mac

val VERIFY: key -> text -> mac -> bool
```

This interface says nothing on the security of MACs.
MAC keys are abstract

module MAC

type text = bytes val macsize

type key

type mac = bytes

val GEN : unit -> key

val MAC : key -> text -> mac

val VERIFY : key -> text -> mac -> bool
Sample ideal functionality:

Keyed cryptographic hashes

MAC keys are abstract

```ocaml
module MAC

type text = bytes  val macsize

type key

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

val GEN : unit -> key
val MAC : key -> text -> mac
val VERIFY: key -> text -> mac -> bool
```
Sample ideal functionality:

**Keyed cryptographic hashes**

```ocaml
module MAC

type text = bytes    val macsize

val type key

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

val predicate Msg of key * text

val GEN : unit -> key

val MAC : k:key -> t:text{Msg(k,t)} -> mac

val VERIFY : k:key -> t:text -> mac
            -> b:bool{ b=true ⇒ Msg(k,t)}
```

**ideal F7 interface**

MAC keys are abstract.

MACs are fixed sized.

Msg is specified by protocols using MACs.

“All verified messages have been MACed.”

---

**module RPC**

**definition** !k,q. Msg(k,Utf8(q)) <-> Request(q)

let client q =
  // precondition:
  // Request(q)
  ... send MAC k (utf8 q)

let server q =
  ... if VERIFY k (utf8 q) m
  then // we have Request(q)
  process q

**sample protocol using MACs**
SYMBOLIC VERIFICATION:
LOGICAL INVARIANTS FOR CRYPTOGRAPHY
Our latest symbolic approach [POPL’10]

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:

```fsharp
let setup (a: str) (b: str) =
    let k = mkKeyAB a b in
    (fun s → client a b k s),
    (fun _ → server a b k),
    (fun _ → assume (Bad(a)); k),
    (fun _ → assume (Bad(b)); k)
```

Opponent Interface (excerpts):

```fsharp
val send: conn → bytespub → unit
val recv: conn → bytespub
val hmacsha1 : keypub → bytespub → bytespub
val hmacsha1Verify : keypub → bytespub → bytespub → unit
val setup: strpub → strpub →
    (strpub → unit) * (unit → unit) * (unit → keypub) * (unit → keypub)
```
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 $I_L, I_R \vdash O : \text{unit}$, then $X[O]$ is semantically safe.
Security 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:

```
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 hmacsha1 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)
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)
  \[ \forall v,v',s,s',t,t'.
  \quad \text{(Responded}(v,s,t) \land \text{Responded}(v',s',t') \land v = v') \Rightarrow (s = s' \land t = t') \]

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:

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

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

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

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

(KeyAB Injective) is a theorem: each key is used by a single pair of principals.

(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:

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

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

(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
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):**

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

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

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

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

\[ a \rightarrow b: \quad \text{aes} \ k_{ab}^e t \mid \text{hmacsha1} \ k_{ab}^m t \]
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:**

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

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.
CASE STUDY

CARDSPACE & WEB SERVICES SECURITY
InfoCard: Information Card Profile

1. Request
2. Here is RP’s Policy (go to IP)
3. Get IP Policy
4. Get Issued Token (T) with card data
5. Submit (T)
6. Response

Selects card and provides password
Initially, C has: cardld, PK(kIP), PK(kRP); IP has: kIP, PK(kRP), Card(cardld, claimsU, pwdU, kIP, kcardld); RP has: kRP, PK(kIP)

C: Request (RP, Mreq)
   C receives an application request
U: Select InfoCard (cardld, C, RP, pwdU, typesRP)
   User selects card and provides password

C → IP: generate fresh k1, η1, η2, ηce
   Fresh session key, two nonces, and client entropy for token key
   Encrypt session key for IP
   Derive message signing key
   Derive message encryption key
   Token request message body
   User authentication token
   Message signature
   Token Request, with encrypted signatures, token and body
   
   let Mok = RSAEnc(PK(kIP), k1) in
   let ksig = PSHA1(k1, η1) in
   let kenc = PSHA1(k1, η2) in
   let Mres = RST(cardld, typesRP, RP, ηce) in
   let Muser = (U, pwdU) in
   let Mmac = HMACSHA1(ksig, (Mres, Muser)) in
   Request Token (Mok, η1, η2, AESSenc(kenc, Mmac), AESSenc(kenc, Muser), AESSenc(kenc, Mres))

IP: Issue Token (U, cardld, claimsU, RP, display)
   IP issues token for U to use at RP
   Fresh nonces, server entropy, token encryption key
   Derive message signing key
   Derive message encryption key
   Compute token key from entropies, encrypt for RP
   Compute PPID using card master key, RP’s identity
   SAML assertion with token key, claims, and PPID
   SAML assertion signed by IP
   Token encryption key, encrypted for RP
   Encrypted issued token
   Token response message body
   Message Signature
   Token Response, with encrypted signature and body

   let ksig = PSHA1(k1, η3) in
   let kenc = PSHA1(k1, η4) in
   let Mokkey = RSAEnc(PK(kIP), PSHA1(ηce, ηse)) in
   let ppidcardld, RP = H1(kcardld, RP) in
   let Mtok = Assertion(IP, Mokkey, claimsU, RP, ppidcardld, RP) in
   let Mtoksig = RSHASH1(k1, Mtok) in
   let Mok = RSAEnc(PK(kRP), k1) in
   let Menctok = (Mok, AESSenc(k2, SAML(Mtok, Mtoksig))) in
   let Mstr = RSTR(Menctok, ηse) in
   let Mmac = HMACSHA1(ksig, Mstr) in
   Token Response (η3, η4, AESSenc(kenc, Mmac), AESSenc(kenc, Mstr))

U: Approve Token (display)
   User approves token
   Fresh session key, three nonces
   Encrypt session key for RP
   Derive message signing key
   Derive message encryption key
   Compute token key from entropies
   Message signature
   Derive a signing key from the issued token key
   Endorsing signature proving possession of token key
   Service Request, with issued token, encrypted signatures and body

   let Mok = RSAEnc(PK(kIP), k2) in
   let ksig = PSHA1(k2, η5) in
   let kenc = PSHA1(k2, η6) in
   let kproof = PSHA1(ηce, ηse) in
   let Mmac = HMACSHA1(ksig, Mreq) in
   let kendorse = PSHA1(kproof, η7) in
   let Mproof = HMACSHA1(kendorse, Mmac) in
   Service Request (Mok, η5, η6, η7, Menctok, AESSenc(kenc, Mmac), AESSenc(kenc, Mproof), AESSenc(kenc, Mreq))

RP: Accept Request (IP, claimsU, Mreq, Mresp)
   RP accepts request and authorizes a response
   Fresh nonces
   Derive message signing key
   Derive message encryption key
   Message signature
   Service Response, with encrypted signatures and body
   C accepts response and sends it to application

   let ksig = PSHA1(k2, η8) in
   let kenc = PSHA1(k2, η9) in
   let Mmac = HMACSHA1(ksig, Mresp) in
   Service Response (η8, η9, AESSenc(kenc, Mmac), AESSenc(kenc, Mresp))

C: Response (Mresp)
InfoCard: Information Card Profile v1.0

Diagram showing the relationships between various technologies and protocols, including CardSpace, WS-Trust, Secure XML RPC, WS-Security, XML-Signature, XML-Encryption, WS-Addressing, Crypto Patterns, Principals, SOAP, Crypto, Db, Xml, Net, and Data.
Download, Compile, Verify, Execute

Identity Provider IP

Relying Party RP

policy

Get Policy

References

policy

Generate

policy2fs

Generate

RP.fs

C.fs

IP.fs

Symbolic Libraries

F7

Security goals

Client app A

Linked

proxy.dll

Concrete Libraries

Stop

Linked

Yes

C.fs

No

Stop

Concrete Libraries
## Performance relative to FS2PV/ProVerif

<table>
<thead>
<tr>
<th>Protocols and Libraries</th>
<th>F# Program Modules</th>
<th>LOCs</th>
<th>F7 Typechecking Interface</th>
<th>Time</th>
<th>FS2PV Verification Queries</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Trusted Libraries (Symbolic)</td>
<td>5</td>
<td>926 *</td>
<td>1167</td>
<td>29s</td>
<td>(Not Verified)</td>
<td></td>
</tr>
<tr>
<td>RPC Protocol</td>
<td>5+1</td>
<td>+ 91</td>
<td>+ 103</td>
<td>10s</td>
<td>4</td>
<td>6.65s</td>
</tr>
<tr>
<td>Principals</td>
<td>1</td>
<td>207</td>
<td>253</td>
<td>9s</td>
<td>(Not Verified)</td>
<td></td>
</tr>
<tr>
<td>Cryptographic Patterns</td>
<td>1</td>
<td>250</td>
<td>260</td>
<td>17.1s</td>
<td>(Not Verified)</td>
<td></td>
</tr>
<tr>
<td>Otway-Rees</td>
<td>2+1</td>
<td>+ 234</td>
<td>+ 255</td>
<td>1m 29.9s</td>
<td>10</td>
<td>8m 2.2s</td>
</tr>
<tr>
<td>Secure Conversations</td>
<td>2+1+1</td>
<td>+ 123</td>
<td>+ 111</td>
<td>29.64s</td>
<td>(Not Verified)</td>
<td></td>
</tr>
<tr>
<td>Web Services Security Library</td>
<td>7</td>
<td>1702</td>
<td>475</td>
<td>48.81s</td>
<td>(Not Verified)</td>
<td></td>
</tr>
<tr>
<td>X.509-based Client Auth</td>
<td>7+1</td>
<td>+ 88</td>
<td>+ 22</td>
<td>+ 10.8s</td>
<td>2</td>
<td>20.2s</td>
</tr>
<tr>
<td>Password-X.509 Mutual Auth</td>
<td>7+1</td>
<td>+ 129</td>
<td>+ 44</td>
<td>+ 12.0s</td>
<td>15</td>
<td>44m</td>
</tr>
<tr>
<td>X.509-based Mutual Auth</td>
<td>7+1</td>
<td>+ 111</td>
<td>+ 53</td>
<td>+ 10.9s</td>
<td>18</td>
<td>51m</td>
</tr>
<tr>
<td>Windows CardSpace</td>
<td>7+1+1</td>
<td>+ 1429</td>
<td>+ 309</td>
<td>+ 6m 3s</td>
<td>6</td>
<td>66m 21s*</td>
</tr>
</tbody>
</table>
Computational Soundness:
Typed Ideal Interfaces for Cryptographic Functionalities
Cryptographic primitives are partially specified

• Symbolic models reason about fully-specified crypto primitives
  – Same rewrite rules apply for the attacker as for the protocol
  – Each crypto primitive yields distinct symbolic terms

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

• Security proofs apply parametrically, for any concrete primitives that meet these assumptions

• Typed interfaces naturally capture partial specifications
Computational Security with F7

• We adapt our F7 typechecker
  – We remove non-determinism
  – We add probabilistic sampling and mutable references
  – We formally prove type safety & parametricity
    for this new extended subset of F7 using Coq/SSReflect

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

• Next: two sample ideal functionalities
  – For MACs (trace properties)
  – For encryption (equivalence properties)
Probabilistic RCF

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

\[ A \xrightarrow{p} A' \]

- We add a new “fair coin-tossing” primitive
- The rest of the semantics is unchanged (reductions, structural rules, robust safety)
Probabilistic RCF

Reduction for expressions: \( A \rightarrow_p A' \) (with \( 0 < p \leq 1 \))

- \( (\text{fun} \ x \rightarrow A) \ N \rightarrow_1 A\{N/x\} \)
- \( \text{let} \ x = M \text{ in} \ A \rightarrow_1 A\{M/x\} \)
- \( \text{let} \ (x_1, x_2) = (N_1, N_2) \text{ in} \ A \rightarrow_1 A\{N_1/x_1\}\{N_2/x_2\} \)
- \( a!M \overset{\rightarrow_1}{\Rightarrow} a? \rightarrow_1 M \quad \text{assert} \ C \rightarrow_1 () \)
- \( (\text{match} \ M \text{ with} h \ x \rightarrow A \text{ else} B) \rightarrow_1 \)
  \[
  \begin{cases}
    A\{N/x\} & \text{if } M = h \ N \text{ for some } N \\
    B & \text{otherwise}
  \end{cases}
  \]

- \( \text{sample} \rightarrow_{\frac{1}{2}} \text{true} \quad \text{sample} \rightarrow_{\frac{1}{2}} \text{false} \)

- \( M \rightarrow_1 M \)

\[
\begin{align*}
A \rightarrow_p A' & \\
\text{let} \ x = A \text{ in} B \rightarrow_p \text{let} \ x = A' \text{ in} B & \\
(A \vdash B) \rightarrow_p (A' \vdash B) & \\
(B \vdash A) \rightarrow_p (B \vdash A') & \\
A \Rightarrow B & \\
B \rightarrow_p B' & \\
B' \Rightarrow A' & \\
A \rightarrow_p A'
\end{align*}
\]
Probabilistic RCF

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

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

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

- We add a typing rule for sampling

\[
E \vdash \diamond
\]

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

- All typing theorems apply unchanged (one possible trace at a time)
MACs & CMA

Sample ideal functionalities for keyed hash functions
Sample ideal functionality:

**Keyed cryptographic hashes**

```
module MAC

type text = bytes
val macsize : bytes

val mac = b:bytes {Length(b)=macsize}

val GEN : unit -> key

val MAC : k:key -> t:text {Msg(k,t)} -> mac

val VERIFY : k:key -> t:text -> mac
    -> b:bool { b=true ⇒ Msg(k,t)}

module RPC

definition !k,q. Msg(k,Utf8(q)) <= Request(q)

let client q = 
    // precondition:
    // Request(q)
    ...
    send MAC k (utf8 q)

let server q =
    ... if VERIFY k (utf8 q) m
    then // we have Request(q)
    process q
```

**ideal F7 interface**

MAC keys are abstract

**MACs are fixed sized**

```plaintext
Msg is specified by protocols using MACs
```

```
"All verified messages have been MACed"
```

**sample protocol using MACs**
Sample ideal functionality:

**Keyed cryptographic hashes**

MAC keys are abstract

module MAC

type text = bytes
val macsize

type key

val macsize

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

predicate Msg of key * text

val GEN : unit -> key

val MAC : k:key -> t:text{Msg(k,t)} -> mac

val VERIFY: k:key -> t:text -> mac

-> b:bool{ b=true ⇒ Msg(k,t)}

ideal F7

interface

MACs are fixed sized

Msg is specified by protocols using MACs

"All verified messages have been MACed"

This can’t be true! (collisions)

concrete F# implementation (using .NET)

module MAC

open System.Security.Cryptography

let macsize = 20

let GEN() = randomBytes 16

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

let VERIFY k t m = (MAC k t = m)
Sample computational assumption:

Resistance to Chosen-Message Existential Forgery Attacks (CMA)

module CMA
open Mac

let k = GEN()
let log = ref []
let mac t =
  log := t::!log
  MAC k t
let verify t m =
  let v = VERIFY k t m in
  assert (not v || mem t !log)
v
MACs: interfaces and implementations

- a plain F# interface
- ... and its refinements
- Concrete Mac
- Ideal Mac
- Mac
- CMA
- some concrete implementation
- RPC
- cannot typecheck in F7!
- LINK
- some sample protocol
MACs: interfaces and implementations

a plain F# interface

... and its refinements

Concrete Mac

Ideal Mac

RPC

some concrete implementation

some error correcting wrapper

some sample protocol

Mac

Ideal Mac

RPC

CMA

LINK
MACs: interfaces and implementations

Concrete Mac | Ideal Mac | RPC | Adversary
---|---|---|---

is always safe (by typing)

is indistinguishable from

Concrete Mac | RPC | Adversary

is safe too, with overwhelming probability

Let $C$ p.p.t. CMA-secure with $\vdash C \sim I^C$.
Let $A$ p.p.t. with $I \vdash A : \text{unit}$.

**Theorem 3** (Asymptotic Safety for MAC). $C \cdot A$ is asymptotically safe.

**Theorem 4** (Ideal Functionality for MAC). $C \cdot A \approx_{\varepsilon} C \cdot F \cdot A$. 

Indistinguishability & Encryption

Secrecy by typing
Modules for secret plaintexts
Ideal functionalities for CPA and CCA2 encryption
Perfect Secrecy by Typing

• Secrecy is expressed using observational equivalences between systems that differ on their secrets
• We can prove secrecy by typing, relying on parametricity over some abstract type

\[ I_\alpha = \alpha, \ldots, x : T_\alpha, \ldots. \]
\[ P_\alpha \text{ range over pure modules such that } \vdash P_\alpha \sim I_\alpha. \]

**THEOREM 6** (Secrecy by Typing).
Let \( A \) such that \( I_\alpha \vdash A : bool. \)
For all \( P^0_\alpha \) and \( P^1_\alpha \), we have \( P^0_\alpha \cdot A \approx P^1_\alpha \cdot A. \)
Plaintext Modules

- Encryption is parameterized by a module that abstractly define plaintexts, with interface

```plaintext
module Plain
val plainsize: int
type plain
type repr = b:bytes{Length(b)=plainsize}
val plain : repr -> plain  // turning bytes to secrets
val repr : plain -> repr  // breaking secrecy!
```

The size of plaintext is fixed (as we cannot hide it)

If we remove the `repr` function, we get perfect secrecy by typing

```plaintext
val respond: plain -> plain  // sample protocol code
```

Plain may also include any code that operates on secrets
An Ideal Interface for CCA2-Secure Encryption

module PKENC
open Plain

val pksize: int
type skey
type pkey = b:bytes{ PKey(k) ∧ Length(b)=pksize }

val ciphersize: int
type cipher = b:bytes{Length(b)=ciphersize}

val GEN: unit -> pkey * skey
val ENC: pkey -> plain -> cipher
val DEC: skey -> cipher -> plain

- Its ideal implementation encrypts zeros instead of plaintexts so it never accesses plaintext representations, and can be typed parametrically
Sample computational assumption:
Indistinguishability against
Chosen Plaintexts & Ciphertexts Attacks

module CCA2
open RSA_OAEP
let k = GEN()
let log = ref []
let b = sample {true,false}
let encrypt p0 p1 =
    let p = if b then p0 else p1
    let e = ENC(k,p)
    log := e::!log
    e
let decrypt e =
    if e in !log
    then None
    else Some(DEC(k,e))

CCA game
(coded in F#)

Asymptotic security
a probabilistic polytime program
calling encrypt and decrypt guesses
which plaintexts are encrypted
only with a negligible advantage
Typed Secrecy from CCA2-Secure Encryption

**Theorem 7** (Asymptotic Secrecy).
Let $P^0$ and $P^1$ p.p.t. secret with $\vdash P^b \sim I_{\text{PLAIN}}$.
Let $C_{\text{ENC}}$ p.p.t. CCA2-secure with $I^c_{\text{PLAIN}} \vdash C_{\text{ENC}} \sim I^c_{\text{ENC}}$.
Let $A$ p.p.t. with $I_{\text{PLAIN}}, I_{\text{ENC}} \vdash A : \text{bool}$.

$$P^0 \cdot C_{\text{ENC}} \cdot A \approx_\epsilon P^1 \cdot C_{\text{ENC}} \cdot A.$$ 

**Theorem 8** (Ideal Functionality).
Let $P$ p.p.t. with $\vdash P \sim I^c_{\text{PLAIN}}$ (not necessarily secret).
Let $C_{\text{ENC}}$ p.p.t. CCA2-secure with $I^c_{\text{PLAIN}} \vdash C_{\text{ENC}} \sim I^c_{\text{ENC}}$.
Let $A$ p.p.t. with $I^c_{\text{PLAIN}}, I_{\text{ENC}} \vdash A$.

$$P \cdot C_{\text{ENC}} \cdot A \approx_\epsilon P \cdot C_{\text{ENC}} \cdot F_{\text{ENC}} \cdot A.$$
Variants: CPA & Authentication

• With **CPA-secure encryption**, we have a **weaker** ideal interface that demands ciphertext integrity before decryption

  ```plaintext
  predicate Encrypted of key * cipher
  val ENC: k:key -> plain -> c:cipher{Encrypted(k,c)}
  val DEC: k:key -> c:cipher{Encrypted(k,c)} -> plain
  ```

• With **authenticated encryption**, we have a **stronger** ideal interface that ensure plaintext integrity (much as MACs)

  ```plaintext
  predicate Msg of key * plain // defined by protocol
  val ENC: k:key -> p:plain{Msg(k,p)} -> cipher
  val DEC: k:key -> cipher -> p:plain{Msg(k,p)} option
  ```
Applications: Cryptographic Constructions

- We program and verify sample crypto constructions such as hybrid encryption and encrypt-then-MAC

```ocaml
module HybridEnc
let pks = PKEnc.pks + SymEnc.ciphersize
let ciphersize = PKEnc.ciphersize + SymEnc.ciphersize

let GEN() = PKEnc.GEN()
let ENC pk plain =
  let k = SymEnc.GEN()
  concat (PKEnc.ENC pk k) (SymEnc.ENC k plain)
let DEC sk cipher =
  let c0,c1 = split SymEnc.ciphersize cipher
  SymEnc.DEC (PKEnc.DEC sk c0) c1
```

- We prove these constructions secure by typechecking against interfaces of Plain, SymEnc, and PKEnc
Applications: Security Protocols

• We program and verify sample protocols (authenticated RPC, encrypted RPC)

• We typecheck larger protocols previously verified using F7 symbolic libraries (secure multiparty sessions)

• Case studies requiring custom crypto
  – **DKM** with cryptographic agility
  – **TLS 1.2** with ad hoc constructions
  – Private authentication with key-hiding & decoys
  – Private data processing with zero-knowledge proofs
We verify protocol implementations by **typechecking**
- Verification is modular
- We use abstract types and refinements to specify cryptography
- We capture standard (probabilistic polynomial time) assumptions
- We precisely control composition using typed interfaces
- Except for new crypto libraries, proofs are automated & fast

We are working towards certification using Coq
- So far, mechanized typing theorems
- New: self-certification for the typechecker
- See also CertiCrypt for code-based cryptographic proofs

Our approach and libraries are language-independent
- So far we use F# & F7