Formal/Computational Verification of Protocol Implementations by Typechecking

Cédric Fournet
Microsoft Research

with Karthik Bhargavan, Andy Gordon, …

http://research.microsoft.com/~fournet
http://msr-inria.inria.fr/projects/sec

Formal/Computational Verification of Protocol Implementations by Typechecking

1. Verifying implementations (Goal)
2. F7: refinement types for F# (Tool)
3. Modular Cryptographic Verification (Symbolic)
4. Cryptographic Soundness of Typechecking

By the way…

- We are offering
  2-year postdocs
  at MSR-INRIA (Orsay)
  http://msr-inria.inria.fr/projects/sec

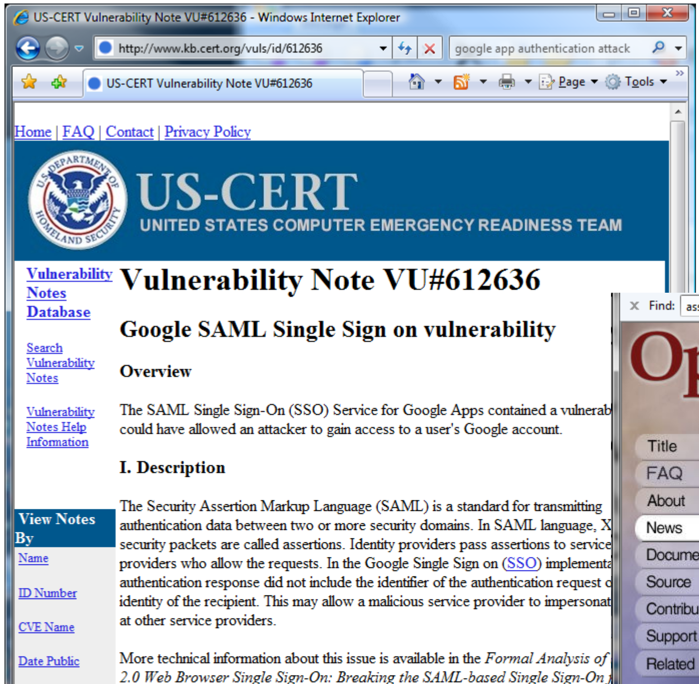
  and also

  3-month internships at
  Microsoft Research (Cambridge)
  PhDs, internships, visiting positions
  at MSR-INRIA (Orsay)

VERIFYING PROTOCOL IMPLEMENTATIONS

Cryptographic Protocols (Still) Go Wrong

- Protocols are designed by experienced cryptographers
  - And implemented by skilled programmers
- Still, serious flaws both in designs and implementations
  - Most standards got it wrong a few times (SSL, SSH, IPSEC)
  - Recent "logical" errors in Google single-sign-on, Microsoft Kerberos, OpenSSL certificate verification



US-CERT Vulnerability Note VU#612636 - Windows Internet Explorer

http://www.kb.cert.org/vuls/id/612636

Home | FAQ | Contact | Privacy Policy

US-CERT
UNITED STATES COMPUTER EMERGENCY READINESS TEAM

Vulnerability Notes Database

Search Vulnerability Notes

Vulnerability Notes Help Information

View Notes By
Name
ID Number
CVE Name
Date Public

Vulnerability Note VU#612636

Google SAML Single Sign on vulnerability

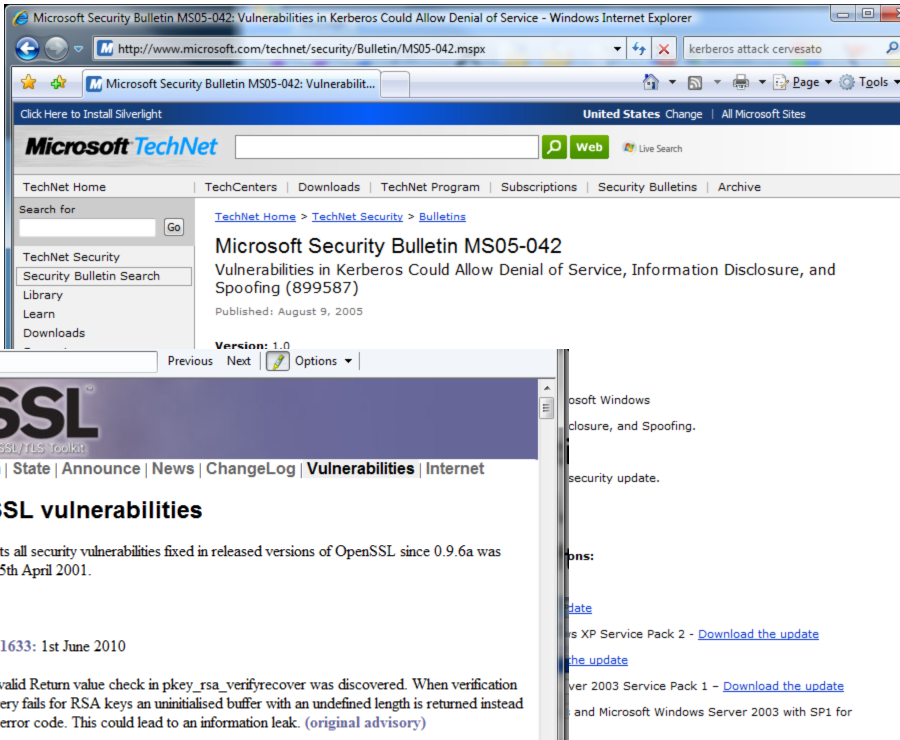
Overview

The SAML Single Sign-On (SSO) Service for Google Apps contained a vulnerab
could have allowed an attacker to gain access to a user's Google account.

I. Description

The Security Assertion Markup Language (SAML) is a standard for transmitting authentication data between two or more security domains. In SAML language, X security packets are called assertions. Identity providers pass assertions to service providers who allow the requests. In the Google Single Sign on (SSO) implementa authentication response did not include the identifier of the authentication request o identity of the recipient. This may allow a malicious service provider to impersonat at other service providers.

More technical information about this issue is available in the *Formal Analysis of 2.0 Web Browser Single Sign-On: Breaking the SAML-based Single Sign-On*



Microsoft Security Bulletin MS05-042: Vulnerabilities in Kerberos Could Allow Denial of Service - Windows Internet Explorer

http://www.microsoft.com/technet/security/Bulletin/MS05-042.mspx

Microsoft TechNet

TechNet Home | TechCenters | Downloads | TechNet Program | Subscriptions | Security Bulletins | Archive

TechNet Security
Security Bulletin Search
Library
Learn
Downloads

TechNet Home > TechNet Security > Bulletins

Microsoft Security Bulletin MS05-042

Vulnerabilities in Kerberos Could Allow Denial of Service, Information Disclosure, and Spoofing (899587)

Published: August 9, 2005

Version: 1.0

osoft Windows

closure, and Spoofing.

security update.

ons:

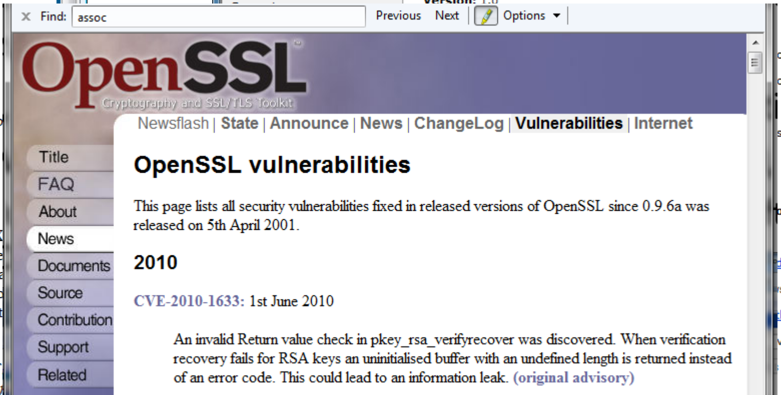
date

s XP Service Pack 2 - Download the update

the update

ver 2003 Service Pack 1 – Download the update

and Microsoft Windows Server 2003 with SP1 for



OpenSSL
Cryptography and SSL/TLS Toolkit

Newsflash | State | Announce | News | ChangeLog | Vulnerabilities | Internet

Title
FAQ
About
News
Documents
Source
Contribution
Support
Related

OpenSSL vulnerabilities

This page lists all security vulnerabilities fixed in released versions of OpenSSL since 0.9.6a was released on 5th April 2001.

2010

CVE-2010-1633: 1st June 2010

An invalid Return value check in pkey_rsa_verifyrecover was discovered. When verification recovery fails for RSA keys an uninitialised buffer with an undefined length is returned instead of an error code. This could lead to an information leak. (original advisory)

Symbolic vs Computational Cryptography

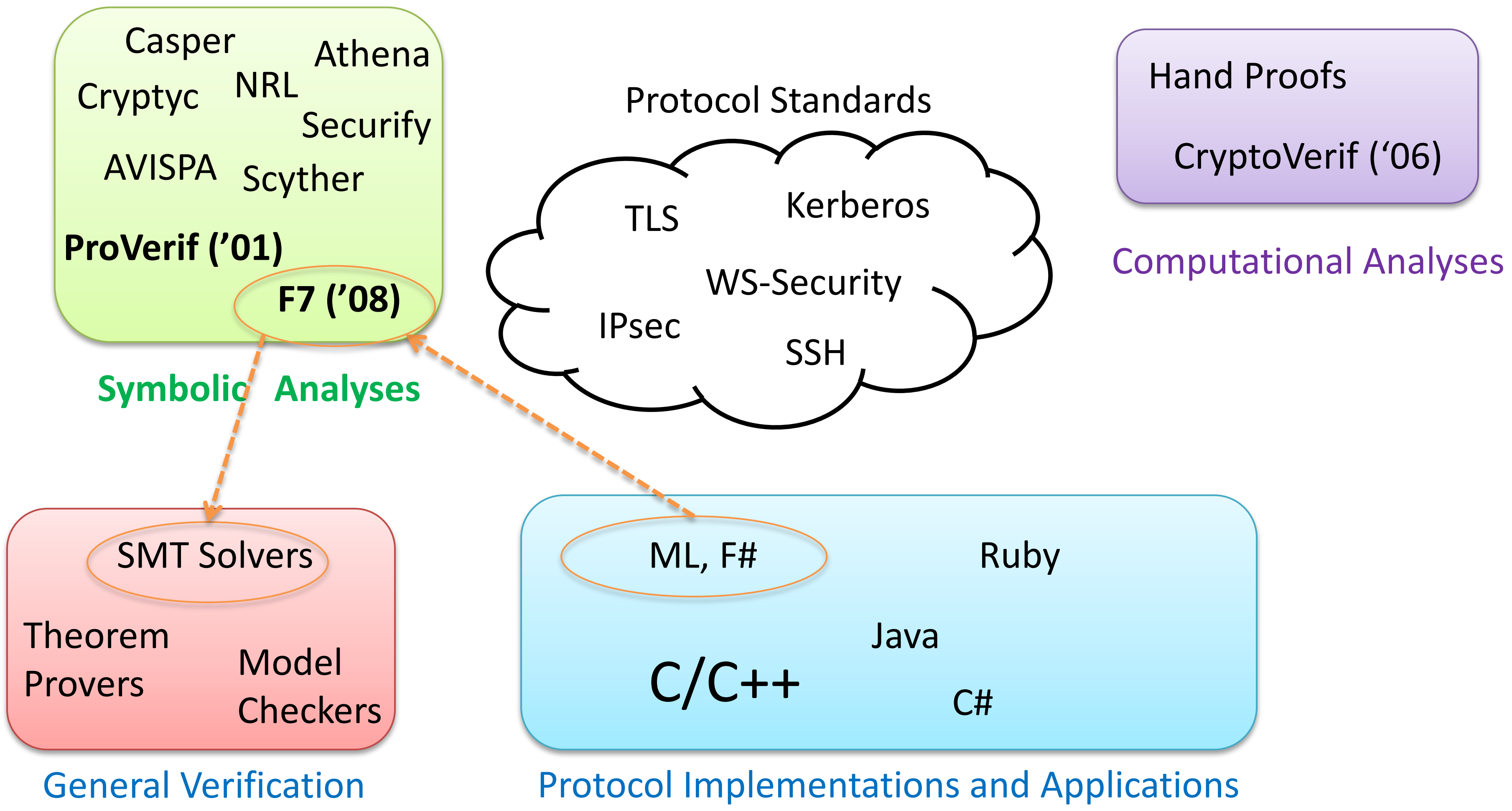
- Two verification approaches have been successfully applied to protocols and programs that use cryptography:

  Symbolic approach (Needham-Schroeder, Dolev-Yao, ... late 70's)
  - Structural view of protocols, using formal languages and methods
  - Compositional, automated verification tools, scales to large systems
  - Too abstract?

  Computational approach (Yao, Goldwasser, Micali, Rivest, ... early 80's)
  - More concrete, algorithmic view; more widely accepted
  - Adversaries range over probabilistic Turing machines
    Cryptographic materials range over bitstrings
  - Delicate (informal) game-based reduction proofs; poor scalability

- Can we get the best of both worlds? Much ongoing work on *computational soundness* for symbolic cryptography
- Can we verify real-world protocols?

Specs, Code, and Formal Tools



Models vs implementations

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

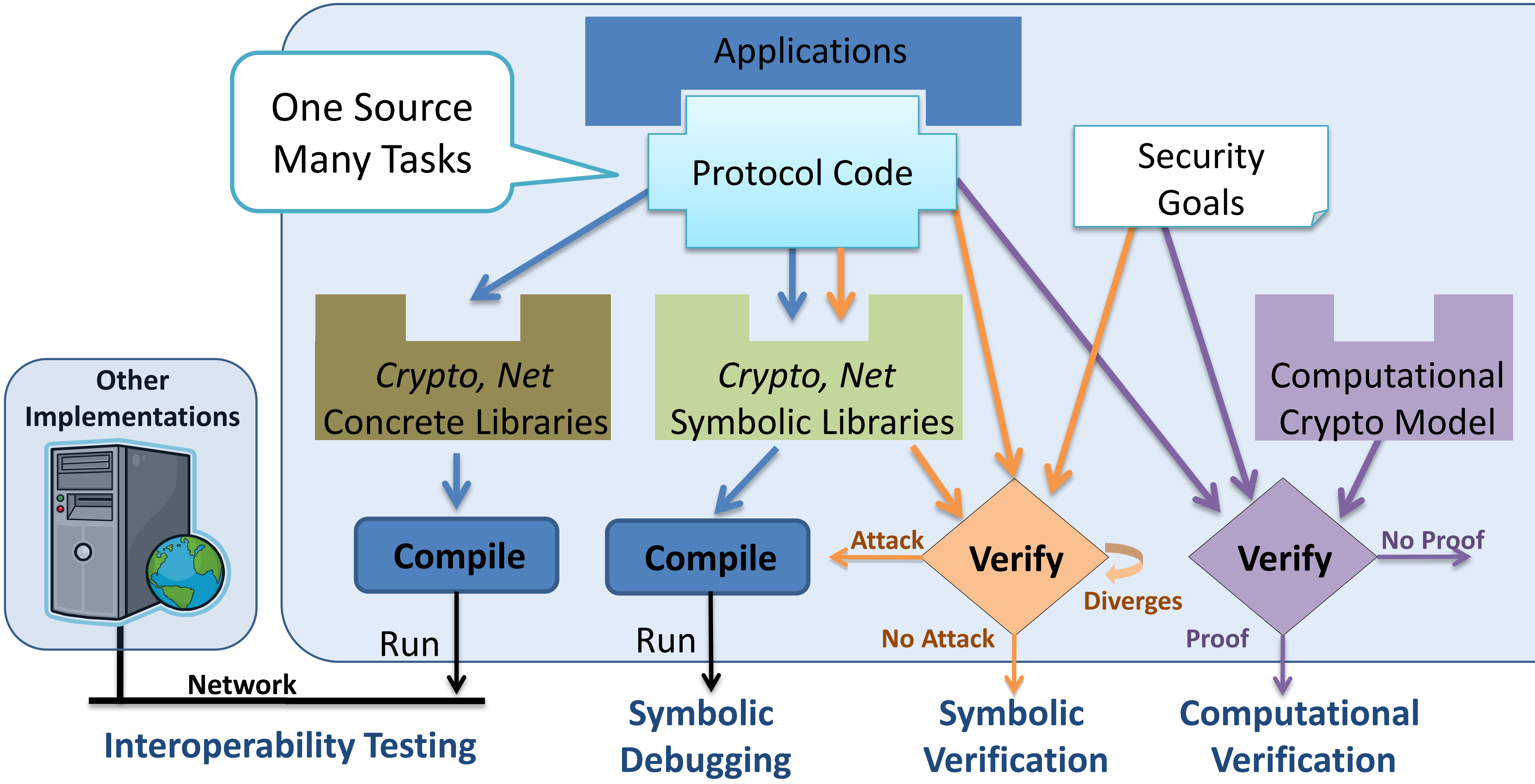
- Models are short, abstract, hand-written
  - They ignore large functional parts of implementations
  - Their formulation is driven by verification techniques
  - It is easy to write models that are safe but dysfunctional (testing & debugging is difficult)

- 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 automatically extract models from protocol code
  - We develop models as executable code too (reference implementations)
- 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, testing, verification tools

Verifying Protocol Code (not just specs)

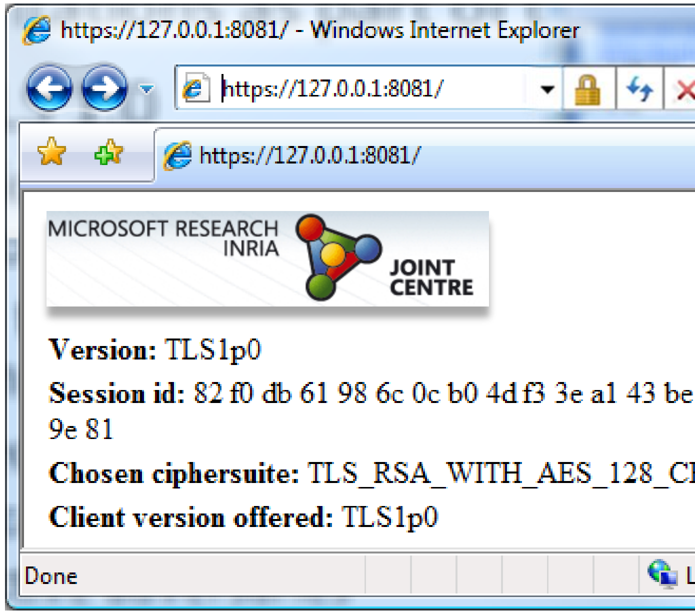


Source language: F#

- F#, a dialect of ML
  http://research.microsoft.com/fsharp
  “Combining the strong typing, scripting and productivity of ML with the efficiency, stability, libraries, cross-language working and tools of .NET.”

- Interop with production code
- Clean strongly-typed semantics
  - Modular programming based on strong interfaces
  - Algebraic data types with pattern matching useful for symbolic cryptography message formats

TLS in F# [CCS'08]


https://127.0.0.1:8081/ - Windows Internet Explorer

https://127.0.0.1:8081/

MICROSOFT RESEARCH INRIA JOINT CENTRE

Version: TLS1p0

Session id: 82 f0 db 61 98 6c 0c b0 4d f3 3e a1 43 be
9e 81

Chosen ciphersuite: TLS_RSA_WITH_AES_128_C

Client version offered: TLS1p0

Done

We implemented a subset of TLS (10 kLOC)

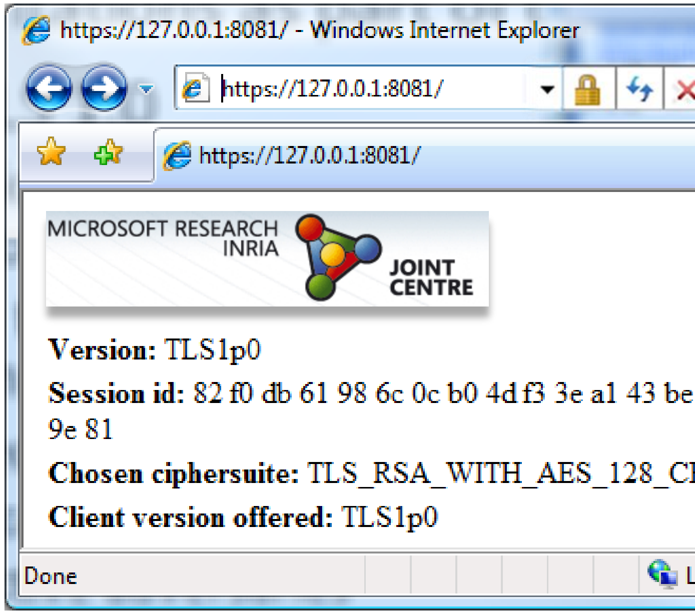
- Supports SSL3.0, TLS1.0, TLS1.1 with session resumption
- Supports any ciphersuite using DES, AES, RC4, SHA1, MD5

We tested it on a few basic scenarios, e.g.

1. An HTTPS client to retrieves pages (interop with IIS, Apache, and F# servers)
2. An HTTPS server to serve pages (interop with IE, Firefox, Opera, and F# client)

We verified our implementation (symbolically & computationally)

TLS in F# [CCS'08]


https://127.0.0.1:8081/ - Windows Internet Explorer
https://127.0.0.1:8081/
https://127.0.0.1:8081/
MICROSOFT RESEARCH INRIA JOINT CENTRE
Done

Version: TLS1p0

Session id: 82 f0 db 61 98 6c 0c b0 4d f3 3e a1 43 be 9e 81

Chosen ciphersuite: TLS_RSA_WITH_AES_128_CI

Client version offered: TLS1p0

We used "global" cryptographic verifiers, treating our F# code as a monster protocol

We reached the limit of this proof method:

- "Automated" verification is fragile, involves code refactoring and expertise
- Verification takes hours on a large machine
- Adding new profiles or composing sub-protocols leads to divergence
- We can't directly reason about protocols using TLS as a component

We need compositional verification techniques
➔ Let's use types!

Verification tool: **Refinement types**

A *refinement type* is a base type qualified with a logical formula;
the formula can express invariants, preconditions, postconditions, …

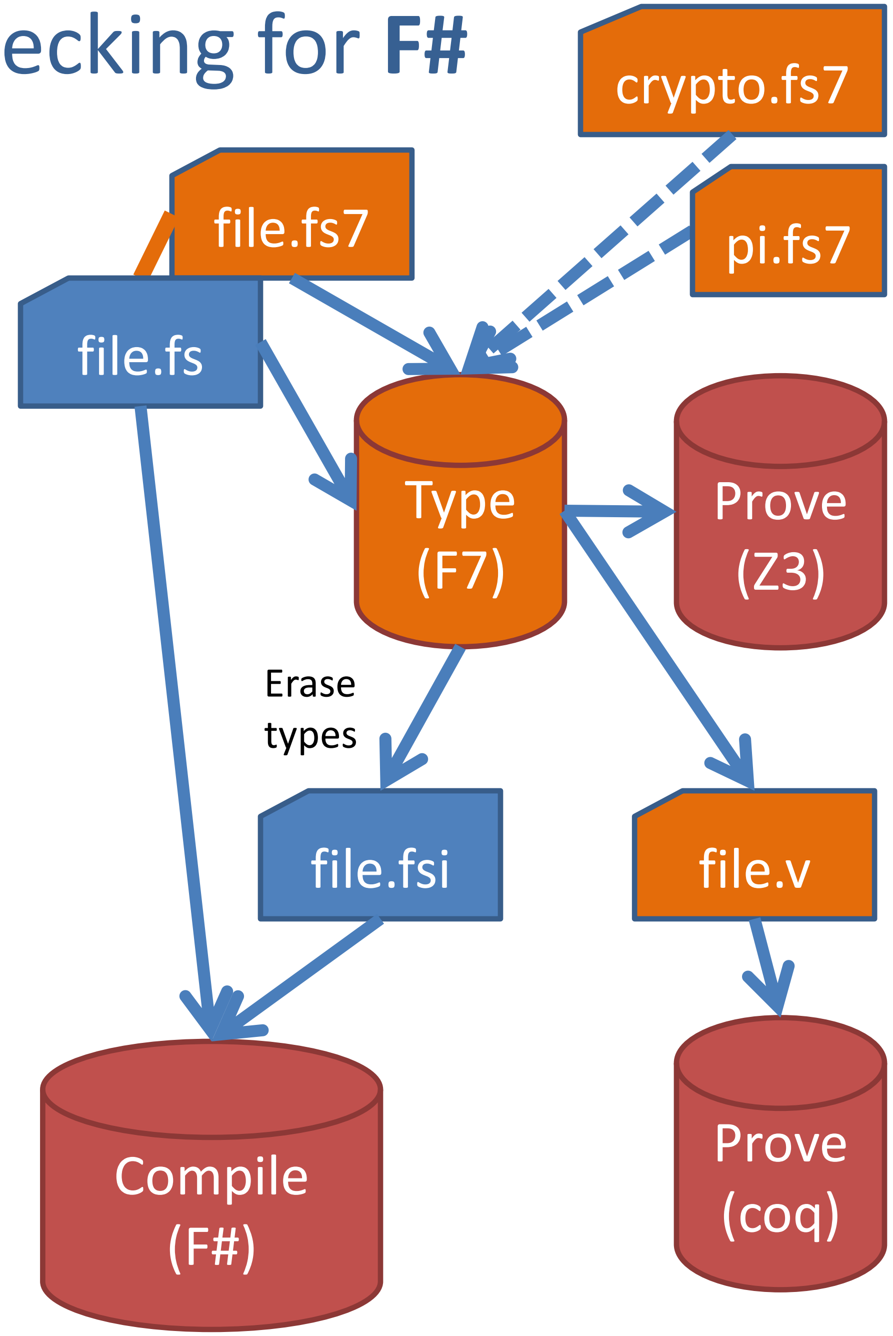
Refinement types are types of the form $x : T\{C\}$ where
– T is the base type,
– x refers to the result of the expression, and
– C is a logical formula

The values of this type are the values M of type T such that $C\{M/x\}$ holds.

Examples:
– $n : \text{int}\{n \geq 0\}$ is the type of positive integers
– $k : \text{bytes}\{KeyAB(k, a, b)\}$ is the type of byte arrays used as keys by a and b

F7: refinement typechecking for **F#**

- We write extended interfaces
  - We typecheck implementations
  - We generate .fsi interfaces by erasure from .fs7
- We do some type inference
  - Plain F# types as usual
  - Refinements require annotations
- We call Z3, an SMT prover, on each proof obligation
- We can also generate coq proof obligations
  - Selected interactive proofs
  - Theorems *assumed* for typechecking & Z3



Typed Theory

A CORE LANGUAGE WITH REFINEMENT TYPES

For more details, see tutorial on **Principles and Applications of Refinement Types** with Andy Gordon, in *International Summer School Logics and Languages for Reliability and Security, Marktoberdorf*. October 2009. Also Technical Report MSR-TR-2009-147.

A formal core for F# and F7 (outline)

- An assembly of standard parts, generalizing ad hoc constructions in language-based security
  - **FPC** (Plotkin 1985, Gunter 1992) – core of ML and Haskell
  - Concurrency in style of the **pi-calculus** (Milner, Parrow, Walker 1989) but for a lambda-calculus (like 80s languages PFL, Poly/ML, CML)
  - Formal crypto is derivable by coding up **seals** (Morris 1973, Sumii and Pierce 2002), not primitive as in spi or applied pi calculi
  - Security specs via **assume/assert** (Floyd, Hoare, Dijkstra 1970s), generalizing eg correspondences (Woo and Lam 1992)
  - To check assertions statically, rely on dependent functions and pairs with subtyping (Cardelli 1988) and **refinement types** (Pfenning 1992, ...) aka **predicate subtyping** (as in PVS, and more recently Russell)

The Core Language (FPC):

x, y, z	variable
$h ::=$	value constructor
inl	left constructor of sum type
inr	right constructor of sum type
fold	constructor of iso-recursive type
$M, N ::=$	value
x	variable
$()$	unit
fun $x \to A$	function (scope of x is A)
(M, N)	pair
$h\ M$	construction
$A, B ::=$	expression
M	value
$M\ N$	application
$M = N$	syntactic equality
let $x = A$ **in** B	let (scope of x is B)
let $(x, y) = M$ **in** A	pair split (scope of x, y is A)
match M **with** $h\ x \to A$ **else** B	constructor match (scope of x is A)

The Reduction Relation: $A \rightarrow A'$

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

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

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

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

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

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

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

Communications and Concurrency:

$A, B ::=$	expression
$\dots$	as before
$(\nu a)A$	local channel
$a!M$	transmission of M on channel a
$a?$	receive message off channel
$A \Rsh B$	parallel composition

$a!M \Rsh a? \rightarrow M$

Communication step

$A \rightarrow A' \quad \text{if } A \Rrightarrow B, B \rightarrow B', B' \Rrightarrow A'$

Reductions step are
"up to structural rearrangements"

$$
\begin{array}{ll}
 & a!0 \Rsh a!1 \Rsh (\textbf{let } x = a? \textbf{ in } (a!(x{+}2) \Rsh x)) \\
\equiv & a!1 \Rsh \textbf{let } x = (a!0 \Rsh a?) \textbf{ in } (a!(x{+}2) \Rsh x) \\
\rightarrow & a!1 \Rsh \textbf{let } x = 0 \textbf{ in } (a!(x{+}2) \Rsh x) \\
\rightarrow & a!1 \Rsh a!2 \Rsh 0
\end{array}
$$

$$
\begin{array}{ll}
 & a!0 \Rsh a!1 \Rsh (\textbf{let } x = a? \textbf{ in } (a!(x{+}2) \Rsh x)) \\
\equiv\rightarrow\rightarrow & a!0 \Rsh a!3 \Rsh 1
\end{array}
$$

Example: Concurrent ML:

$(T)\mathsf{chan} \triangleq (T \to \mathsf{unit}) * (\mathsf{unit} \to T)$

$\mathsf{chan} \triangleq \mathbf{fun}\, x \to (\nu a)(\mathbf{fun}\, x \to a!x, \mathbf{fun}\, \_ \to a?)$	
$\mathsf{send} \triangleq \mathbf{fun}\, c\, x \to \mathbf{let}\ (s,r) = c\ \mathbf{in}\ s\ x$	send x on c
$\mathsf{recv} \triangleq \mathbf{fun}\, c \to \mathbf{let}\ (s,r) = c\ \mathbf{in}\ r\ ()$	block for x on c
$\mathsf{fork} \triangleq \mathbf{fun}\, f \to (f() \upharpoonright ())$	run f in parallel

Example: Mutable State:

$(T)\mathbf{ref} \triangleq (T)\mathsf{chan}$

$\mathbf{ref}\ M \triangleq \mathbf{let}\ r = \mathsf{chan}\ \texttt{"r"}\ \mathbf{in}\ \mathsf{send}\ r\ M; r$	new reference to M
$\mathsf{deref}\ M \triangleq \mathbf{let}\ x = \mathsf{recv}\ M\ \mathbf{in}\ \mathsf{send}\ M\ x; x$	dereference M
$M := N \triangleq \mathbf{let}\ x = \mathsf{recv}\ M\ \mathbf{in}\ \mathsf{send}\ M\ N$	update M with N

LOGICAL 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 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:

$C ::= p(M_1,\ldots,M_n) \mid M = M' \mid C \wedge C' \mid C \vee C' \mid \neg C \mid C \Rightarrow C' \mid \forall x.C \mid \exists x.C$

$\{C_1,\ldots,C_n\} \vdash C$	deducibility relation

Assume and Assert:

$A, B ::=$	expression
$\quad \ldots$	as before
$\quad$ **assume** C	assumption of formula C
$\quad$ **assert** C	assertion of formula C

assume $C \Rrightarrow$ **assume** $C \Rsh ()$

assert $C \to ()$

Semantics: expression safety

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

$$\mathbf{S} ::= (\nu a_1)\ldots(\nu a_\ell)\left((\prod_{i\in 1..m} \textbf{assume } C_i) \upharpoonright (\prod_{j\in 1..n} c_j!M_j) \upharpoonright (\prod_{k\in 1..o} \mathscr{L}_k\{e_k\}) \right)$$

active assumptions — pending messages — running threads

- An expression is **safe** when, for all runs of A, **all assertions succeed**

Are these expressions safe?

assert (p ∧ q ⇒ q)
assert (p ∨ q ⇒ q)
assume (p ⇒ q); **assert** (p ∨ q ⇒ q)

let $x = 0$ **in assert**$(x = 1)$
a!0 ↱ a!1 ↱ (**let** x = a? **in assert** $(x{=}0 \vee x{=}1)$)
a!0 ↱ a!1 ↱ (**let** x = a? **in assert** $x{=}1$)
a!0 ↱ a!1 ↱ (**let** x = a? **in if** $x > 0$ **then assert** $x{=}1$)
. . .

A FIRST 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

```
type facts = CanRead of string | CanWrite of string

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 ∀x. CanWrite(x) → 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)
```

Logging dynamic events

- Security policies often stated in terms of dynamic events such as role activations or data checks
- We mark such events by adding formulas to the log with **assume**

```
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

How to prove safety for all runs of a system?

SAFETY BY TYPING

Starting Point: The Type System for FPC:

$$\frac{E \vdash \diamond \quad (x:T) \in E}{E \vdash x : T} \qquad \frac{E \vdash A : T \quad E, x:T \vdash B : U}{E \vdash \textbf{let}\ x = A\ \textbf{in}\ B : U}$$

$$\frac{E \vdash \diamond}{E \vdash () : \mathsf{unit}} \qquad \frac{E \vdash M : T \quad E \vdash N : U}{E \vdash M = N : \mathsf{unit} + \mathsf{unit}}$$

$$\frac{E, x:T \vdash A : U}{E \vdash \textbf{fun}\, x \to A : (T \to U)} \qquad \frac{E \vdash M : (T \to U) \quad E \vdash N : T}{E \vdash M\ N : U}$$

$$\frac{E \vdash M : T \quad E \vdash N : U}{E \vdash (M,N) : (T \times U)} \qquad \frac{E \vdash M : (T \times U) \quad E, x:T, y:U \vdash A : V}{E \vdash \textbf{let}\ (x,y) = M\ \textbf{in}\ A : V}$$

$$\frac{h : (T,U) \quad E \vdash M : T \quad E \vdash U}{E \vdash h\ M : U} \qquad \frac{E \vdash M : T \quad h : (H,T) \quad E, x:H \vdash A : U \quad E \vdash B : U}{E \vdash \textbf{match}\ M\ \textbf{with}\ h\ x \to A\ \textbf{else}\ B : U}$$

$\mathsf{inl}{:}(T, T+U) \qquad \mathsf{inr}{:}(U, T+U) \qquad \mathsf{fold}{:}(T\{\mu\alpha.T/\alpha\}, \mu\alpha.T)$

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 logic judgment $E \vdash C$ meaning that *C* follows from the refinement types in *E*
3. To manage refinement formulas, we need (1) dependent versions of the function and pair types, and (2) subtyping

- A value of $x{:}T \to U$ is a function M such that if N has type T, then $M\ N$ has type $U\{N/x\}$.
- A value of $x{:}T \times U$ is a pair (M,N) such that M has type T and N has type $U\{M/x\}$.
- If $A : T$ and $T <: U$ then $A : U$.

Syntax of RCF Types:

$H, T, U, V ::=$	type
unit	unit type
$x{:}T \to U$	dependent function type (scope of x is U)
$x{:}T \times U$	dependent pair type (scope of x is U)
$T + U$	disjoint sum type
$\mu\alpha.T$	iso-recursive type (scope of α is T)
α	iso-recursive type variable
$\{x : T \mid C\}$	refinement type (scope of x is C)

$\{C\} \triangleq \{\_ : \mathsf{unit} \mid C\}$	ok-type
$\mathsf{bool} \triangleq \mathsf{unit} + \mathsf{unit}$	Boolean type

Starting Point: The Type System for FPC:

$$\frac{E \vdash \diamond \quad (x:T) \in E}{E \vdash x : T} \qquad \frac{E \vdash A : T \quad E, x:T \vdash B : U}{E \vdash \textbf{let}\ x = A\ \textbf{in}\ B : U}$$

$$\frac{E \vdash \diamond}{E \vdash () : \mathsf{unit}} \qquad \frac{E \vdash M : T \quad E \vdash N : U}{E \vdash M = N : \mathsf{unit} + \mathsf{unit}}$$

$$\frac{E, x:T \vdash A : U}{E \vdash \textbf{fun}\, x \to A : (T \to U)} \qquad \frac{E \vdash M : (T \to U) \quad E \vdash N : T}{E \vdash M\ N : U}$$

$$\frac{E \vdash M : T \quad E \vdash N : U}{E \vdash (M, N) : (T \times U)} \qquad \frac{E \vdash M : (T \times U) \quad E, x:T, y:U \vdash A : V}{E \vdash \textbf{let}\ (x, y) = M\ \textbf{in}\ A : V}$$

$$\frac{h : (T, U) \quad E \vdash M : T \quad E \vdash U}{E \vdash h\ M : U} \qquad \frac{E \vdash M : T \quad h : (H, T) \quad E, x:H \vdash A : U \quad E \vdash B : U}{E \vdash \textbf{match}\ M\ \textbf{with}\ h\ x \to A\ \textbf{else}\ B : U}$$

$\mathsf{inl}{:}(T, T+U) \quad \mathsf{inr}{:}(U, T+U) \quad \mathsf{fold}{:}(T\{\mu\alpha.T/\alpha\}, \mu\alpha.T)$

Starting Point: The Type System for FPC:

$$\frac{E \vdash \diamond \quad (x:T) \in E}{E \vdash x : T} \qquad \frac{E \vdash A : T \quad E, x:T \vdash B : U}{E \vdash \textbf{let}\ x = A\ \textbf{in}\ B : U}$$

$$\frac{E \vdash \diamond}{E \vdash () : \mathsf{unit}} \qquad \frac{E \vdash M : T \quad E \vdash N : U}{E \vdash M = N : \mathsf{unit} + \mathsf{unit}}$$

$$\frac{E, x:T \vdash A : U}{E \vdash \textbf{fun}\, x \to A : (T \to U)} \qquad \frac{E \vdash M : (T \to U) \quad E \vdash N : T}{E \vdash M\, N : U}$$

$$\frac{E \vdash M : T \quad E \vdash N : U}{E \vdash (M,N) : (T \times U)} \qquad \frac{E \vdash M : (T \times U) \quad E, x:T, y:U \vdash A : V}{E \vdash \textbf{let}\ (x,y) = M\ \textbf{in}\ A : V}$$

$$\frac{h : (T,U) \quad E \vdash M : T \quad E \vdash U}{E \vdash h\, M : U} \qquad \frac{E \vdash M : T \quad h : (H,T) \quad E, x:H \vdash A : U \quad E \vdash B : U}{E \vdash \textbf{match}\ M\ \textbf{with}\ h\, x \to A\ \textbf{else}\ B : U}$$

$\mathsf{inl}{:}(T, T+U) \qquad \mathsf{inr}{:}(U, T+U) \qquad \mathsf{fold}{:}(T\{\mu\alpha.T/\alpha\}, \mu\alpha.T)$

Rules for refinements

We can refine any type with any formula that follows from *E*

$$\frac{E \vdash M : T \quad E \vdash C\{M/x\}}{E \vdash M : \{x : T \mid C\}}$$

$$\frac{E \vdash T <: T'}{E \vdash \{x : T \mid C\} <: T'} \qquad \frac{E \vdash T <: T' \quad E, x : T \vdash C}{E \vdash T <: \{x : T' \mid C\}}$$

Rules for assume and assert

$$\frac{E \vdash \diamond \quad \mathit{fnfv}(C) \subseteq \mathit{dom}(E)}{E \vdash \mathbf{assume}\ C : \{\_ : \mathsf{unit} \mid C\}} \qquad \frac{E \vdash C}{E \vdash \mathbf{assert}\ C : \mathsf{unit}}$$

We can assume any formula

We can assert any formula that follows from *E*

Type Judgements & Type safety

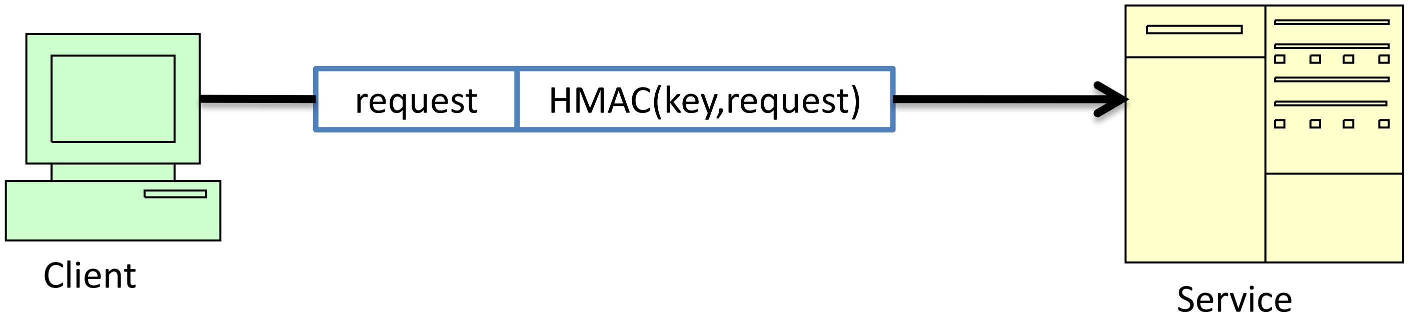
$E ::= x_1 : T_1, \ldots, x_n : T_n$	environment
$E \vdash \diamond$	E is syntactically well-formed
$E \vdash T$	in E, type T is syntactically well-formed
$E \vdash C$	formula C is derivable from E
$E \vdash T <: U$	in E, type T is a subtype of type U
$E \vdash A : T$	in E, expression A has type T

Lemma If $\varnothing \vdash \mathbf{S} : T$ then **S** is statically safe.
Lemma If $E \vdash A : T$ and $A \Rrightarrow A'$ then $E \vdash A' : T$.
Lemma If $E \vdash A : T$ and $A \longrightarrow A'$ then $E \vdash A' : T$.

Theorem If $\varnothing \vdash A : T$ then A is safe.
(For any A' and **S** such that $A \longrightarrow^* A'$ and $A' \Rrightarrow \mathbf{S}$
we need that **S** is statically safe.)

Summary on RCF

- RCF supports
  - functional programming a la ML and Haskell
  - concurrency in the style of process calculus, and
  - refinement types allowing correctness properties to be stated in the style of dependent type theory.
- Security applications
  - Access control and authorization policies
  - Information flow control
  - Cryptographic protocols (next)
- Implementations: F7, …, Fable, Fine, Fx, Fe, …, Aura, …



Programming Example:

MESSAGE AUTHENTICATION

Formal/Computational Verification of Protocol Implementations by Typing

1. Verifying reference implementations (Goal)
2. F7: refinement types for F# (Tool)

3. Modular Cryptographic Verification (Symbolic)
4. Computational Soundness of Typechecking

Refinement types (review)

A *refinement type* is a base type qualified with a logical formula;
the formula can express invariants, preconditions, postconditions, …

Refinement types are types of the form $x : T\{C\}$ where
– T is the base type,
– x refers to the result of the expression, and
– C is a logical formula

The values of this type are the values M of type T such that $C\{M/x\}$ holds.

Examples:
– $n : \mathrm{int}\{n \geq 0\}$ is the type of positive integers
– $k : \mathrm{bytes}\{KeyAB(k, a, b)\}$ is the type of byte arrays used as keys by a and b
– $x : \mathrm{str}\{Request(a, b, x)\}$ is the type of strings sent as requests from a to b

Our crypto libraries for F7 v2.0

LOGICAL INVARIANTS FOR SYMBOLIC CRYPTOGRAPHY

Invariants for Cryptographic Structures

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

(2) We use a "Public" predicate and events keep track of protocols.

– $Pub(x)$ holds when the value x is known to the adversary.

– $Request(a,b,x)$ holds when a intends to send message x to b.

(3) We define logical invariants on cryptographic structures.

– $Bytes(x)$ holds when the value x appears in the protocol run.

– $KeyAB(k_{ab},a,b)$ holds when key k_{ab} is shared between a and b.

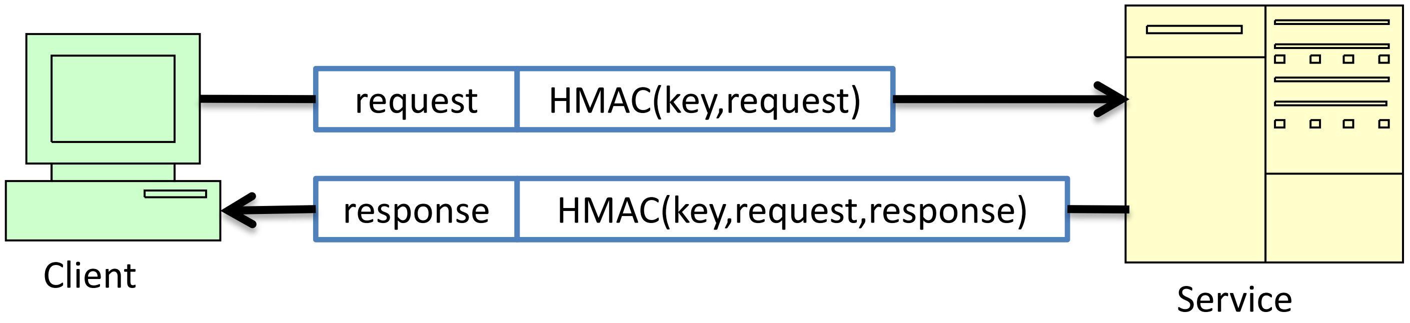
– After verifying the MAC (if no principals are compromised), $KeyAB(k_{ab},a,b) \wedge Bytes(hash\ k_{ab}\ x) \Longrightarrow Request(a,b,x)$.

(4) We verify that the protocol code maintains these invariants (by typing)

– $KeyAB(k_{ab},a,b) \wedge Request(a,b,x)$ is a precondition for computing $hash\ k_{ab}\ x$

Sample protocol: an authenticated RPC

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



Informal Description

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

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

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

Events record the main steps of the protocol:

- *Request*(a,b,s) before a sends message 1;
- *Response*(a,b,s,t) before b sends message 2;
- *KeyAB*(k,a,b) before issuing a key k associated with a and b;
- *Bad*(a) before leaking any key associated with a.

Authentication goals are stated in terms of events:

- *RecvRequest*(a,b,s) after b accepts message 1;
- *RecvResponse*(a,b,s,t) after a accepts message 2;

where the predicates *RecvRequest* and *RecvResponse* are defined by

$$\forall a,b,s.\ RecvRequest(a,b,s) \Leftrightarrow (Request(a,b,s) \vee Bad(a) \vee Bad(b))$$

$$\forall a,b,s,t.\ RecvResponse(a,b,s,t) \Leftrightarrow (Request(a,b,s) \wedge Response(a,b,s,t)) \vee Bad(a) \vee Bad(b)$$

F# Implementation

$$
\begin{array}{l}
1.\ a \rightarrow b :\ \mathit{utf8}\ s \mid (\mathit{hmacsha1}\ k_{ab}\ (\mathit{request}\ s)) \\
2.\ b \rightarrow a :\ \mathit{utf8}\ t \mid (\mathit{hmacsha1}\ k_{ab}\ (\mathit{response}\ s\ t))
\end{array}
$$

Our F# implementation of the protocol:

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

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

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

Test

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

The messages exchanged over TCP are:

```
Connecting to localhost:8080
Sending {BgAyICsgMj9mhJa7iDAcW3Rrk...} (28 bytes)
Listening at ::1:8080
Received Request 2 + 2?
Sending {AQA0NccjcuL/WOaYSOGGtOtPm...} (23 bytes)
Received Response 4
```

Modelling Opponents as F# Programs

We program a protocol-specific interface for the opponent:

```
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):

```
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 $I_L, I_R \vdash O : \mathrm{unit}$, then $X[O]$ is semantically safe.

Security Proof: MACs

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

For MACs, this interface is

Refinement Types for MACs in the *Crypto* library:

private val *hmac_keygen*: unit $\rightarrow k$:key$\{MKey(k)\}$
val *hmacsha1*:
  k:key $\rightarrow$
  b:bytes$\{\ (MKey(k) \wedge MACSays(k,b)) \vee (Pub(k) \wedge Pub(b))\ \} \rightarrow$
  h:bytes$\{\ IsMAC(h,k,b) \wedge (Pub(b) \Rightarrow Pub(h))\ \}$
val *hmacsha1Verify*:
  k:key$\{MKey(k) \vee Pub(k)\} \rightarrow b$:bytes $\rightarrow h$:bytes $\rightarrow$ unit$\{IsMAC(h,k,b)\}$

(C1. By expanding the definition of IsMAC)
$\forall h,k,b.\ IsMAC(h,k,b) \wedge Bytes(h) \Rightarrow MACSays(k,b) \vee Pub(k)$
(C2. MAC keys are public iff they may be used with any logical payload)
$\forall k.\ MKey(k) \Rightarrow (Pub(k) \Leftrightarrow \forall m.\ MACSays(k,m))$

Security proof: 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)

$\forall v,v',s,s',t'.\ (\mathit{Requested}(v,s) \wedge \mathit{Responded}(v',s',t')) \Rightarrow (v \neq v')$

(request is injective)

$\forall v,v',s,s'.\ (\mathit{Requested}(v,s) \wedge \mathit{Requested}(v',s') \wedge v = v') \Rightarrow (s = s')$

(response is injective)

$\forall v,v',s,s',t,t'.$

$(\mathit{Responded}(v,s,t) \wedge \mathit{Responded}(v',s',t') \wedge v = v') \Rightarrow (s = s' \wedge t = t')$

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)
$\forall a,b,k,m.\ KeyAB(k,a,b) \Rightarrow (\ MACSays(k,m) \Leftrightarrow$
$(\ (\exists s.\ Requested(m,s) \wedge Request(a,b,s)) \vee$
$(\exists s,t.\ Responded(m,s,t) \wedge Response(a,b,s,t)) \vee$
$(Bad(a) \vee Bad(b))))$

(KeyAB Injective)
$\forall k,a,b,a',b'.\ KeyAB(k,a,b) \wedge KeyAB(k,a',b') \Rightarrow (a=a') \wedge (b=b')$

(KeyAB Pub Bad)
$\forall a,b,k.\ KeyAB(k,a,b) \wedge Pub(k) \Rightarrow Bad(a) \vee Bad(b)$

(KeyAB MACSays) is a *definition* for the library predicate *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)

$\forall a,b,k,m.\ \mathit{KeyAB}(k,a,b) \Rightarrow (\ \mathit{MACSays}(k,m) \Leftrightarrow$
$(\ (\exists s.\ \mathit{Requested}(m,s) \wedge \mathit{Request}(a,b,s)) \vee$
$(\exists s,t.\ \mathit{Responded}(m,s,t) \wedge \mathit{Response}(a,b,s,t)) \vee$
$(\mathit{Bad}(a) \vee \mathit{Bad}(b))))$

(KeyAB Injective)

$\forall k,a,b,a',b'.\ \mathit{KeyAB}(k,a,b) \wedge \mathit{KeyAB}(k,a',b') \Rightarrow (a=a') \wedge (b=b')$

(KeyAB Pub Bad)

$\forall a,b,k.\ \mathit{KeyAB}(k,a,b) \wedge \mathit{Pub}(k) \Rightarrow \mathit{Bad}(a) \vee \mathit{Bad}(b)$

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

Symbolic Crypto Models

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) $\forall x,y.\ Pub(x) \wedge Pub(y) \Rightarrow Pub(pair(x,y))$
  - 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 $\forall k,a,b.\ PubKey(k,a) \wedge PubKey(k,b) \Rightarrow a = b$

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

Refined Modules

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

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

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

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):

$a \rightarrow b$: $\quad e \mid \mathit{hmacsha1}\ k^m_{ab}\ e$ where $e = \mathit{aes}\ k^e_{ab}\ t$

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

$a \rightarrow b$: $\quad \mathit{aes}\ k^e_{ab}\ (t \mid \mathit{hmacsha1}\ k^m_{ab}\ t)$

MAC-and-Encrypt (as in SSH):

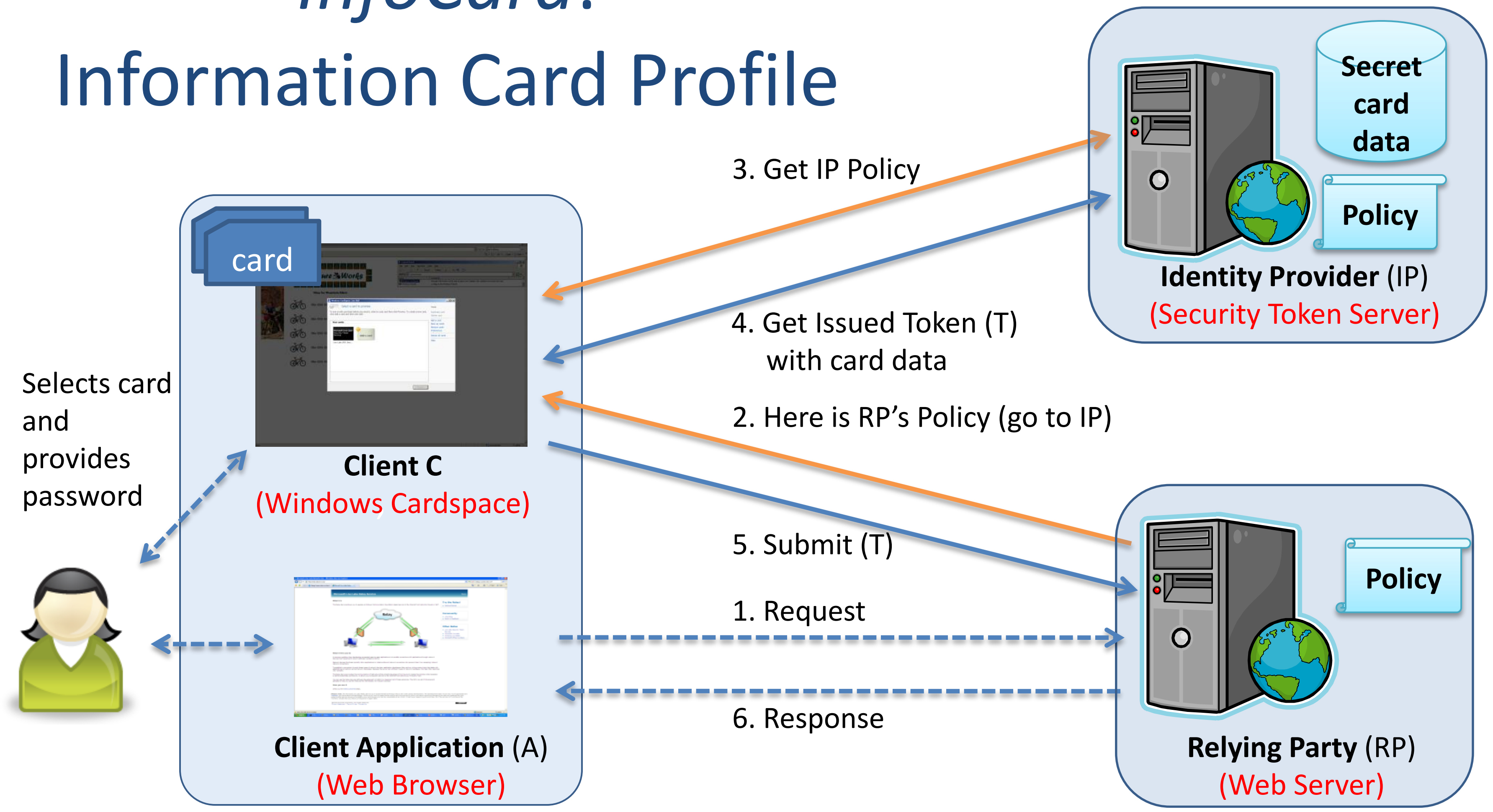
$a \rightarrow b$: $\quad \mathit{aes}\ k^e_{ab}\ t \mid \mathit{hmacsha1}\ k^m_{ab}\ t$

CASE STUDY

CARDSPACE

& WEB SERVICES SECURITY

InfoCard: Information Card Profile

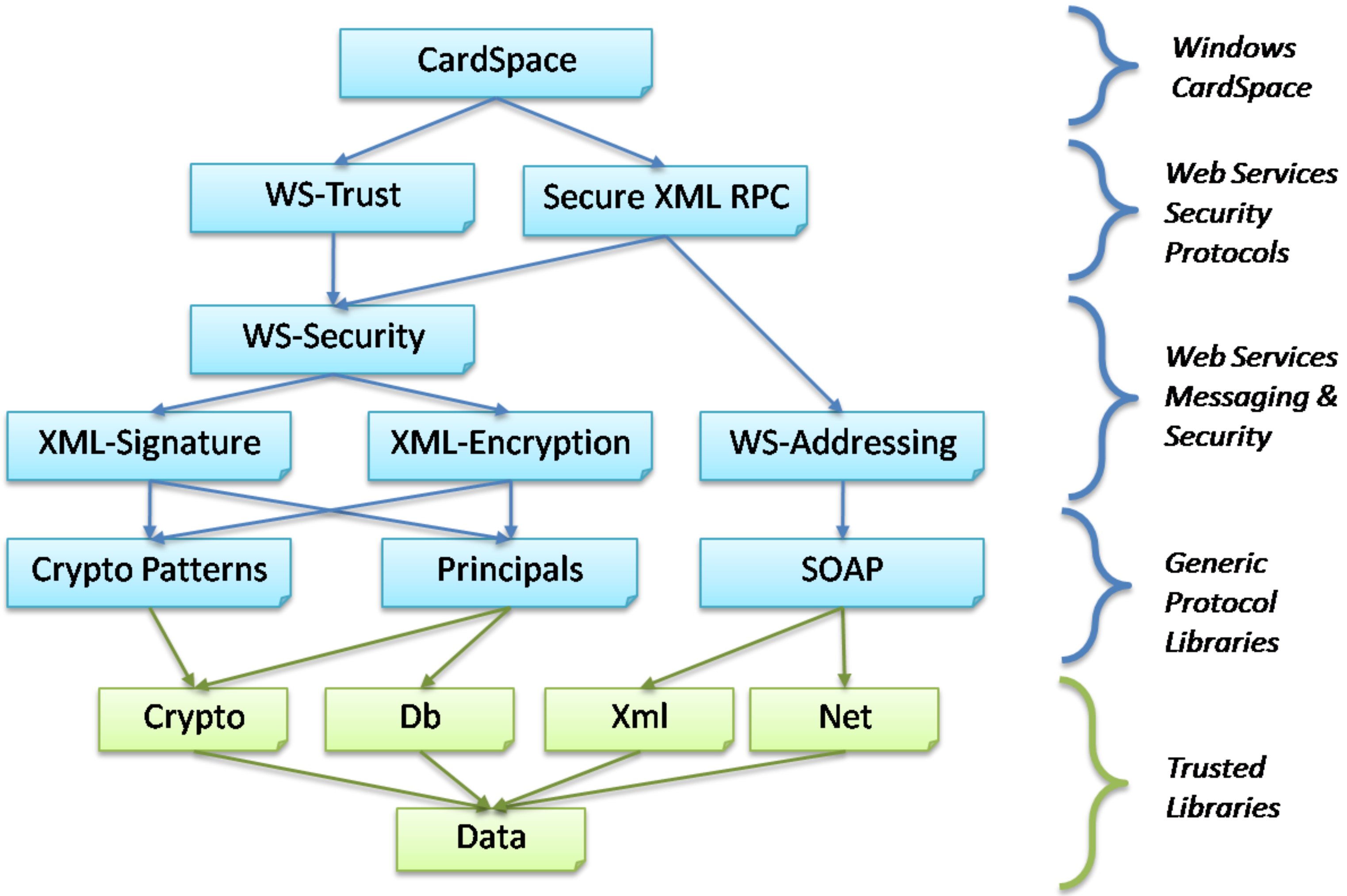


Protocol Narration (Managed Card)

Initially, **C has:** $cardId$, $\mathtt{PK}(k_{\mathrm{IP}})$, $\mathtt{PK}(k_{\mathrm{RP}})$; **IP has:** k_{IP}, $\mathtt{PK}(k_{\mathrm{RP}})$, $Card(cardId, claims_{\mathrm{U}}, pwd_{\mathrm{U,IP}}, k_{cardId})$; **RP has:** k_{RP}, $\mathtt{PK}(k_{\mathrm{IP}})$

C :	*Request* (RP, M_{req})	*C receives an application request*
U :	*Select InfoCard* (*cardId*, C, RP, $pwd_{\mathrm{U,IP}}$, $types_{\mathrm{RP}}$)	*User selects card and provides password*
C :	*generate fresh* $k_1, \eta_1, \eta_2, \eta_{ce}$	*Fresh session key, two nonces, and client entropy for token key*
C → IP :	let $M_{ek} = \mathtt{RSAEnc}(\mathtt{PK}(k_{\mathrm{IP}}), k_1)$ in	*Encrypt session key for* IP
	let $k_{sig} = \mathtt{PSHA1}(k_1, \eta_1)$ in	*Derive message signing key*
	let $k_{enc} = \mathtt{PSHA1}(k_1, \eta_2)$ in	*Derive message encryption key*
	let $M_{rst} = \mathtt{RST}(cardId, types_{\mathrm{RP}}, \mathrm{RP}, \eta_{ce})$ in	*Token request message body*
	let $M_{user} = (\mathrm{U}, pwd_{\mathrm{U}})$ in	*User authentication token*
	let $M_{mac} = \mathtt{HMACSHA1}(k_{sig}, (M_{rst}, M_{user}))$ in	*Message signature*
	Request Token $(M_{ek}, \eta_1, \eta_2,$ $\mathtt{AESEnc}(k_{enc}, M_{mac}), \mathtt{AESEnc}(k_{enc}, M_{user}),$ $\mathtt{AESEnc}(k_{enc}, M_{rst}))$	*Token Request, with encrypted signatures, token and body*
IP :	*Issue Token* (U, *cardId*, $claims_{\mathrm{U}}$, RP, *display*)	*IP issues token for U to use at RP*
IP :	*generate fresh* $\eta_3, \eta_4, \eta_{se}, k_t$	*Fresh nonces, server entropy, token encryption key*
IP → C :	let $k_{sig} = \mathtt{PSHA1}(k_1, \eta_3)$ in	*Derive message signing key*
	let $k_{enc} = \mathtt{PSHA1}(k_1, \eta_4)$ in	*Derive message encryption key*
	let $M_{tokkey} = \mathtt{RSAEnc}(\mathtt{PK}(k_{\mathrm{RP}}), \mathtt{PSHA1}(\eta_{ce}, \eta_{se}))$ in	*Compute token key from entropies, encrypt for* RP
	let $ppid_{cardId,\mathrm{RP}} = H_1(k_{cardId}, \mathrm{RP})$ in	*Compute PPID using card master key,* RP*'s identity*
	let $M_{tok} = \mathtt{Assertion}(\mathrm{IP}, M_{tokkey}, claims_{\mathrm{U}}, \mathrm{RP}, ppid_{cardId,\mathrm{RP}})$ in	*SAML assertion with token key, claims, and PPID*
	let $M_{toksig} = \mathtt{RSASHA1}(k_{\mathrm{IP}}, M_{tok})$ in	*SAML assertion signed by* IP
	let $M_{ek} = \mathtt{RSAEnc}(\mathtt{PK}(k_{\mathrm{RP}}), k_t)$ in	*Token encryption key, encrypted for* RP
	let $M_{enctok} = (M_{ek}, \mathtt{AESEnc}(k_t, \mathtt{SAML}(M_{tok}, M_{toksig})))$ in	*Encrypted issued token*
	let $M_{rstr} = \mathtt{RSTR}(M_{enctok}, \eta_{se})$ in	*Token response message body*
	let $M_{mac} = \mathtt{HMACSHA1}(k_{sig}, M_{rstr})$ in	*Message Signature*
	Token Response $(\eta_3, \eta_4, \mathtt{AESEnc}(k_{enc}, M_{mac}), \mathtt{AESEnc}(k_{enc}, M_{rstr}))$	*Token Response, with encrypted signature and body*
U :	*Approve Token* (*display*)	*User approves token*
C :	*generate fresh* $k_2, \eta_5, \eta_6, \eta_7$	*Fresh session key, three nonces*
C → RP :	let $M_{ek} = \mathtt{RSAEnc}(\mathtt{PK}(k_{\mathrm{RP}}), k_2)$ in	*Encrypt session key for* RP
	let $k_{sig} = \mathtt{PSHA1}(k_2, \eta_5)$ in	*Derive message signing key*
	let $k_{enc} = \mathtt{PSHA1}(k_2, \eta_6)$ in	*Derive message encryption key*
	let $k_{proof} = \mathtt{PSHA1}(\eta_{ce}, \eta_{se})$ in	*Compute token key from entropies*
	let $M_{mac} = \mathtt{HMACSHA1}(k_{sig}, M_{req})$ in	*Message signature*
	let $k_{endorse} = \mathtt{PSHA1}(k_{proof}, \eta_7)$ in	*Derive a signing key from the issued token key*
	let $M_{proof} = \mathtt{HMACSHA1}(k_{endorse}, M_{mac})$ in	*Endorsing signature proving possession of token key*
	Service Request $(M_{ek}, \eta_5, \eta_6, \eta_7, M_{enctok},$ $\mathtt{AESEnc}(k_{enc}, M_{mac}), \mathtt{AESEnc}(k_{enc}, M_{proof}),$ $\mathtt{AESEnc}(k_{enc}, M_{req}))$	*Service Request, with issued token, encrypted signatures and body*
RP :	*Accept Request* (IP, $claims_{\mathrm{U}}$, M_{req}, M_{resp})	RP *accepts request and authorizes a response*
RP :	*generate fresh* η_8, η_9	*Fresh nonces*
RP → C :	let $k_{sig} = \mathtt{PSHA1}(k_2, \eta_8)$ in	*Derive message signing key*
	let $k_{enc} = \mathtt{PSHA1}(k_2, \eta_9)$ in	*Derive message encryption key*
	let $M_{mac} = \mathtt{HMACSHA1}(k_{sig}, M_{resp})$ in	*Message signature*
	Service Response $(\eta_8, \eta_9,$ $\mathtt{AESEnc}(k_{enc}, M_{mac}), \mathtt{AESEnc}(k_{enc}, M_{resp}))$	*Service Response, with encrypted signatures and body*
C :	*Response* (M_{resp})	C *accepts response and sends it to application*

InfoCard: modular reference implementation

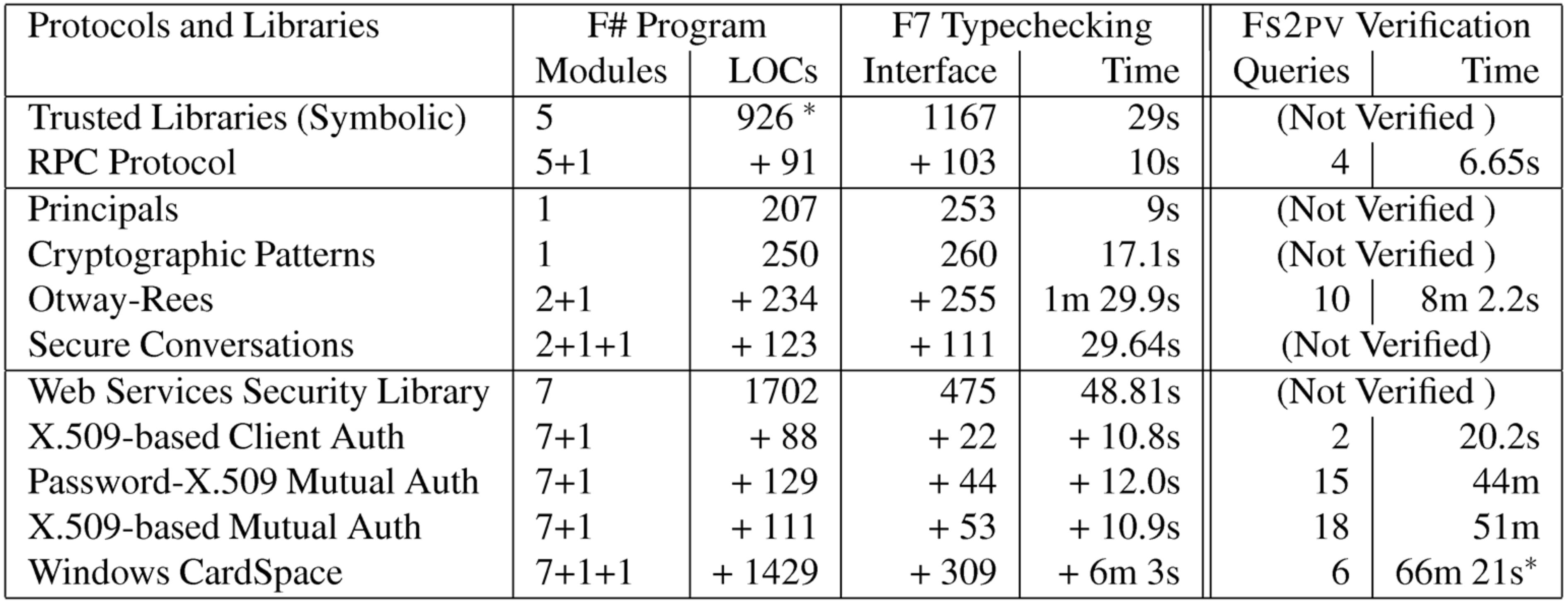


Verifying CardSpace

- We reviewed the protocol design
- We built a **modular reference implementation**
  - For the three CardSpace roles: client, relying party, identity provider
  - For the protocol stack: WS-Security standards & XML formats
  - For the underlying cryptographic primitives
- We first analyzed this code using PS2PV and ProVerif
- We now verify the same code by typing using **F7**
  - No change needed!
  - Fast, modular verification of F# code
  - We get stronger security properties,
    for a more precise model (reflecting all details of the XML format)

Evaluation

relative to FS2PV/ProVerif



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

- **Refinement typechecking is an effective, scalable verification technique for security protocols**

Tomorrow's lecture:

COMPUTATIONAL SOUNDNESS FOR TYPECHECKING ?

Symbolic vs Computational Cryptography

- Two verification approaches have been successfully applied to protocols and programs that use cryptography:

  Symbolic approach (Needham-Schroeder, Dolev-Yao, ... late 70's)
  - Structural view of protocols, using formal languages and methods
  - Compositional, automated verification tools, scales to large systems
  - Too abstract?

  Computational approach (Yao, Goldwasser, Micali, Rivest, ... early 80's)
  - More concrete, algorithmic view; more widely accepted
  - Adversaries range over probabilistic Turing machines
    Cryptographic materials range over bitstrings
  - Delicate (informal) game-based reduction proofs; poor scalability
- Can we get the best of both worlds? Much ongoing work on *computational soundness* for symbolic cryptography
- Can we verify real-world protocols?

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
  - Many "computational crypto" type systems already exist,
    sometimes easily adapted from "symbolic crypto" type systems

Computational soundness for F7

We rely on our existing F7 typechecker and code base

- Substantial implementation effort
- Flexible support for high-level security properties: authentication, authorization, secrecy
- Case studies: many protocol implementations, a few large ones
- Good basis for comparison with other F# tool chains:
  - fs2pv/ProVerif,
  - fs2cv/Cryptoverif

Computational soundness for F7

We rely on our existing F7 typechecker and code base

1. We typecheck protocols and applications against *refined typed interfaces* for cryptography (automatically)

2. We relate several implementations of our interface (once for all)
   - A *symbolic, well-typed implementation* (much as before)
   - A *concrete implementation* (not typable in F7)
   - Intermediate implementations, to show computational soundness by applying "code-based game-rewriting" onto F# code

We obtain computational soundness
both for *robust safety* and for *strong secrecy*
(for ptime protocols, applications, and adversaries)

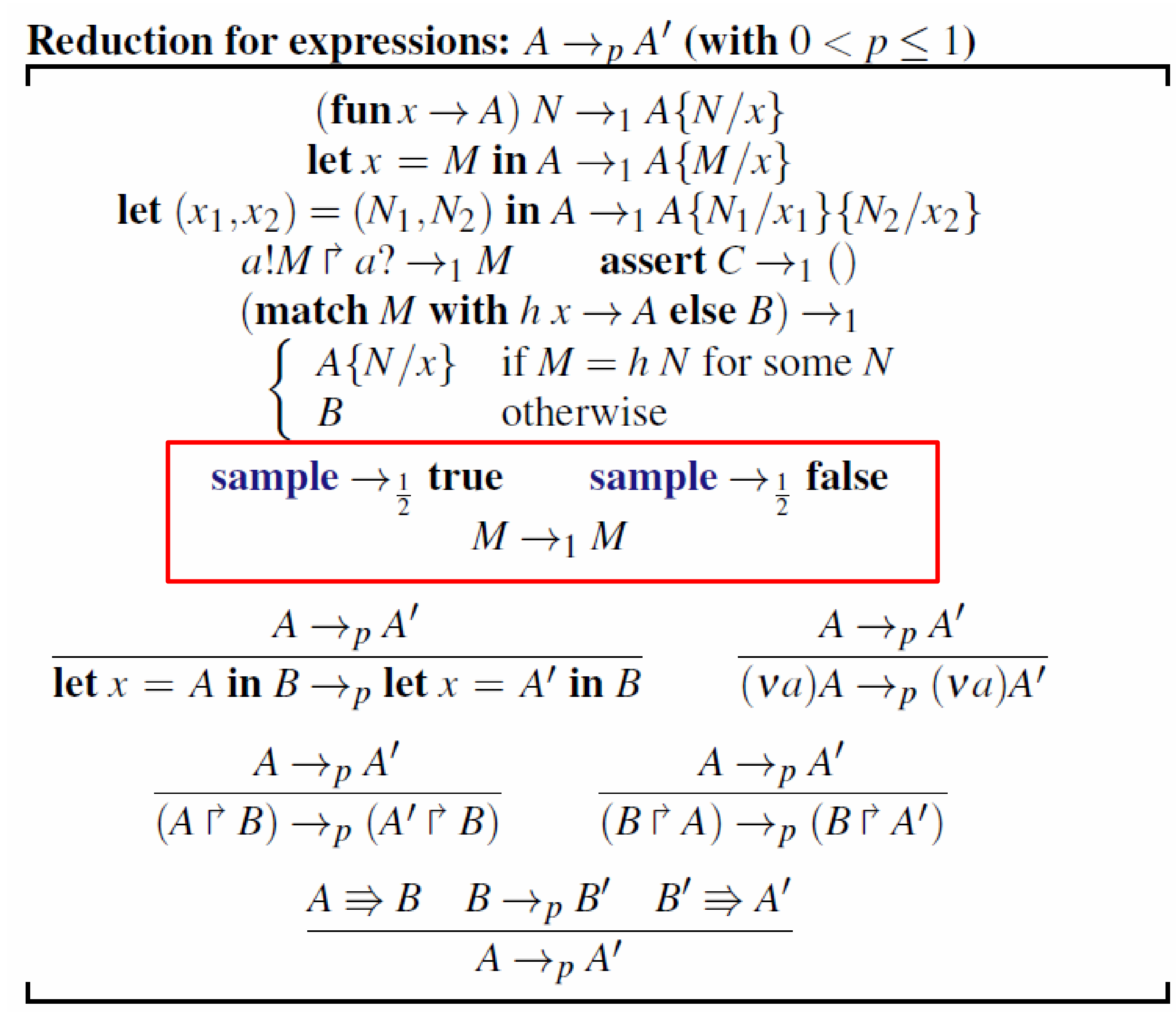
Probabilistic RCF

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

$$A \rightarrow_p A'$$

  - 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** $0 < p \leq 1$**)**

$$(\mathbf{fun}\, x \rightarrow A)\ N \rightarrow_1 A\{N/x\}$$

$$\mathbf{let}\ x = M\ \mathbf{in}\ A \rightarrow_1 A\{M/x\}$$

$$\mathbf{let}\ (x_1,x_2) = (N_1,N_2)\ \mathbf{in}\ A \rightarrow_1 A\{N_1/x_1\}\{N_2/x_2\}$$

$$a!M \Rsh a? \rightarrow_1 M \qquad \mathbf{assert}\ C \rightarrow_1 ()$$

$$(\mathbf{match}\ M\ \mathbf{with}\ h\ x \rightarrow A\ \mathbf{else}\ B) \rightarrow_1 \begin{cases} A\{N/x\} & \text{if } M = h\ N \text{ for some } N \\ B & \text{otherwise} \end{cases}$$

$$\boxed{\begin{array}{c} \mathbf{sample} \rightarrow_{\frac{1}{2}} \mathbf{true} \qquad \mathbf{sample} \rightarrow_{\frac{1}{2}} \mathbf{false} \\ M \rightarrow_1 M \end{array}}$$

$$\frac{A \rightarrow_p A'}{\mathbf{let}\ x = A\ \mathbf{in}\ B \rightarrow_p \mathbf{let}\ x = A'\ \mathbf{in}\ B} \qquad \frac{A \rightarrow_p A'}{(\nu a)A \rightarrow_p (\nu a)A'}$$

$$\frac{A \rightarrow_p A'}{(A \Rsh B) \rightarrow_p (A' \Rsh B)} \qquad \frac{A \rightarrow_p A'}{(B \Rsh A) \rightarrow_p (B \Rsh A')}$$

$$\frac{A \Rrightarrow B \quad B \rightarrow_p B' \quad B' \Rrightarrow A'}{A \rightarrow_p A'}$$

Probabilistic RCF

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

let x_0 = **sample in** … **let** x_{n-1} = **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.

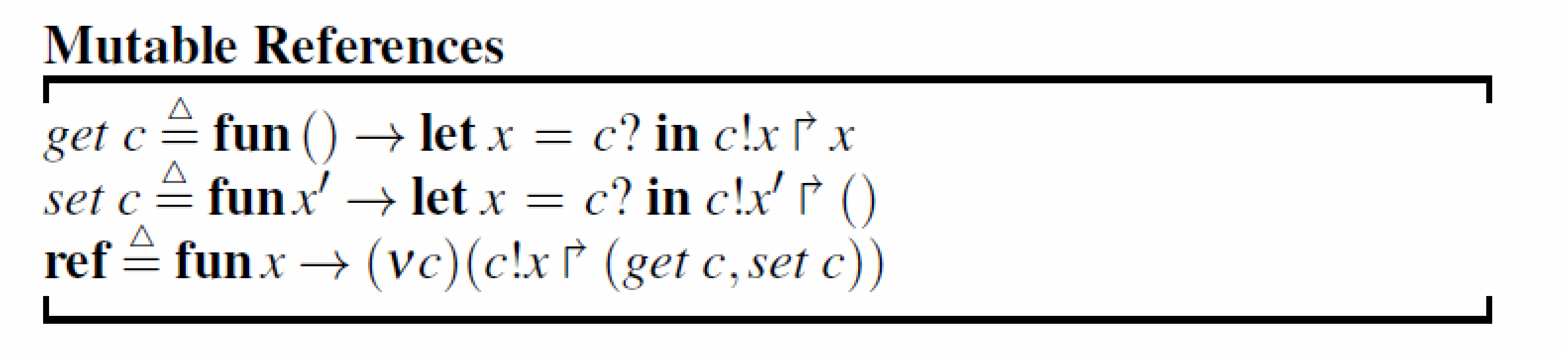
  - We add a typing rule for sampling

$$\frac{E \vdash \diamond}{E \vdash \textbf{sample} : \text{bool}}$$

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

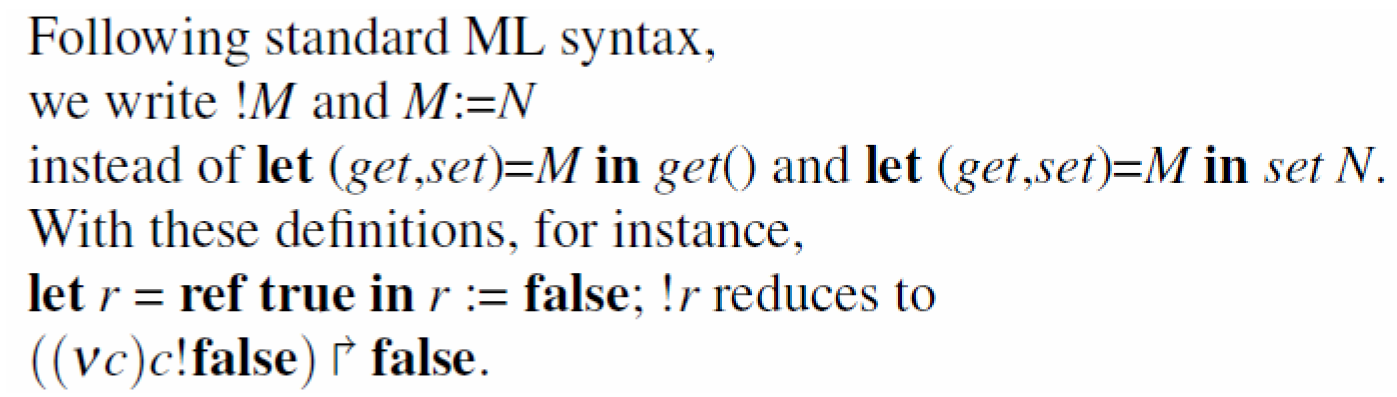
Probabilistic RCF

- We rule out internal non-determinism (to match crypto assumptions)
  - We exclude race conditions on communications
  - We still use private channels for encoding mutable references and public channels for networking and adversarial control



Mutable References

$$\mathit{get}\ c \triangleq \mathbf{fun}\ () \to \mathbf{let}\ x = c?\ \mathbf{in}\ c!x \Rsh x$$
$$\mathit{set}\ c \triangleq \mathbf{fun}\ x' \to \mathbf{let}\ x = c?\ \mathbf{in}\ c!x' \Rsh ()$$
$$\mathbf{ref} \triangleq \mathbf{fun}\ x \to (\nu c)(c!x \Rsh (\mathit{get}\ c, \mathit{set}\ c))$$



Following standard ML syntax,
we write $!M$ and $M{:=}N$
instead of **let** $(\mathit{get},\mathit{set}){=}M$ **in** $\mathit{get}()$ and **let** $(\mathit{get},\mathit{set}){=}M$ **in** $\mathit{set}\ N$.
With these definitions, for instance,
let r = **ref true in** r := **false**; $!r$ reduces to
$((\nu c)c!\mathbf{false}) \Rsh \mathbf{false}$.

Probabilistic RCF

- We equip RCF with a probabilistic semantics (Markov chains)
- We rule out internal non-determinism (to match crypto assumptions)
- We cut down and adapt our cryptographic libraries

Sample computational soundness for keyed hash functions

HMAC & INT-CMA

Sample computational soundness:

Keyed cryptographic hashes

```
module Hmac
type key
type bytes = string
type text = bytes
type mac = bytes

val GEN: unit → key
val MAC: key → text → mac
val VERIFY: key → text → mac → bool
```

plain F#
interface

```
open System.Security.Cryptography

let rng = new RNGCryptoServiceProvider()
let randomBytes n =
   let b = Bytearray.make n in rng.GetBytes b; Key b

let GEN () = randomBytes 32 (* 256 bits *)
let MAC (Key k) (t:text) =
   base64 ((new HMACSHA1(k)).ComputeHash (utf8 t))
let VERIFY k t sv = (MAC k t = sv)
```

concrete F#
implementation
(calling .NET)

Sample computational soundness:

Keyed cryptographic hashes

```
module Hmac
type key
type bytes = string
type text = bytes
type mac = bytes
type authentic = Msg of text

val GEN: unit → key
val MAC: k:key → t:text{Msg(t)} → mac
val VERIFY: k:key → t:text → m:mac → b:bool{b=true ⇒ Msg(t)}
```

"All verified messages are authentic"

"ideal" F7 interface

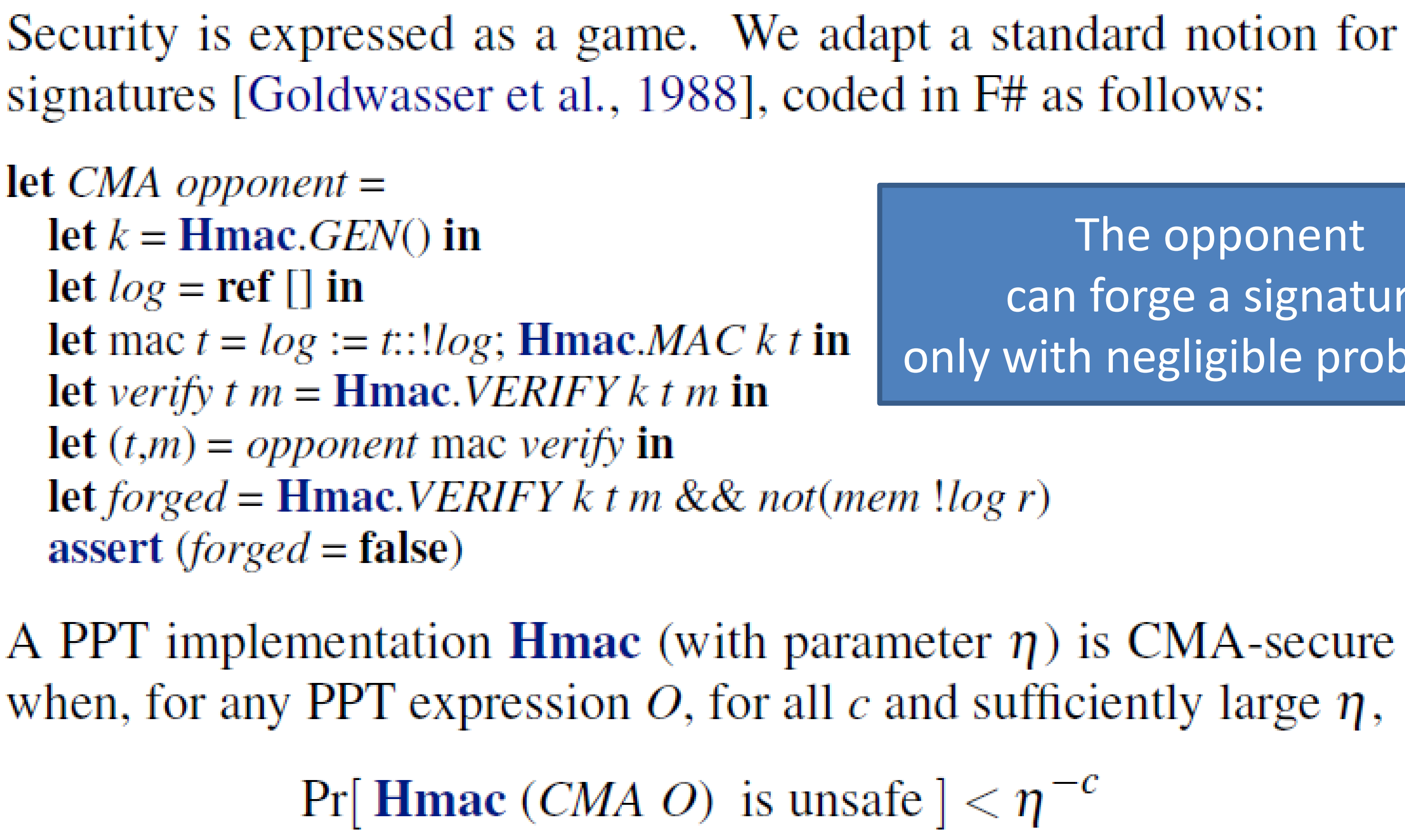
```
open System.Security.Cryptography

let rng = new RNGCryptoServiceProvider()
let randomBytes n =
  let b = Bytearray.make n in rng.GetBytes b; Key b

let GEN () = randomBytes 32 (* 256 bits *)
let MAC (Key k) (t:text) =
  base64 ((new HMACSHA1(k)).ComputeHash (utf8 t))
let VERIFY k t sv = (MAC k t = sv)
```

concrete F# implementation (calling .NET)

Can't be true (many collisions)

Cryptographic assumption: resistance against

Adaptive Chosen-Message existential forgery Attacks



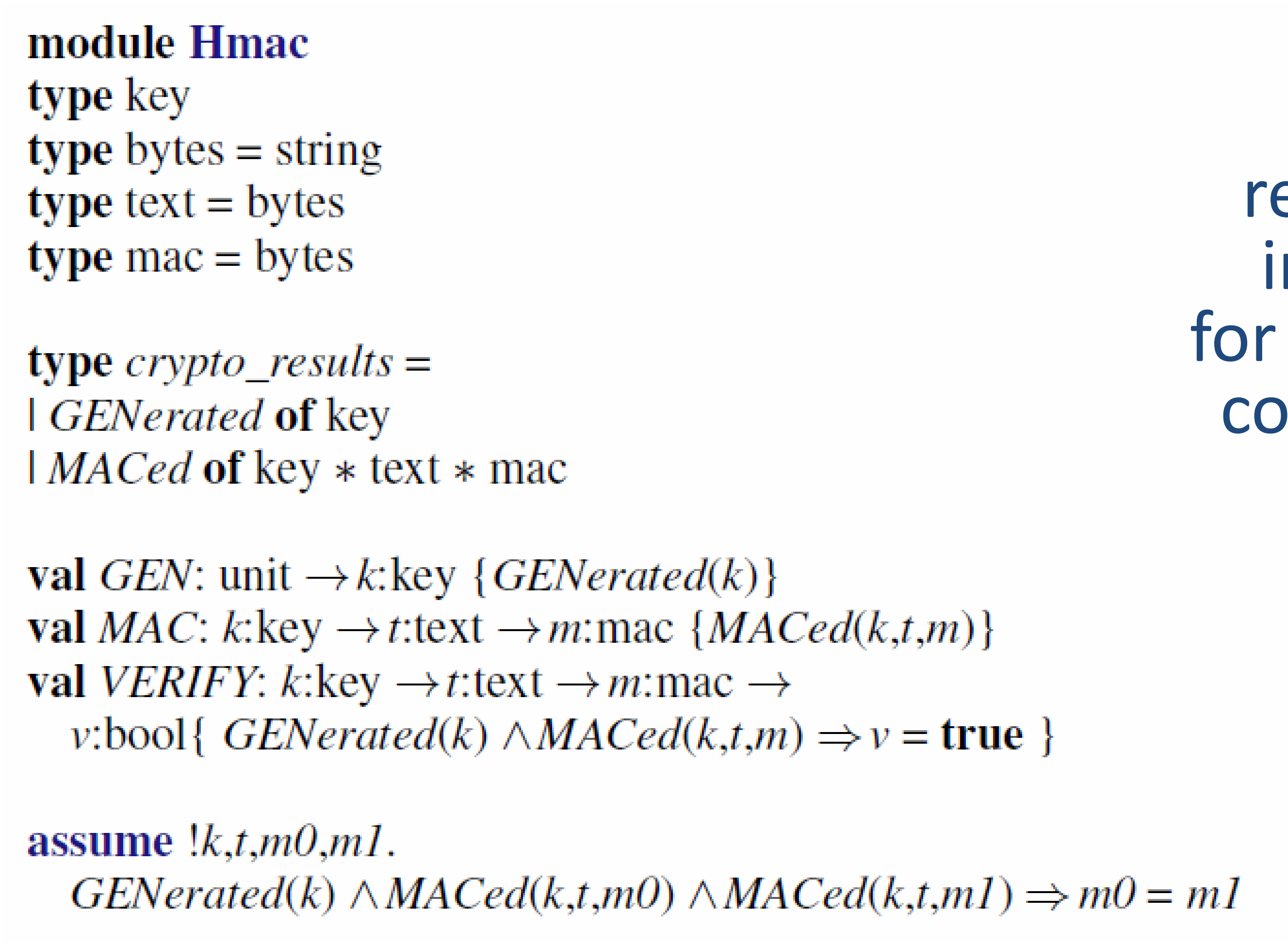
Security is expressed as a game. We adapt a standard notion for signatures [Goldwasser et al., 1988], coded in F# as follows:

```
let CMA opponent =
  let k = Hmac.GEN() in
  let log = ref [] in
  let mac t = log := t::!log; Hmac.MAC k t in
  let verify t m = Hmac.VERIFY k t m in
  let (t,m) = opponent mac verify in
  let forged = Hmac.VERIFY k t m && not(mem !log r)
  assert (forged = false)
```

The opponent
can forge a signature
only with negligible probability

A PPT implementation **Hmac** (with parameter η) is CMA-secure when, for any PPT expression O, for all c and sufficiently large η,

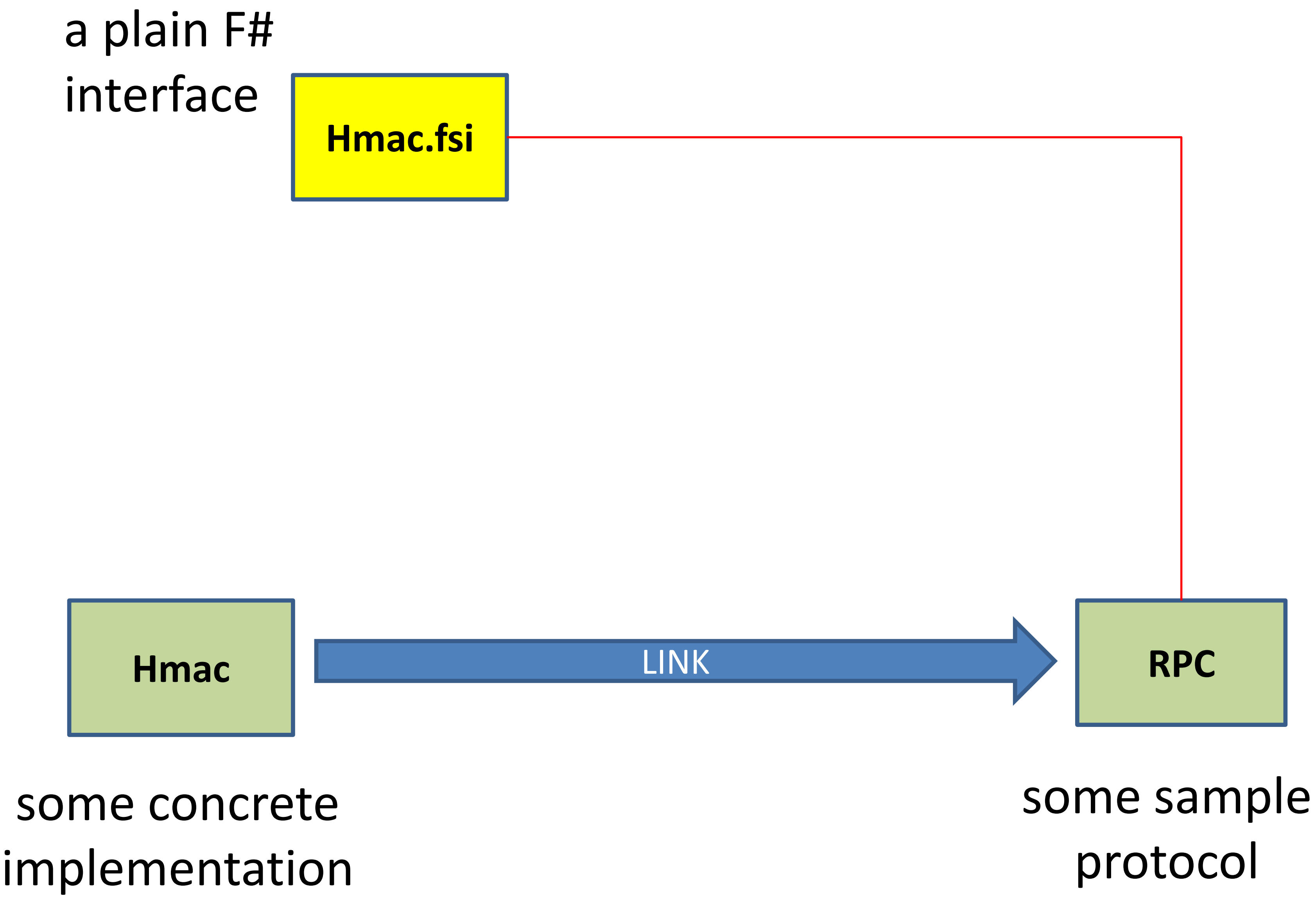
$$\Pr[\,\mathbf{Hmac}\ (CMA\ O)\ \text{ is unsafe}\,] < \eta^{-c}$$



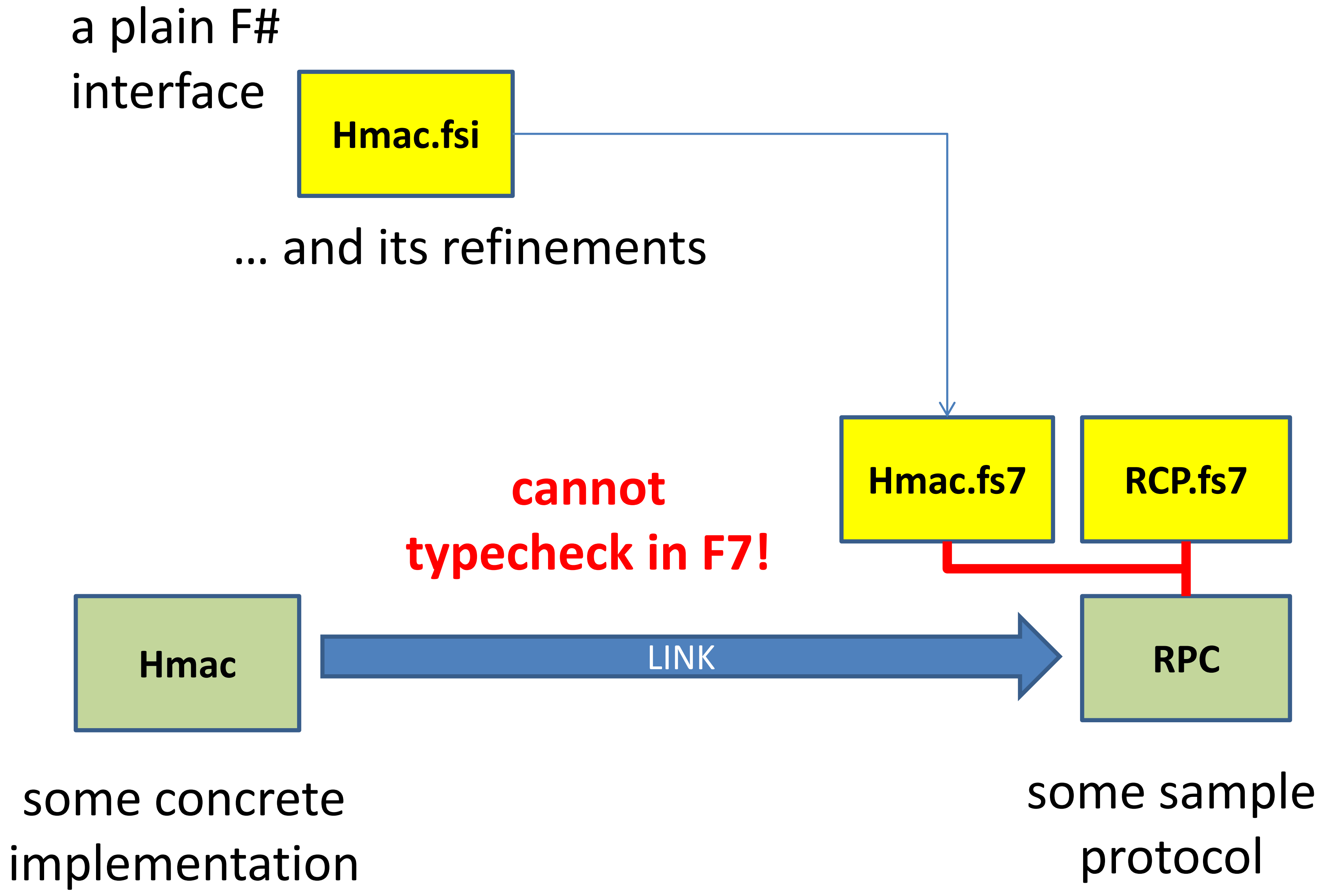
```
module Hmac
type key
type bytes = string
type text = bytes
type mac = bytes

type crypto_results =
| GENerated of key
| MACed of key ∗ text ∗ mac

val GEN: unit → k:key {GENerated(k)}
val MAC: k:key → t:text → m:mac {MACed(k,t,m)}
val VERIFY: k:key → t:text → m:mac →
   v:bool{ GENerated(k) ∧MACed(k,t,m) ⇒ v = true }

assume !k,t,m0,m1.
   GENerated(k) ∧MACed(k,t,m0) ∧MACed(k,t,m1) ⇒ m0 = m1
```

refined F7 interface for functional correctness

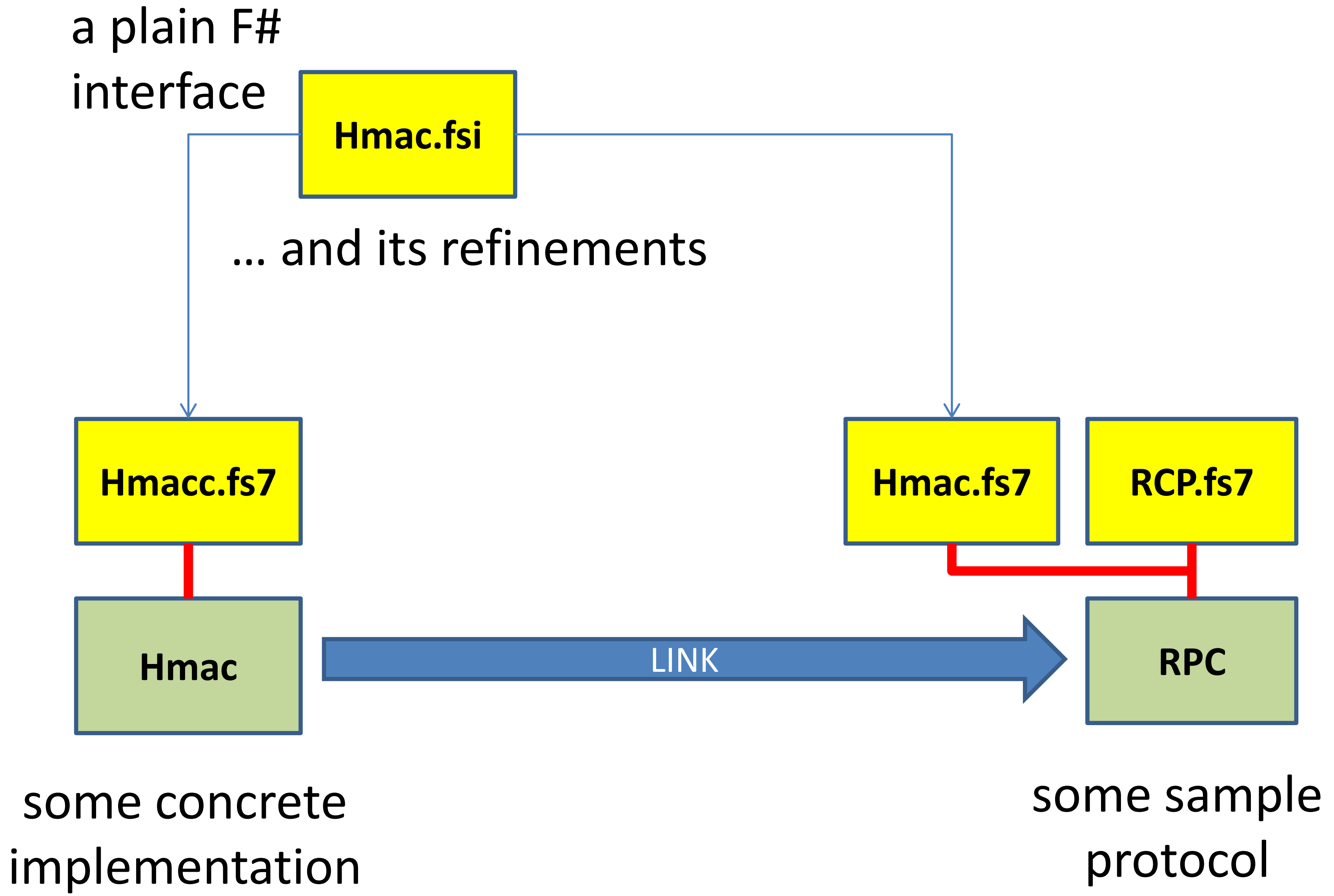
MACs: interfaces and implementations



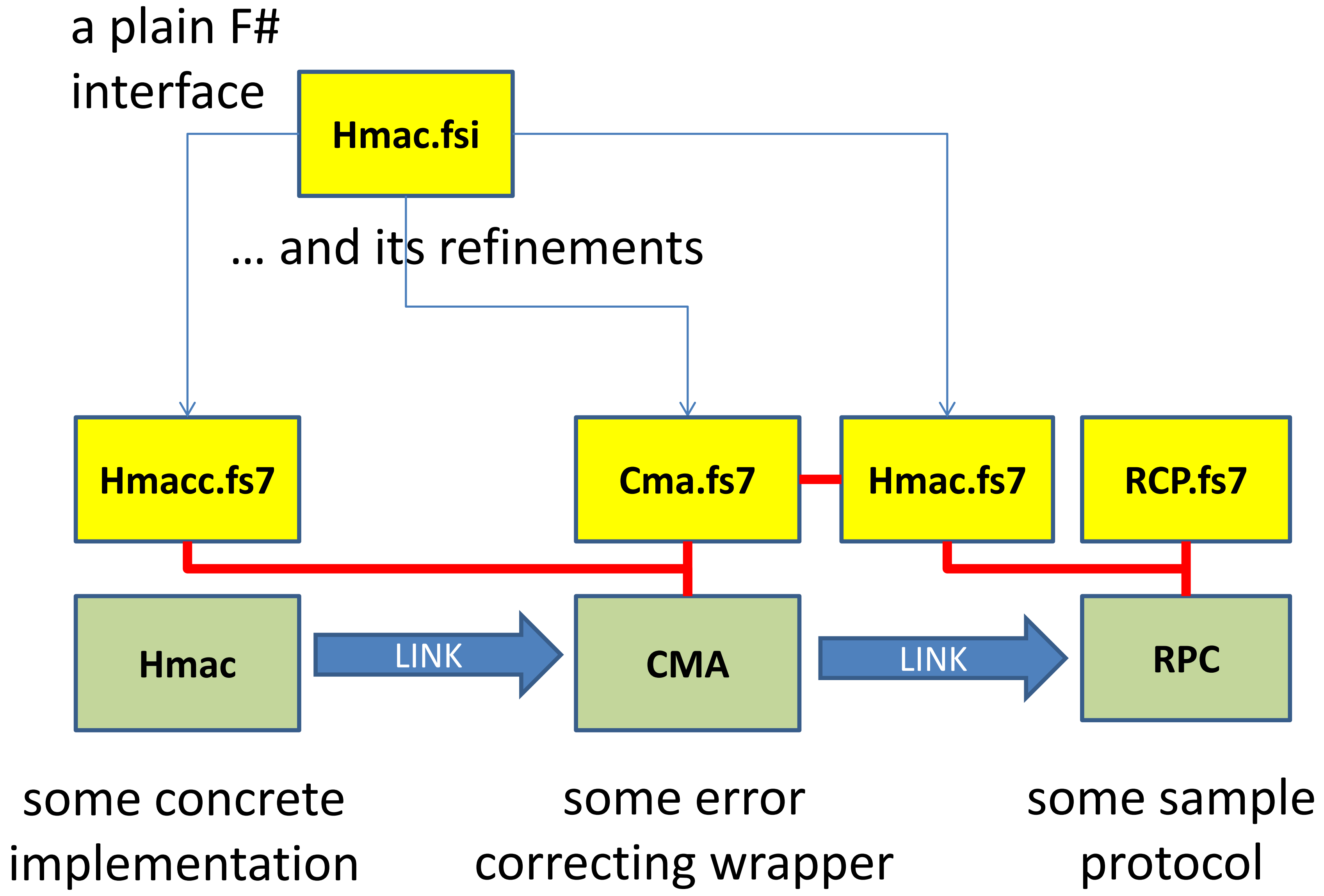
MACs: interfaces and implementations



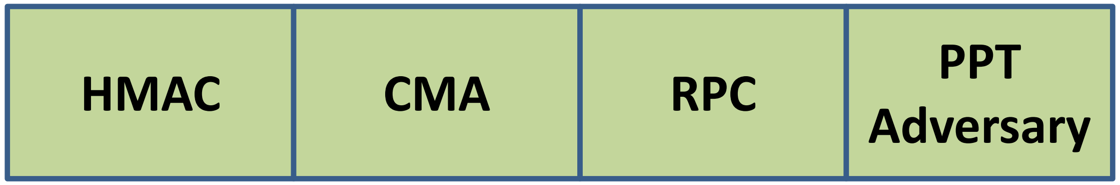
MACs: interfaces and implementations



MACs: interfaces and implementations

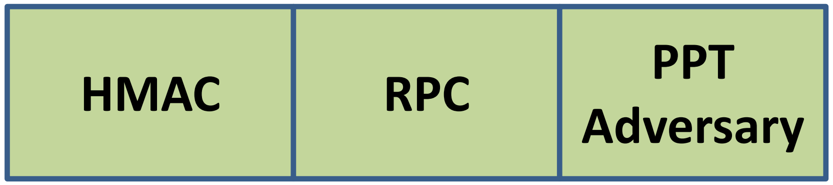


MACs: interfaces and implementations

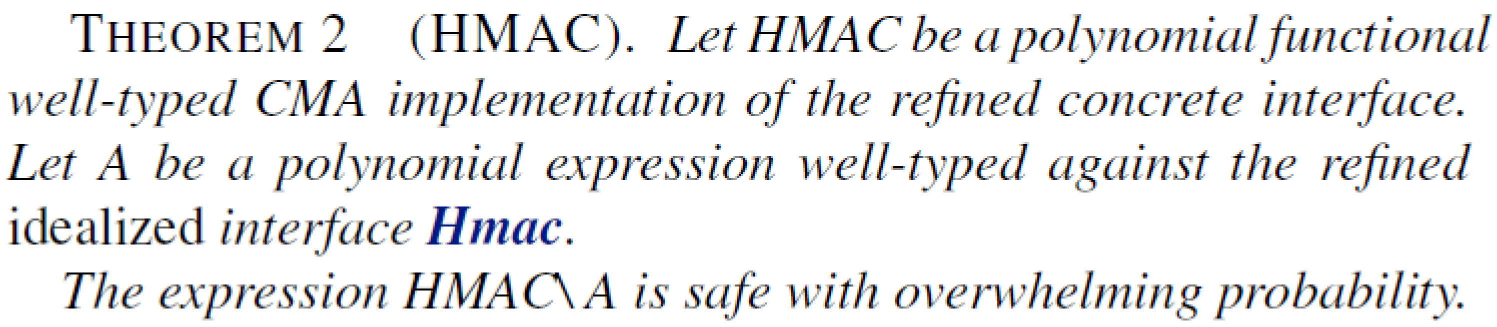


is always safe (by typing)

is indistinguishable from



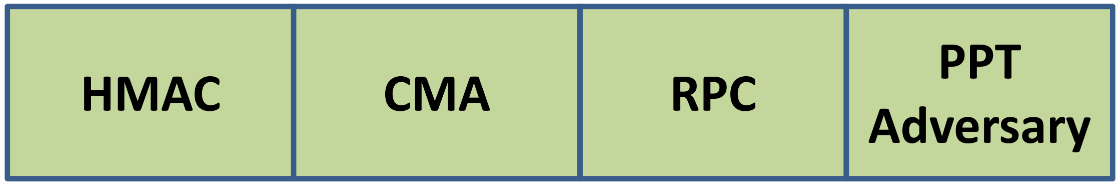
is safe too, with overwhelming probability



THEOREM 2 (HMAC). *Let HMAC be a polynomial functional well-typed CMA implementation of the refined concrete interface. Let A be a polynomial expression well-typed against the refined* idealized *interface* ***Hmac***.

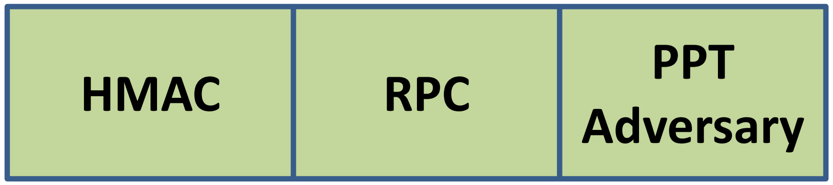
The expression HMAC\A is safe with overwhelming probability.

MACs: interfaces and implementations

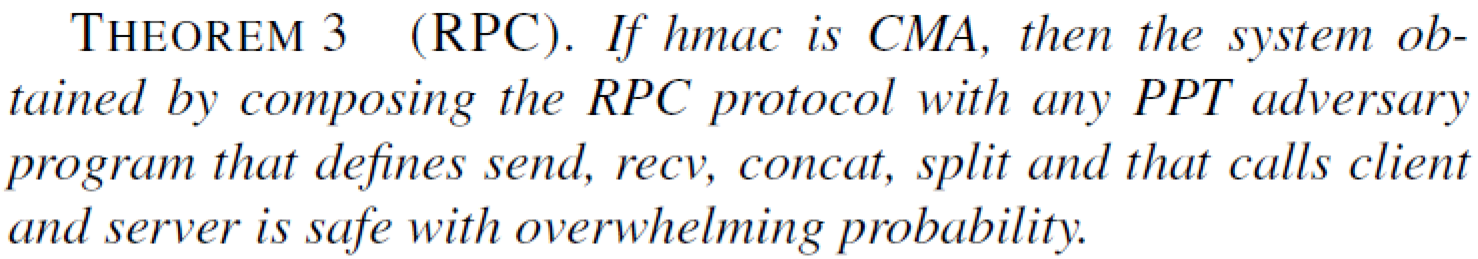


is always safe (by typing)

is indistinguishable from



is safe too, with overwhelming probability



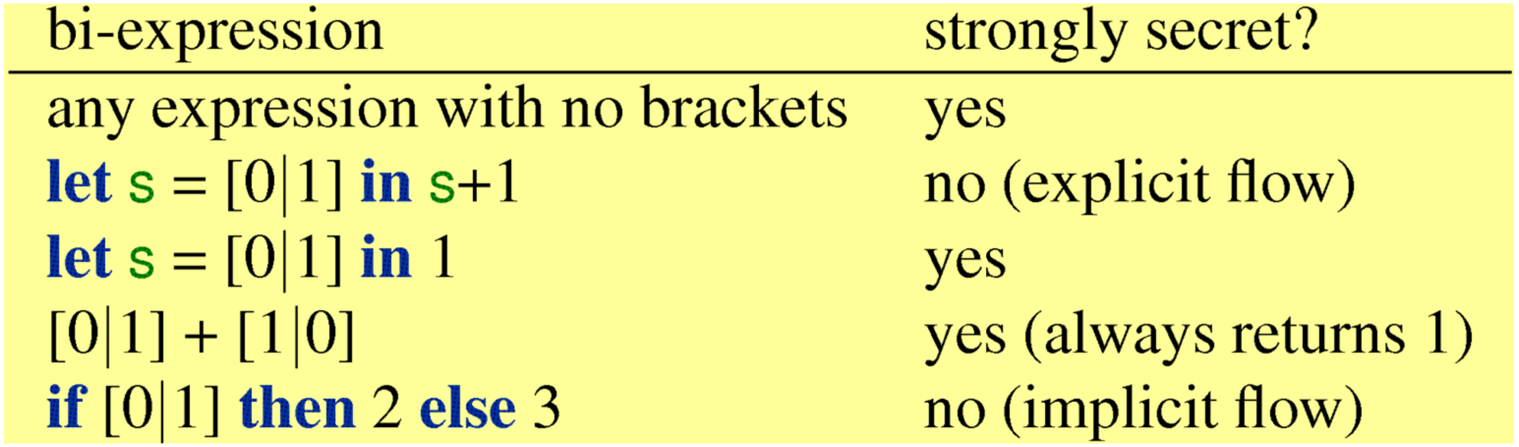
THEOREM 3 (RPC). *If hmac is CMA, then the system obtained by composing the RPC protocol with any PPT adversary program that defines send, recv, concat, split and that calls client and server is safe with overwhelming probability.*

Computational Soundness
for Strong Secrecy

ENCRYPTION & CCA2

Strong secrecy (Indistinguishability)

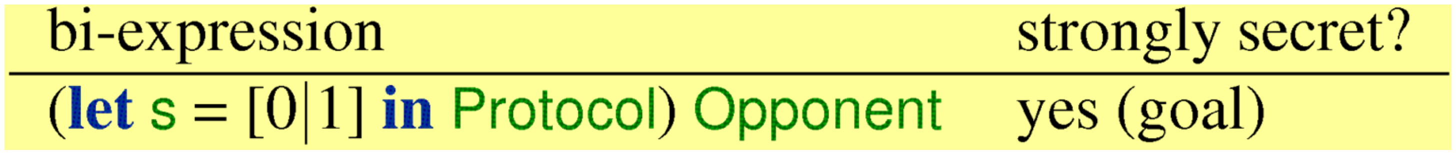
- Secrecy is expressed as **observational equivalence** between two variants of a program that differ only on **selected sub-expressions**
  - We use brackets **[A_0 | A_1]** to write both variants as a single program (as proposed by Pottier, and used by ProVerif for proving equivalences)
  - **Can we observe the contents of brackets?**
    We obtain two programs by selecting on the left vs selecting on the right
    We run both programs and compare the results



bi-expression	strongly secret?
any expression with no brackets	yes
let s = $[0 \mid 1]$ **in** s+1	no (explicit flow)
let s = $[0 \mid 1]$ **in** 1	yes
$[0 \mid 1] + [1 \mid 0]$	yes (always returns 1)
if $[0 \mid 1]$ **then** 2 **else** 3	no (implicit flow)

Strong secrecy (Indistinguishability)

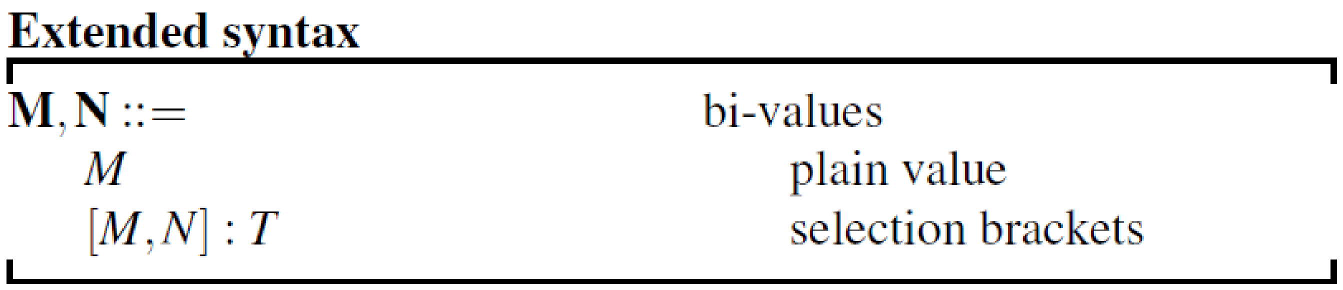
- Secrecy is expressed as **observational equivalence** between two variants of a program that differ only on **selected sub-expressions**
  - We use brackets **[A_0 | A_1]** to write both variants as a single program (as proposed by Pottier, and used by ProVerif for proving equivalences)
  - **Can we observe the contents of brackets?**
    We obtain two programs by selecting on the left vs selecting on the right
    We run both programs and compare the results



bi-expression	strongly secret?
(**let** s = [0|1] **in** Protocol) Opponent	yes (goal)

Strong secrecy (Indistinguishability)

- Secrecy is expressed as **observational equivalence** between two variants of a program that differ only on **selected sub-expressions**
  - We use brackets **[A_0 | A_1]** to write both variants as a single program (as proposed by Pottier, and used by ProVerif for proving equivalences)



Extended syntax

$\mathbf{M}, \mathbf{N} ::=$	bi-values
M	plain value
$[M, N] : T$	selection brackets

  - In F# programs, we use instead **select A_0 A_1** where select is globally bound to either (fun x_0 x_1 → x_0) or (fun x_0 x_1 → x_1)

Strong secrecy (Indistinguishability)

- Secrecy is expressed as **observational equivalence** between two variants of a program that differ only on **selected sub-expressions**
  - We use brackets **[A_0 | A_1]** to write both variants as a single program (as proposed by Pottier, and used by ProVerif for proving equivalences)

- The "bi-expression" **A** *preserves strong secrecy* when $\mathbf{A_0}\,\mathcal{O} \approx \mathbf{A_1}\,\mathcal{O}$ for all opponent expressions *O* (with no secret sub-expressions)

- Computationally, the bi-expression **A** *preserves secrecy* when, for all ptime opponent expressions *O* (with no secret sub-expressions),

$$|Pr[\mathbf{A}^0\ O \text{ returns } 0] - Pr[\mathbf{A}^1\ O \text{ returns } 0]| \leq \varepsilon(\eta)$$

Strong secrecy by typing

- Secrecy is expressed as **observational equivalence** between two variants of a program that differ only on **selected sub-expressions**

- Secrecy is proved by typing (relying on parametricity) with a new typing rule

$$\frac{E \vdash M : U \quad E \vdash N : U \quad E \vdash T}{E \vdash ([M,N] : T) : T}$$

This rule, for instance, enables us to type

$$\alpha \vdash \mathbf{fun}\, x\ y \rightarrow ([x,y] : \alpha) : U \rightarrow U \rightarrow \alpha$$

Public-Key Encryption (using RSA-OAEP)

```
module RSA
type bytes = byte[]
type plain = bytes
type cipher = bytes
type pkey
type skey
val GEN: unit → pkey ∗ skey
val ENC: pkey → plain → cipher
val DEC: skey → cipher → plain
```

```
open System.Security.Cryptography
type pkey = PubKey of RSAParameters
type skey = SecKey of RSAParameters

let GEN () =
    let RSA = new RSACryptoServiceProvider () in
    let pubkey = RSA.ExportParameters(false) in
    let seckey = RSA.ExportParameters(true ) in
    (PubKey pubkey, SecKey seckey)

let ENC (PubKey pubkey) plaintext =
    let RSA = new RSACryptoServiceProvider () in
    RSA.ImportParameters(pubkey);
    RSA.Encrypt (plaintext,true)

let DEC (SecKey seckey) ciphertext =
    let RSA = new RSACryptoServiceProvider () in
    RSA.ImportParameters(seckey);
    RSA.Decrypt (ciphertext,true)
```

Public-Key Encryption (more abstractly)

```
module RSA
type bytes = byte[]
type plain = bytes
type cipher = bytes
type pkey
type skey
val GEN: unit → pkey ∗ skey
val ENC: pkey → plain → cipher
val DEC: skey → cipher → plain
```

```
type bytes = RSA.bytes
type cipher = RSA.cipher
type plain = α
type pkey
type skey
val GEN: unit → pkey ∗ skey
val ENC: pkey → plain → cipher
val DEC: skey → cipher → plain
```

IND-CCA2

Security (IND-CC2) The security assumption is expressed as an indistinguishability game.

```
let cca2game o =
  let ke, kd = RSA.GEN () in
  let log = ref (empty ke) in
  let e x0 x1 =
    let x = select x0 x1 in
    let v = RSA.ENC ke x in
    log := cons ke (v,x) !log;
    v in
  let d v =
    match assoc ke v !log with
    | Some(x) → x
    | None → RSA.DEC kd v in
  o e d ke
```

An encryption scheme *PKE* is IND-CCA2 secure when, for any closed polynomial expression *O*, The bi-expression *PKE* (*cca2game O*) is indistinguishable.

Computational secrecy by typing

$$\begin{aligned} GED \triangleq\ & \alpha, pkey, skey, GEN : \mathrm{unit} \to pkey * skey, \\ & ENC : pkey \to \alpha \to \mathrm{bytes}, DEC : skey \to \mathrm{bytes} \to \alpha \end{aligned}$$

THEOREM 5 (CCA2). Let PKE be a polynomial functional well-typed IND-CCA2 implementation and $\mathbf{A}$ a polynomial bi-expression such that $GED \vdash \mathbf{A} : \mathrm{bool}$. Then $PKE\ \mathbf{A}$ is indistinguishable:

$$|Pr[PKE\ \mathbf{A}^0 = 0] - Pr[PKE\ \mathbf{A}^1 = 0]| \leq \varepsilon(\eta)$$

VERIFYING PROTOCOL IMPLEMENTATIONS (SUMMARY)

Summary (Computational Soundness)

- We obtain computational soundness for general classes of protocol implementations coded in F# and well-typed in F7
  - Our results apply to large protocol implementations (>5 kLOC)
  - We relate different implementations of the same interfaces
- We use refinement types to control the usage of cryptography
  - We need a typable ideal functionality—we don't care whether it is a traditional Dolev-Yao model!
  - The proofs are elementary and/or largely automated
- How flexible/general is it?
  - Can also model malleability, key derivation, key compromise, …
- Typing vs CryptoVerif [Blanchet] ?
  - Compositionality, scalability, complementarity (via fs2cv)

Questions?

- We obtain computational soundness for general classes of protocol implementations coded in F# and well-typed in F7
  - Our results apply to large protocol implementations (>5 kLOC)
  - We relate different implementations of the same interfaces
- We use refinement types to control the usage of cryptography
  - We need a typable ideal functionality—we don't care whether it is a traditional Dolev-Yao model!
  - The proofs are elementary and/or largely automated
- How flexible/general is it?
  - Can also model malleability, key derivation, key compromise, …
- Typing vs CryptoVerif [Blanchet] ?
  - Compositionality, scalability, complementarity (via fs2cv)