Modèles et Outils pour la Vérification Cryptographique

Cédric Fournet

Microsoft Research Cambridge & **MSR-INRIA Orsay**

Modèles et Outils pour la Vérification Cryptographique

1. Vérifier les protocoles & programmes cryptographiques
2. F7: Un outil de vérification pour F# + types dépendents
3. Vérification symbolique d’un petit protocole
4. Application: CardSpace
5. Vérification calculatoire
6. Demo: Distributed Key Manager

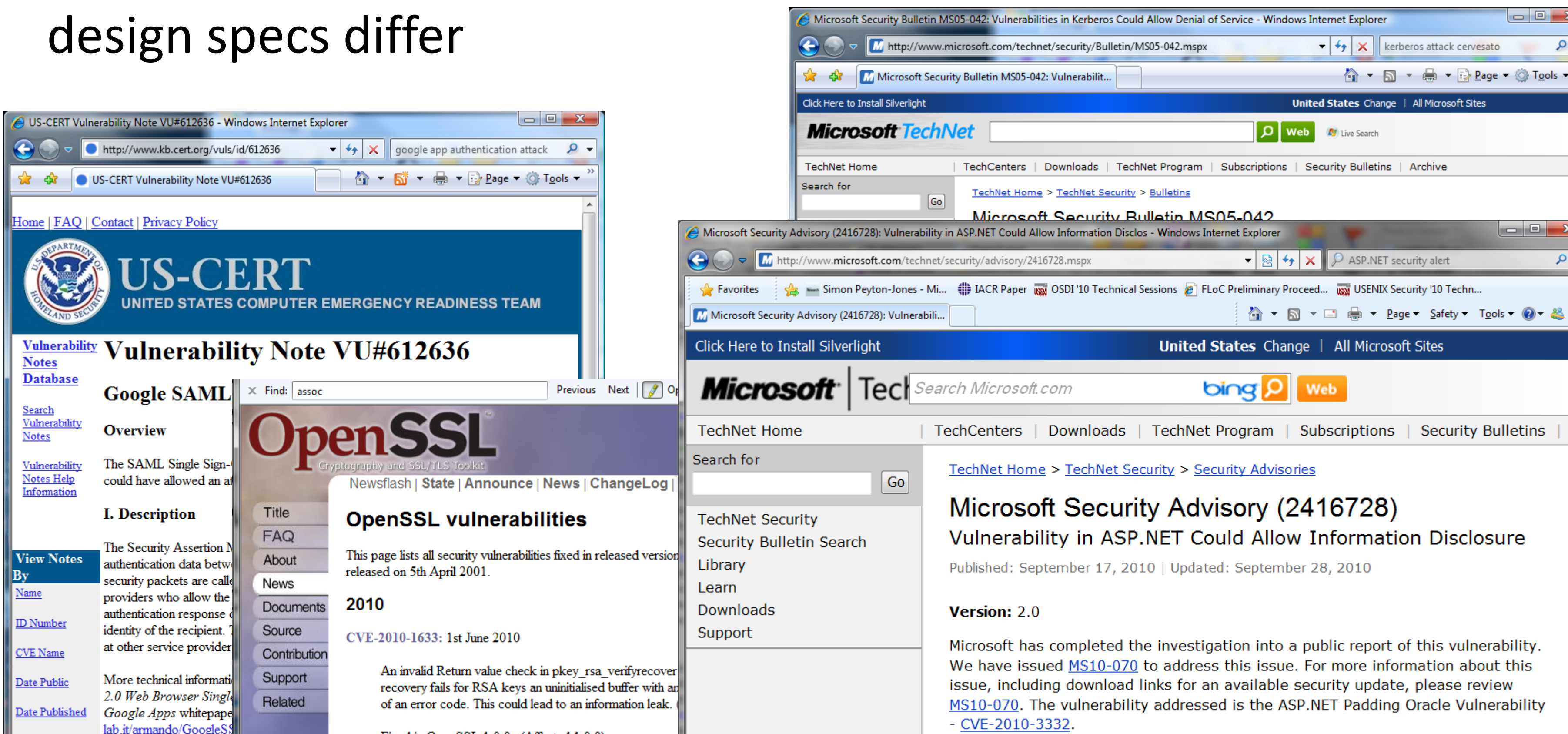
http://research.microsoft.com/~fournet

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

Verifying Protocol Implementations

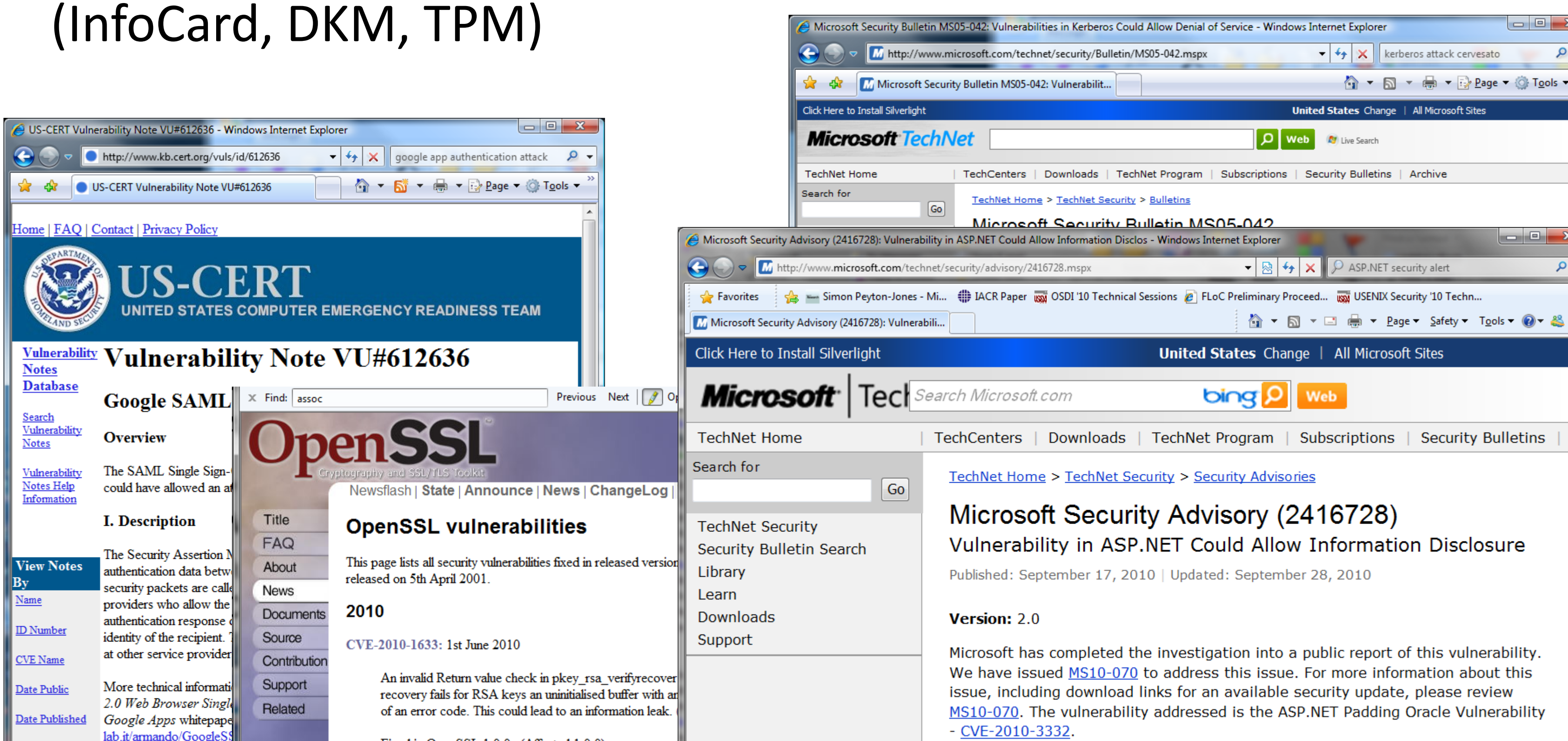
Cryptographic protocols (still) go wrong

- Design & implementation errors often lead to serious security vulnerabilities: SAML, OpenSSL, ASP.NET
- Traditional crypto models miss most details
- Production code and design specs differ



Goal: Verify production code relying on Cryptography

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



Symbolic vs Computational Cryptography

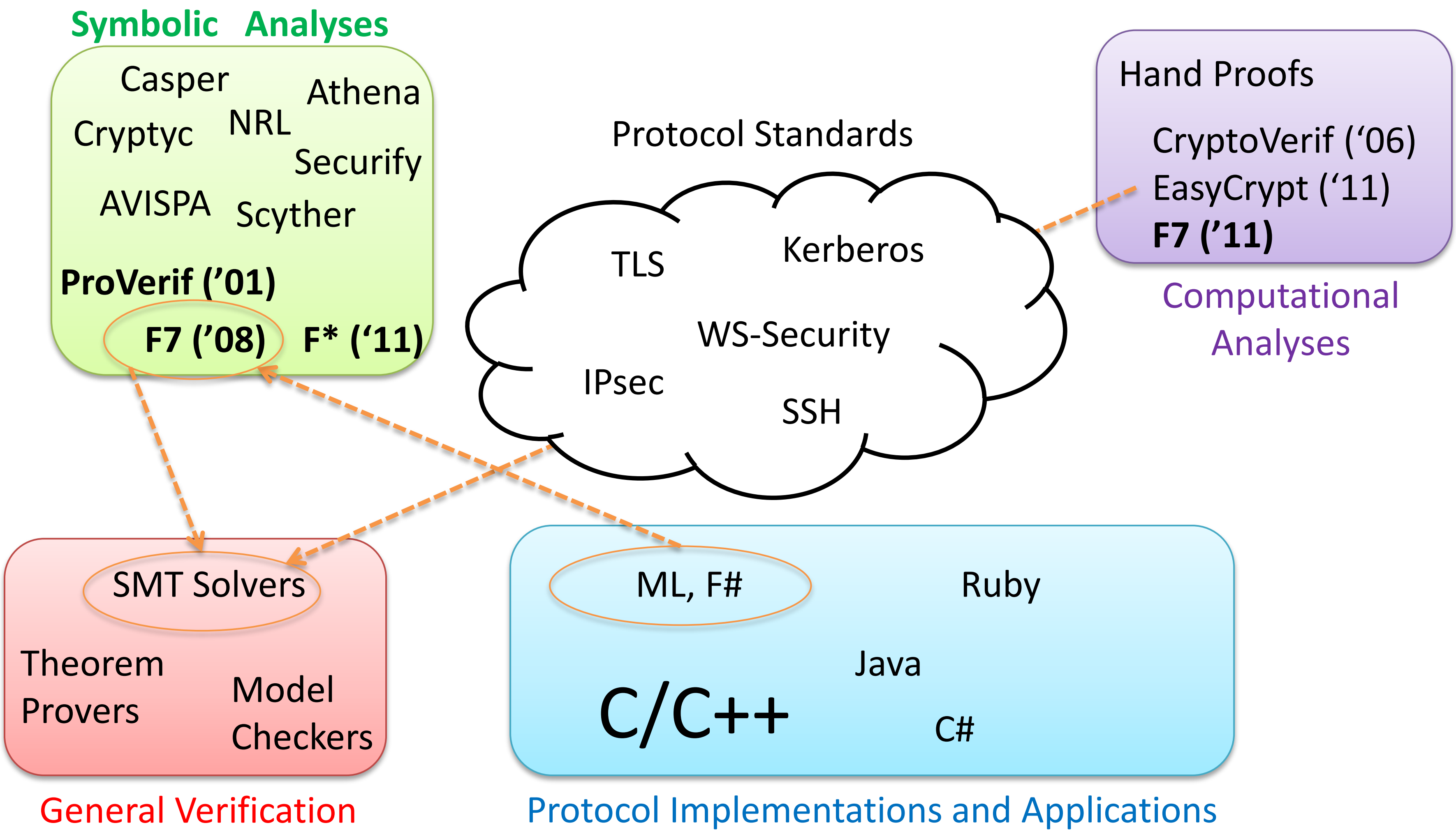
- Two verification approaches have been successfully applied to protocols and programs that use cryptography:

  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

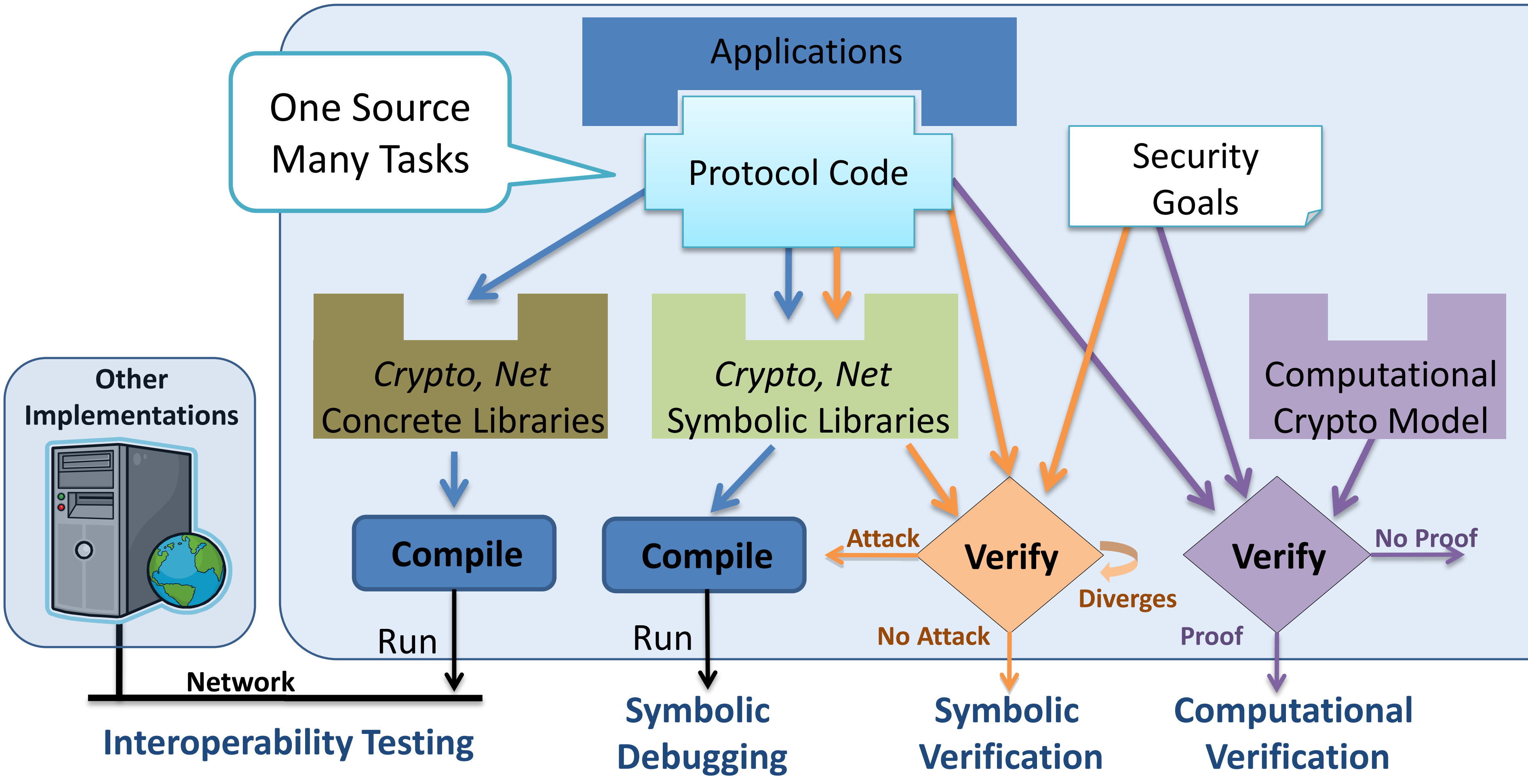
- Protocol specifications remain largely informal
  - They focus on message formats and interoperability, not on local enforcement of security properties
- Models are short, abstract, hand-written
  - They ignore large functional parts of implementations
  - Their formulation is driven by verification 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)



F7:

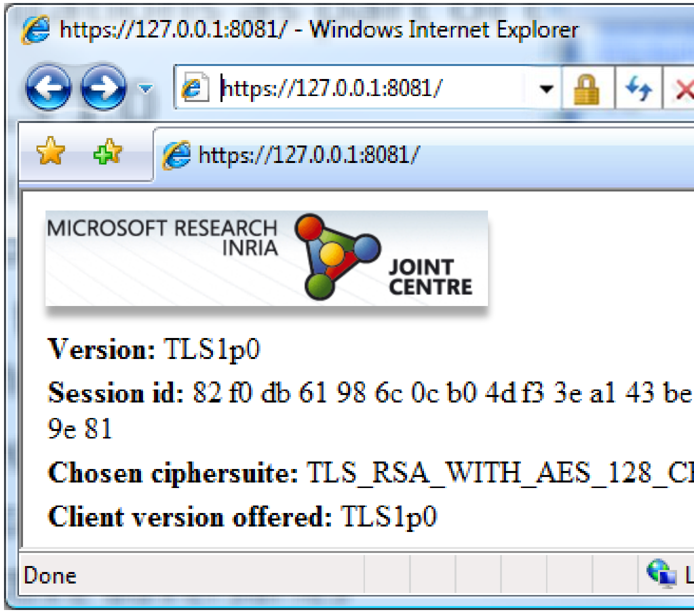
automated program verification
using refinement types

Programming Language: F#

- “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
- Great for research & prototyping
- Clean strongly-typed semantics
  - Modular programming based on strong interfaces
  - Algebraic data types with pattern matching useful for cryptography & message formats

TLS in F#



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)

Basis for Verification: **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:
– $n : \mathrm{int}\{n \geq 0\}$ 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

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 program assertions
  - Such predicates usefully represent security-related concepts like roles, permissions, events, compromises

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
```

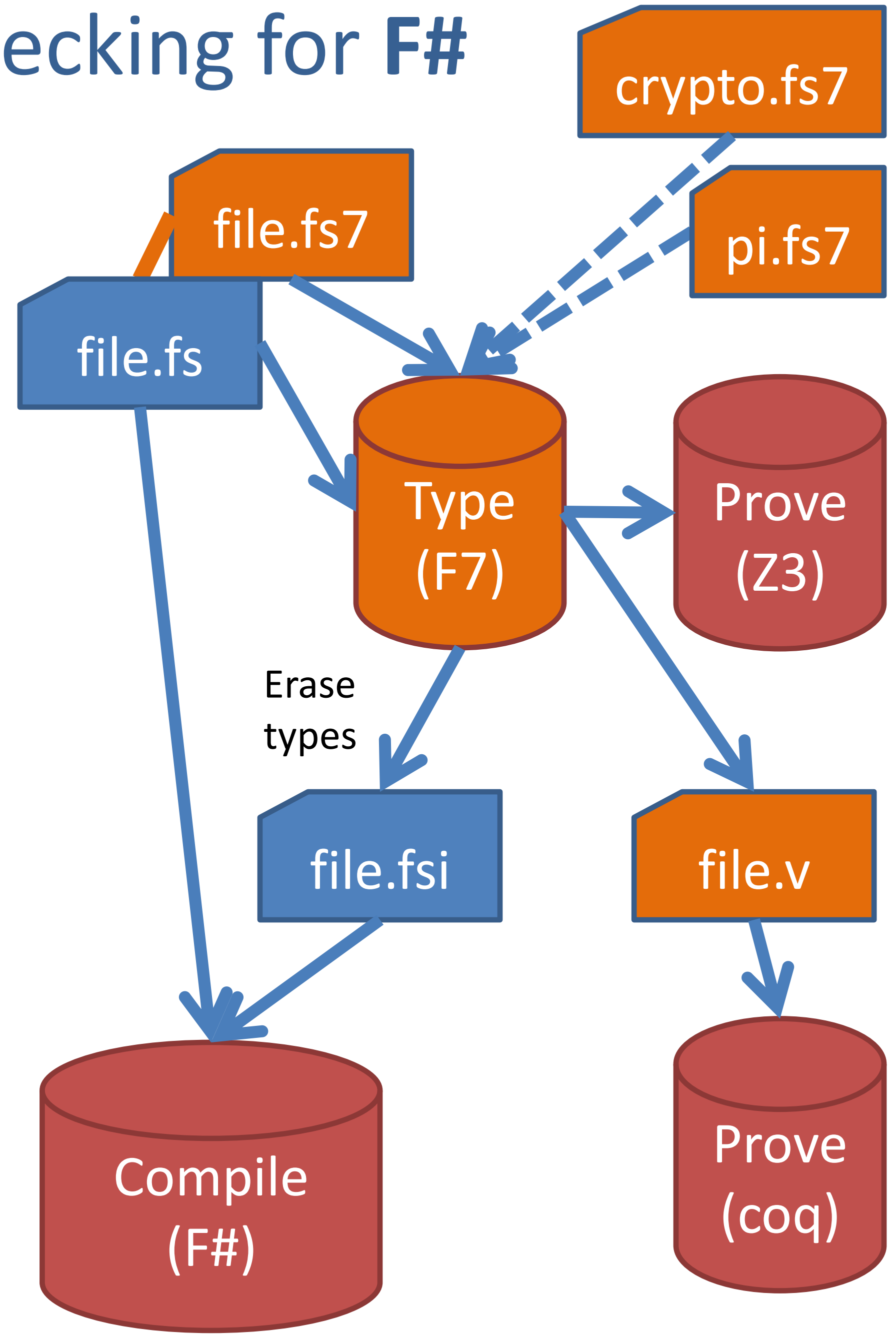
Access control with refinement types

$$
\begin{aligned}
&\textbf{val}\ \text{read: file:string}\{\text{CanRead(file)}\} \rightarrow \text{string}\\
&\textbf{val}\ \text{delete: file:string}\{\text{CanDelete(file)}\} \rightarrow \text{unit}\\
&\textbf{val}\ \text{publicfile: file:string} \rightarrow \text{unit}\{\text{PublicFile(file)}\}
\end{aligned}
$$

- 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 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



The Core Language (FPC):

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

Refinement types

- An assembly of standard components

$$
\begin{array}{lll}
H, T, U ::= & & \text{type} \\
 & \alpha & \text{type variable} \\
 & \mathsf{unit} & \text{unit type} \\
 & \Pi x : T.\, U & \text{dependent function type (scope of } x \text{ is } U) \\
 & \Sigma x : T.\, U & \text{dependent pair type (scope of } x \text{ is } U) \\
 & T + U & \text{disjoint sum type} \\
 & \mu\alpha.T & \text{iso-recursive type (scope of } \alpha \text{ is } T) \\
 & \{x : T \mid C\} & \text{refinement type (scope of } x \text{ is } C)
\end{array}
$$

- For example, **type** filename = x:string{ CanRead(x)}
  declares a type of strings for filename with the read access right

Safety by typing

$E \vdash \diamond$	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**
$E \vdash T :: \nu$	in E, type T has kind $\nu \in \{\mathbf{pub}, \mathbf{tnt}\}$
$E \vdash T <: U$	in E, type T is a subtype of type U
$E \vdash A : T$	in E, expression A has type T

$\mu ::=$	environment entry
$\quad \alpha :: \nu$	kinding
$\quad \alpha <: \alpha'$	subtyping
$\quad a : T \updownarrow$	name (of channel type)
$\quad x : T$	variable (of any type)
$E ::= \mu_1, \ldots, \mu_n$	environment

An expression A is *safe* if and only if,
in all evaluations of A, all assertions succeed.

Theorem 1 (Safety by Typing) *If $\varnothing \vdash A : T$ then A is safe.*

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

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

We can assume any formula

We can assert any formula that follows from *E*

Logical Invariants for Symbolic Cryptography

Our crypto libraries for F7 v2.0

Symbolic Method:
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.

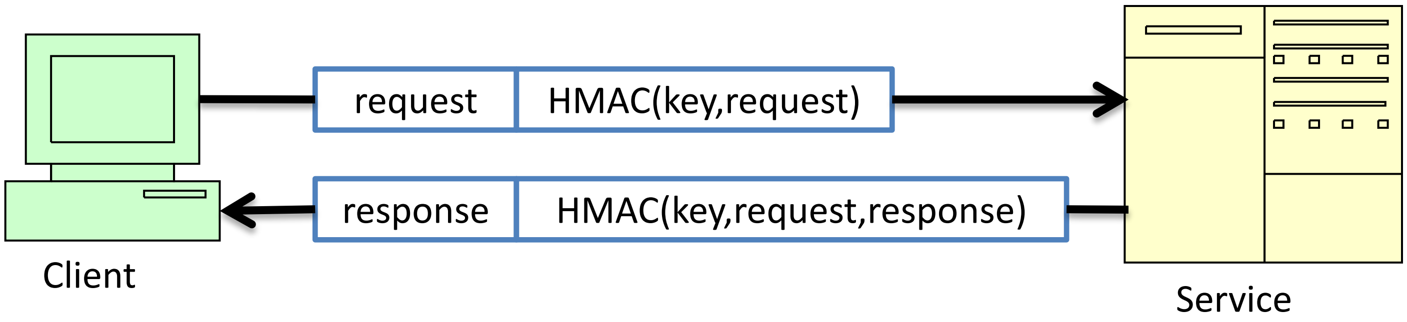
– After verifying the MAC (if no principals are compromised),
$KeyAB(k_{ab},a,b) \wedge Bytes(hash\ k_{ab}\ x) \Longrightarrow Request(a,b,x)$.

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

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

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

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

F# Implementation

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

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:

$$
\begin{array}{l}
\textbf{let } \mathit{setup}\ (a{:}\mathrm{str})\ (b{:}\mathrm{str}) = \\
\quad \textbf{let } k = \mathit{mkKeyAB}\ a\ b\ \textbf{in} \\
\quad (\textbf{fun } s \to \mathit{client}\ a\ b\ k\ s), \\
\quad (\textbf{fun } \_ \to \mathit{server}\ a\ b\ k), \\
\quad (\textbf{fun } \_ \to \textbf{assume}\ (\mathit{Bad}(a));\ k), \\
\quad (\textbf{fun } \_ \to \textbf{assume}\ (\mathit{Bad}(b));\ k)
\end{array}
$$

Opponent Interface (excerpts):

$$
\begin{array}{l}
\textbf{val } \mathit{send}{:}\ \mathrm{conn} \to \mathrm{bytespub} \to \mathrm{unit} \\
\textbf{val } \mathit{recv}{:}\ \mathrm{conn} \to \mathrm{bytespub} \\
\textbf{val } \mathit{hmacsha1} : \mathrm{keypub} \to \mathrm{bytespub} \to \mathrm{bytespub} \\
\textbf{val } \mathit{hmacsha1Verify} : \mathrm{keypub} \to \mathrm{bytespub} \to \mathrm{bytespub} \to \mathrm{unit} \\
\textbf{val } \mathit{setup}{:}\ \mathrm{strpub} \to \mathrm{strpub} \to \\
\quad (\mathrm{strpub} \to \mathrm{unit}) * (\mathrm{unit} \to \mathrm{unit}) * (\mathrm{unit} \to \mathrm{keypub}) * (\mathrm{unit} \to \mathrm{keypub})
\end{array}
$$

Sample 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 (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 $\rightarrow k$: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$\{IsMAC(h,k,b)\}$

$\forall h,k,b.\ IsMAC(h,k,b) \wedge Bytes(h) \Rightarrow MACSays(k,b) \vee Pub(k)$

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

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

(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.

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):

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

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

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

MAC-and-Encrypt (as in SSH):

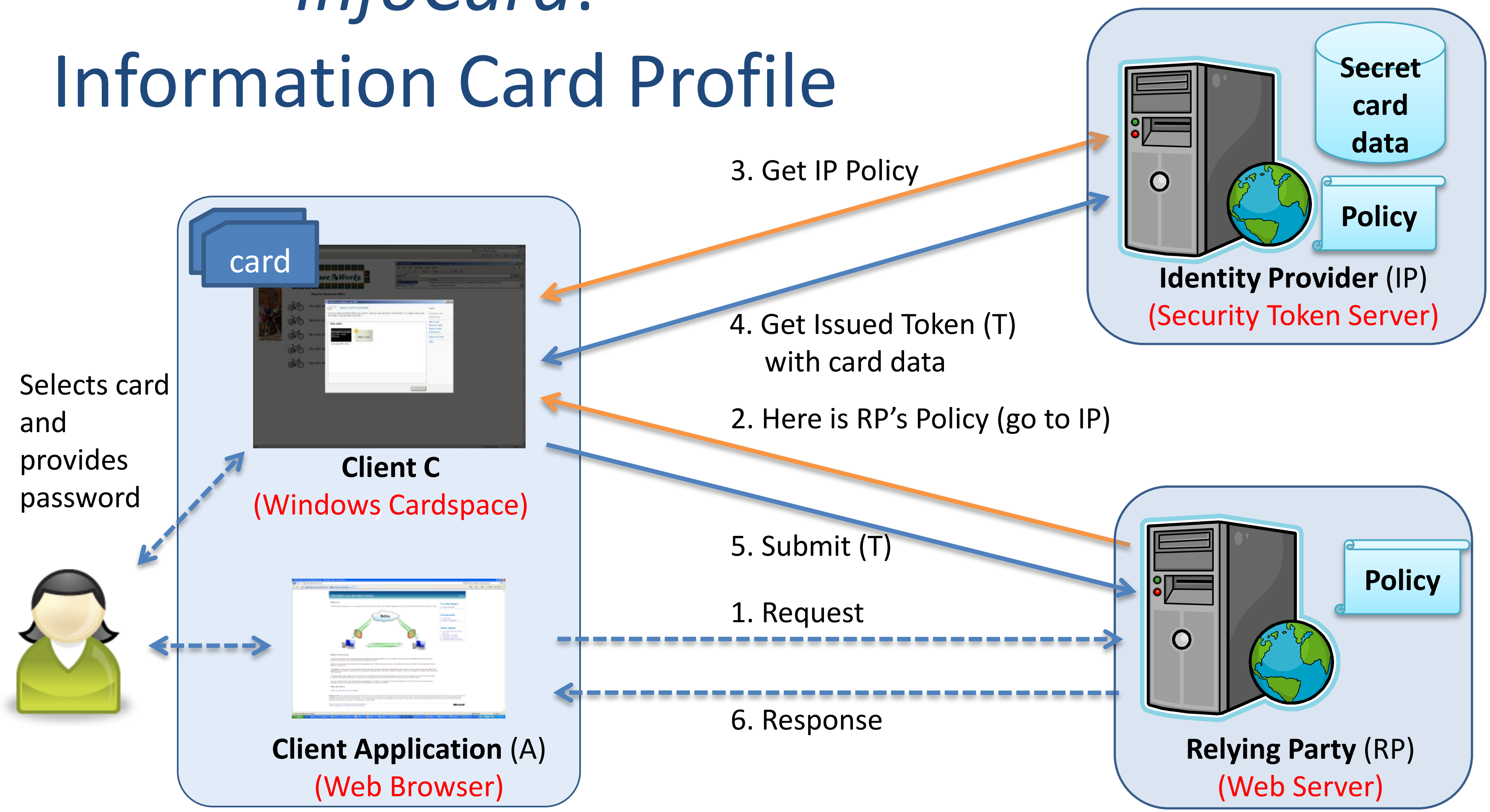
$a \to b: \qquad \mathit{aes}\ k^e_{ab}\ t \mid \mathit{hmacsha1}\ k^m_{ab}\ t$

CardSpace

& Web Services Security

Verification Case Study

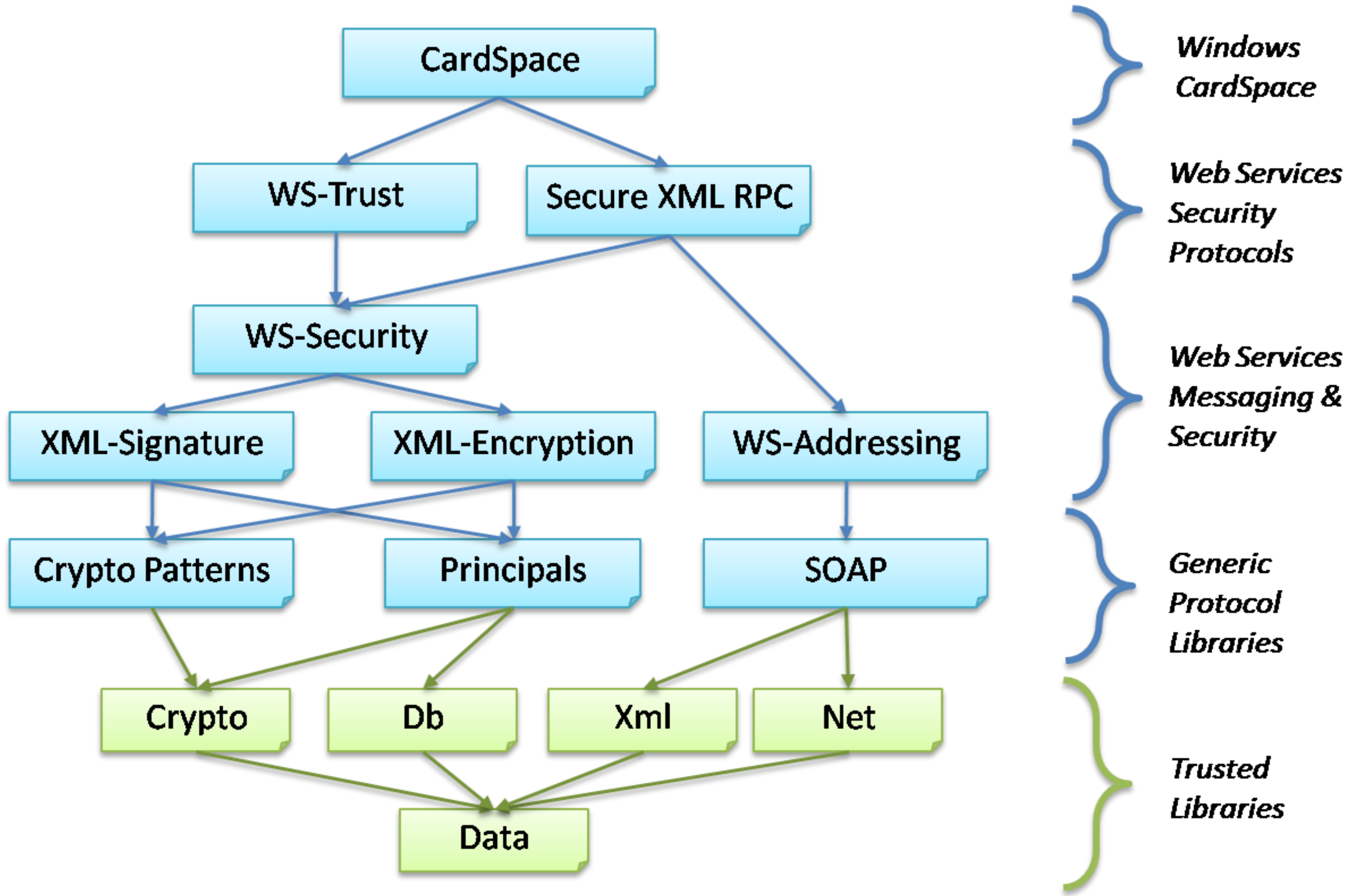
InfoCard: Information Card Profile



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

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

Computational Soundness for Cryptographic Typechecking

What about standard crypto assumptions?
(concrete, probabilistic, poly-time)

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

1. We adapt our language for probabilistic polynomial-time assumptions
2. We type protocols and applications against
   refined typed interfaces for cryptography (automatically)

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

HMAC & Int-CMA

Sample computational soundness for keyed hash functions

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:

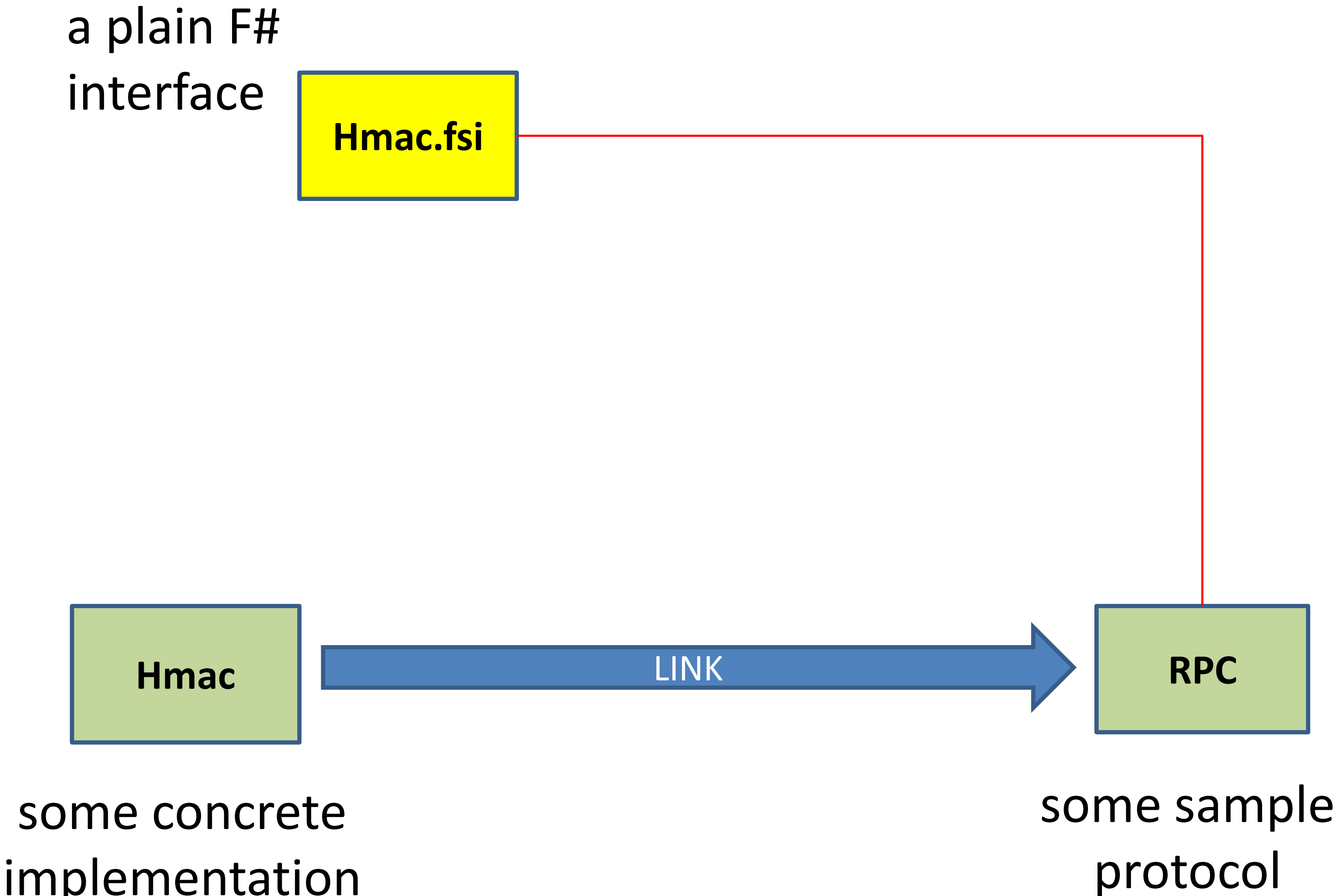
```
let CMA opponent =
  let k = Hmac.GEN() in
  let log = ref [] in
  let mac t = log := t::!log; Hmac.MAC k t in
  let verify t m = Hmac.VERIFY k t m in
  let (t,m) = opponent mac verify in
  let forged = Hmac.VERIFY k t m && not(mem !log r)
  assert (forged = false)
```

The opponent
can forge a signature
only with negligible probability

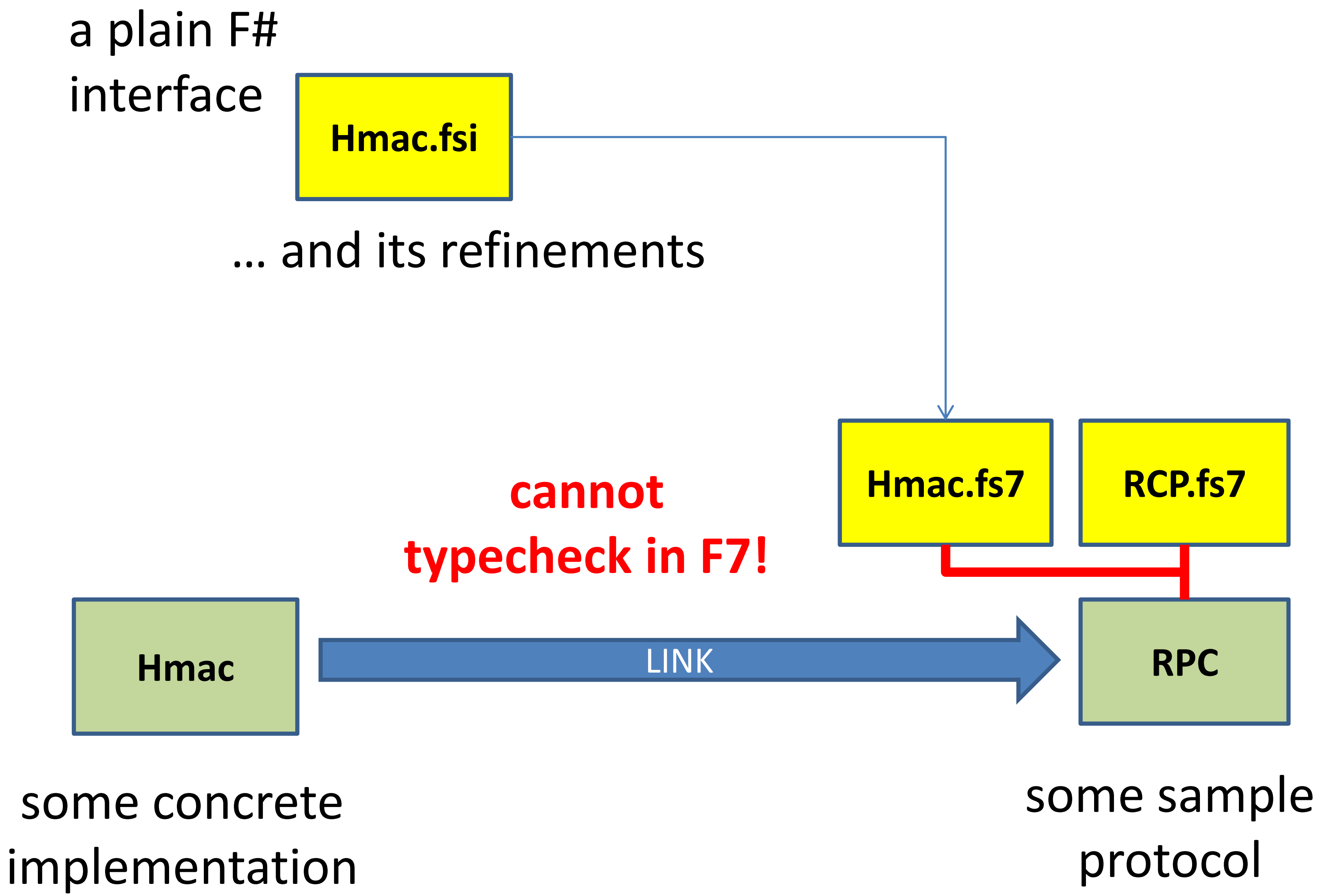
A PPT implementation **Hmac** (with parameter η) is CMA-secure when, for any PPT expression O, for all c and sufficiently large η,

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

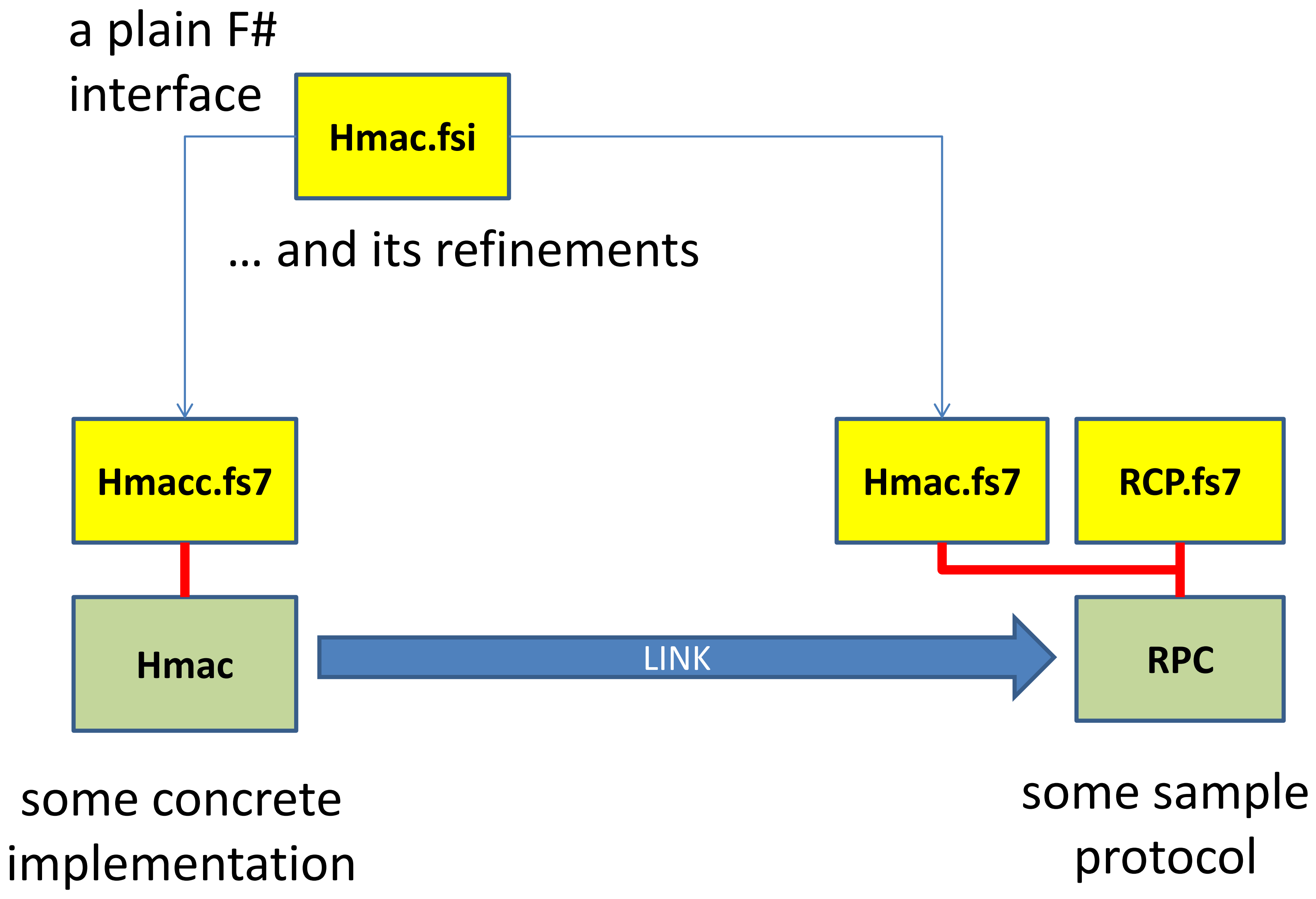
MACs: interfaces and implementations



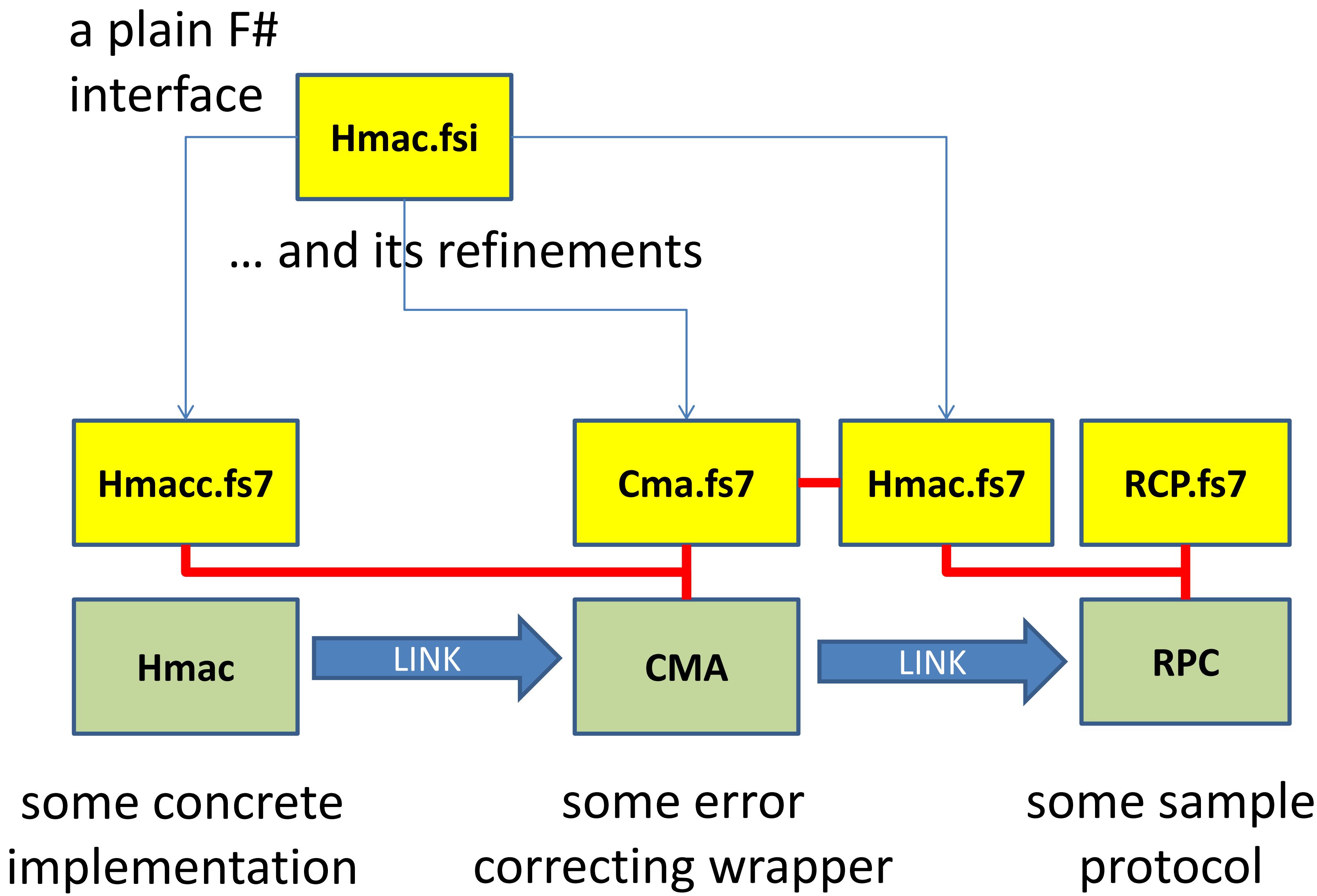
MACs: interfaces and implementations



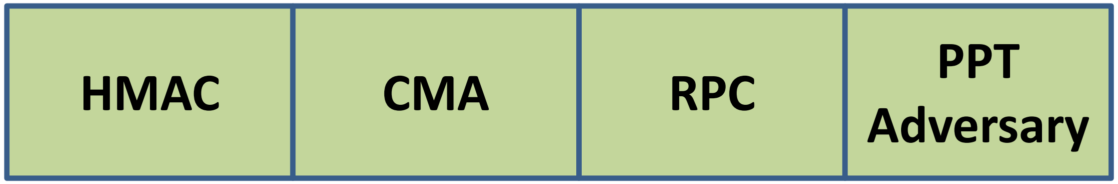
MACs: interfaces and implementations



MACs: interfaces and implementations

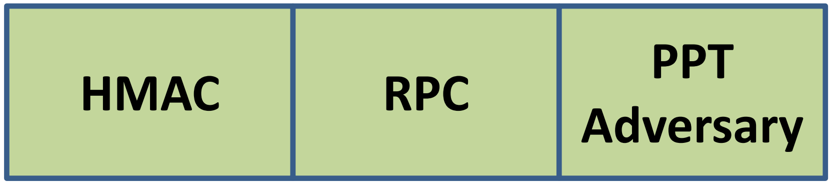


MACs: interfaces and implementations



is always safe (by typing)

is indistinguishable from

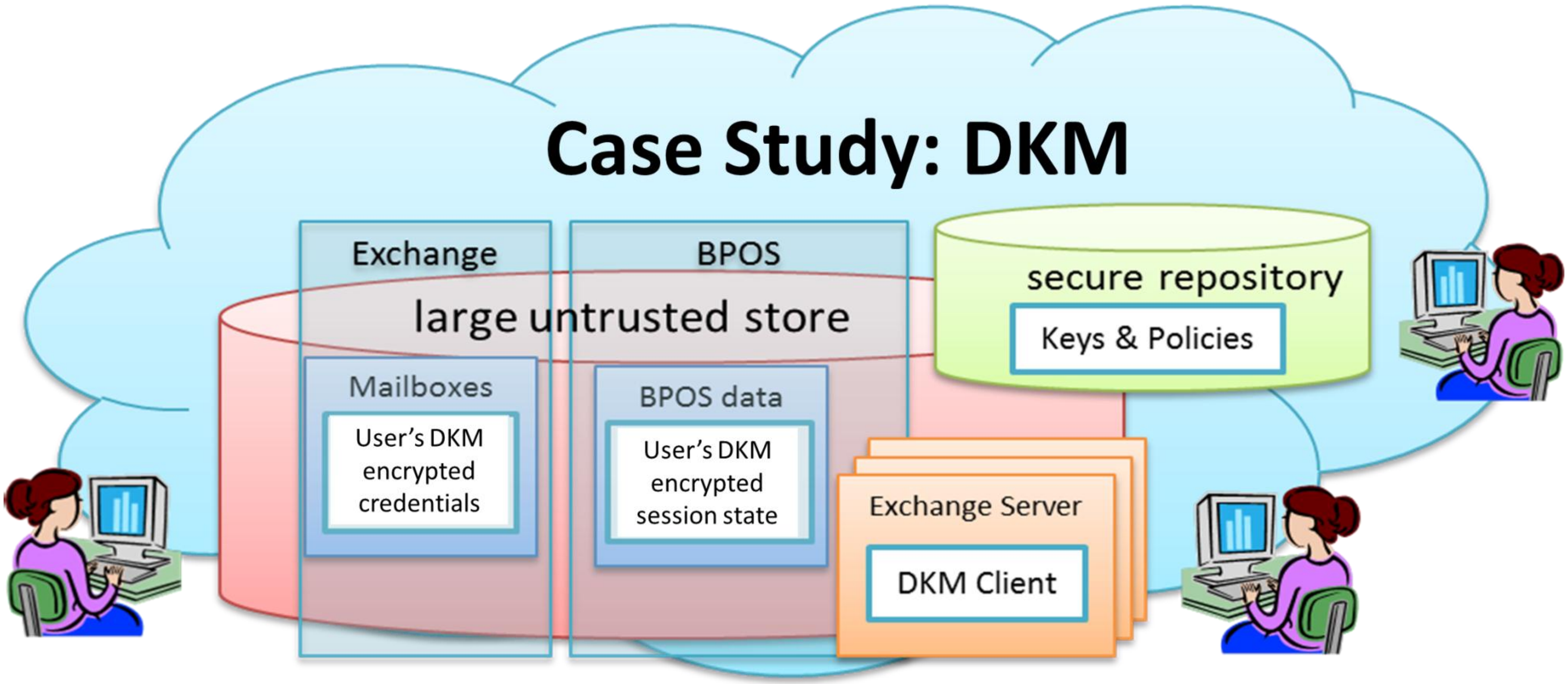


is safe too, with overwhelming probability

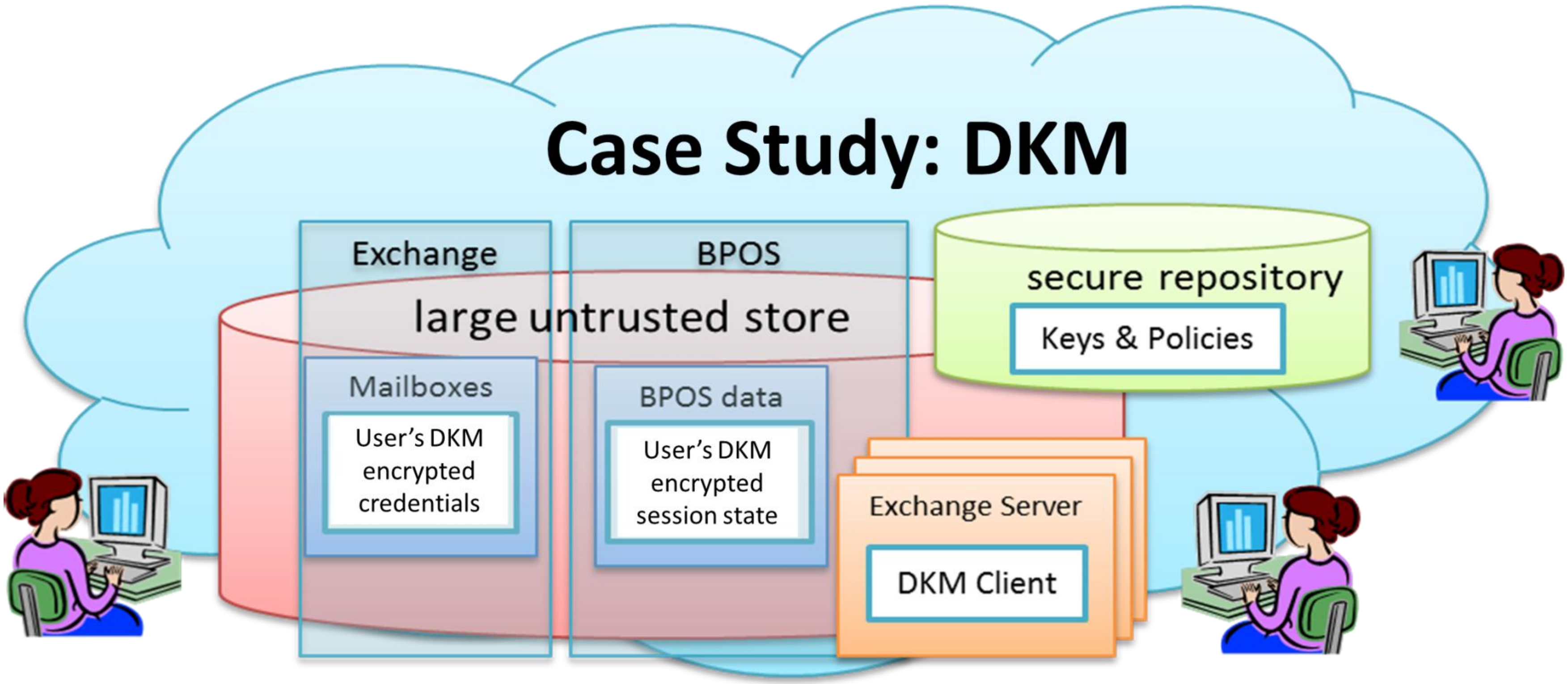
Verifying DKM,
a fresh data protection API

Case Study & Demo

Case Study: DKM



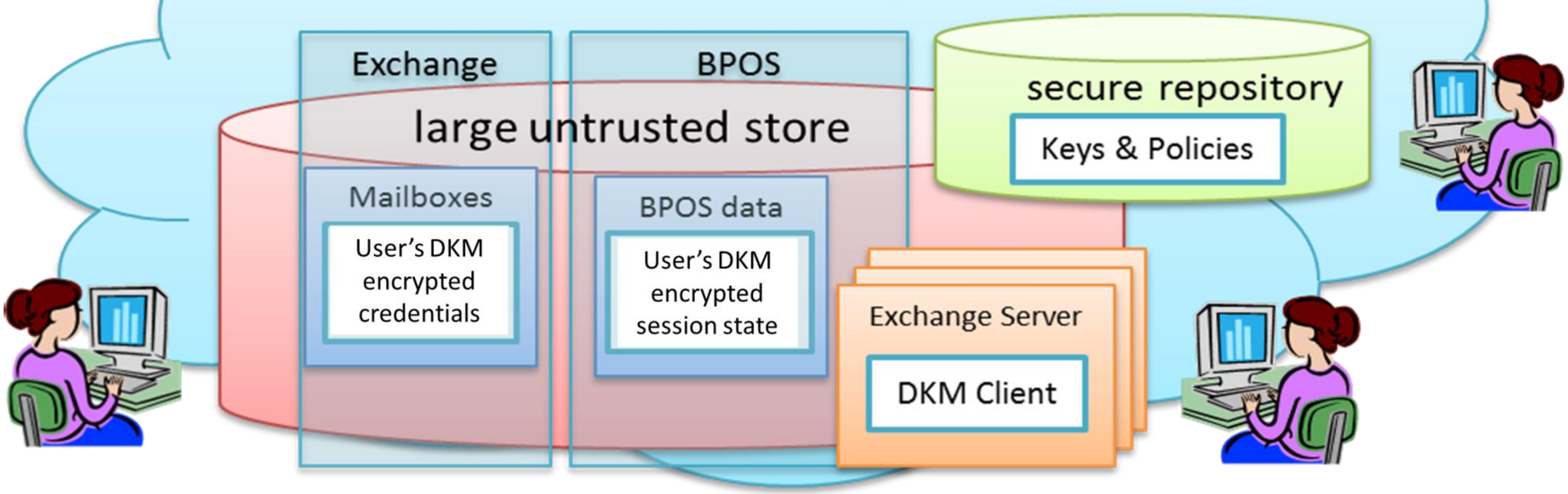
A new security API for **securing data at rest**
between multiple machines and multiple users/service accounts



DKM unburdens developers from many common tasks.

- **Key management**. The caller does not have to worry about cryptography. The DKM library figures out which key to use based on the specified group.
- **Automated key update**.
- **Crypto agility**.
  DKM supports flexible policies for selecting crypto algorithms

Case Study: DKM



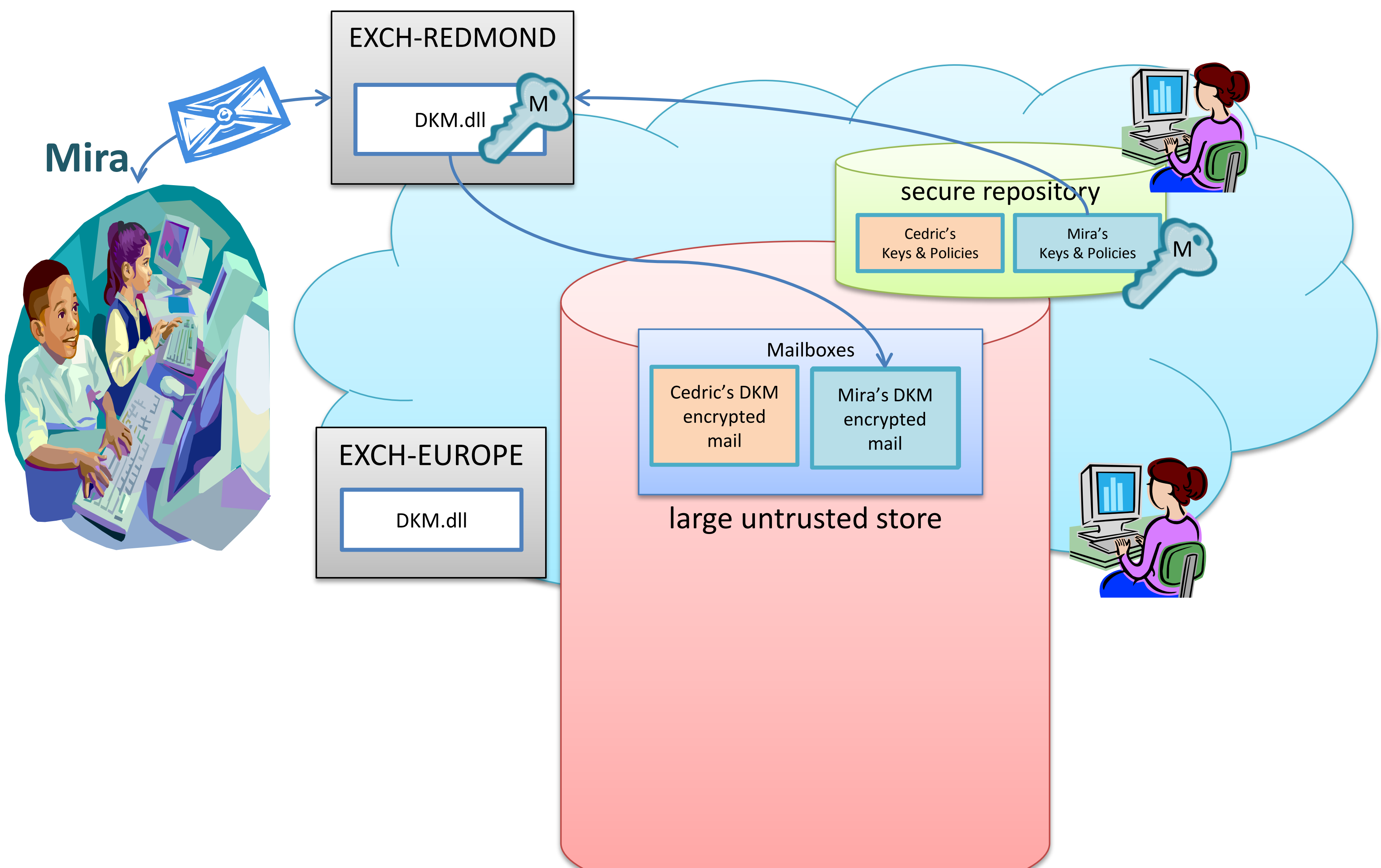
DKM is largely deployed in Microsoft datacenters

We wrote together auxiliary reference code (aka specification)

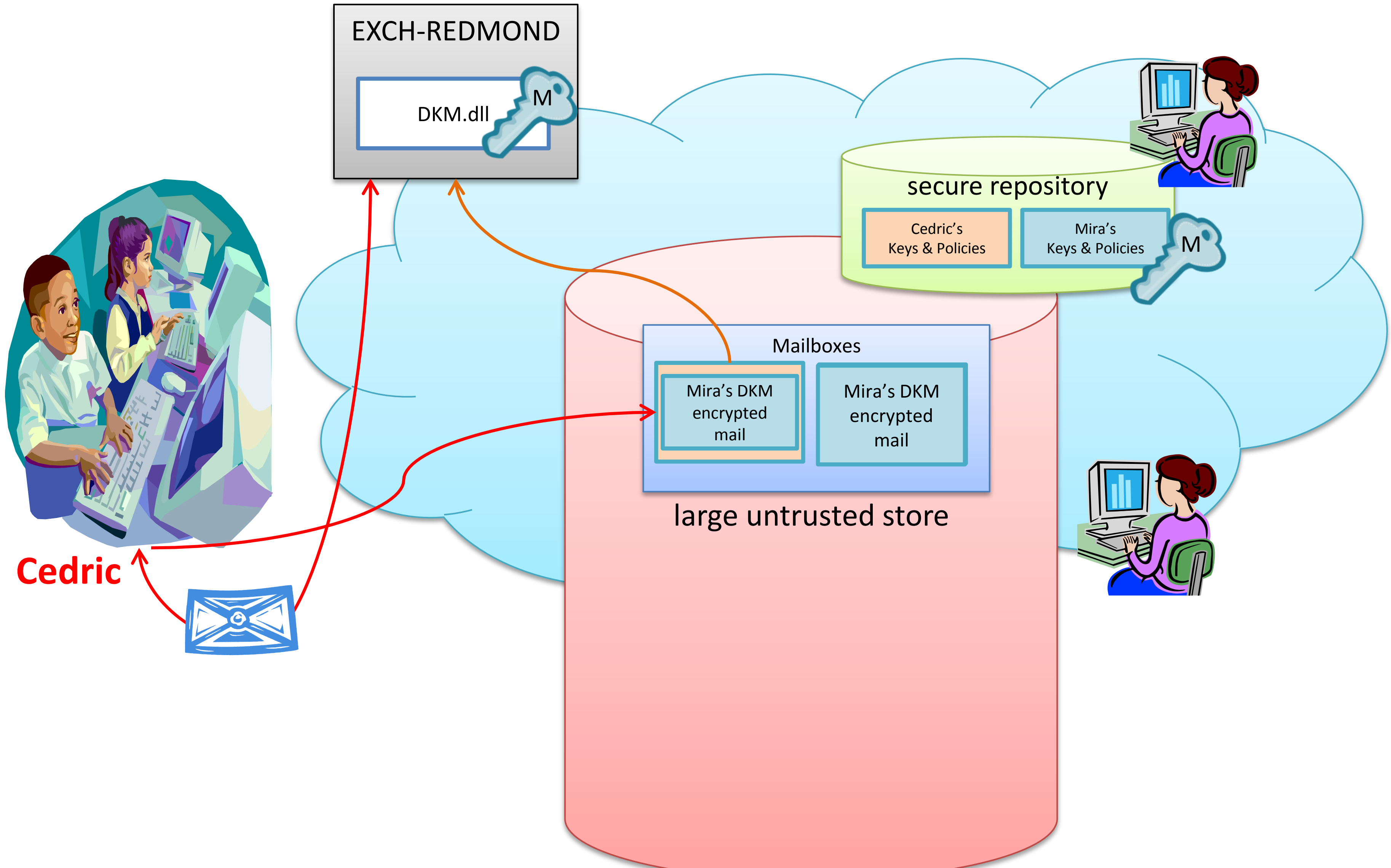
We found and fixed serious flaws (early in the process)

We verified DKM once fixed (verification time: 17 s)

DKM Encrypted Email (Normal)



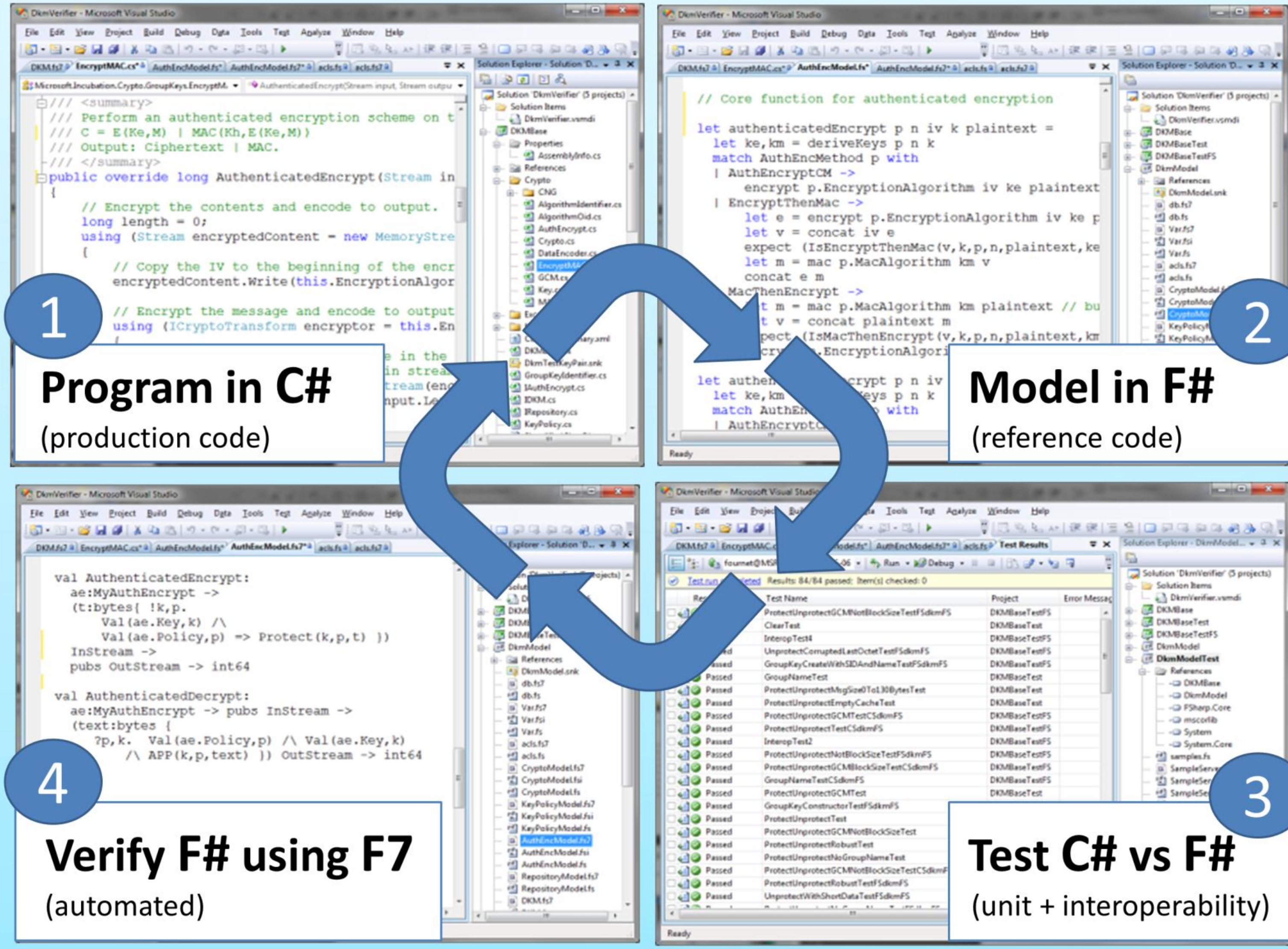
DKM Encrypted Email (Attack)



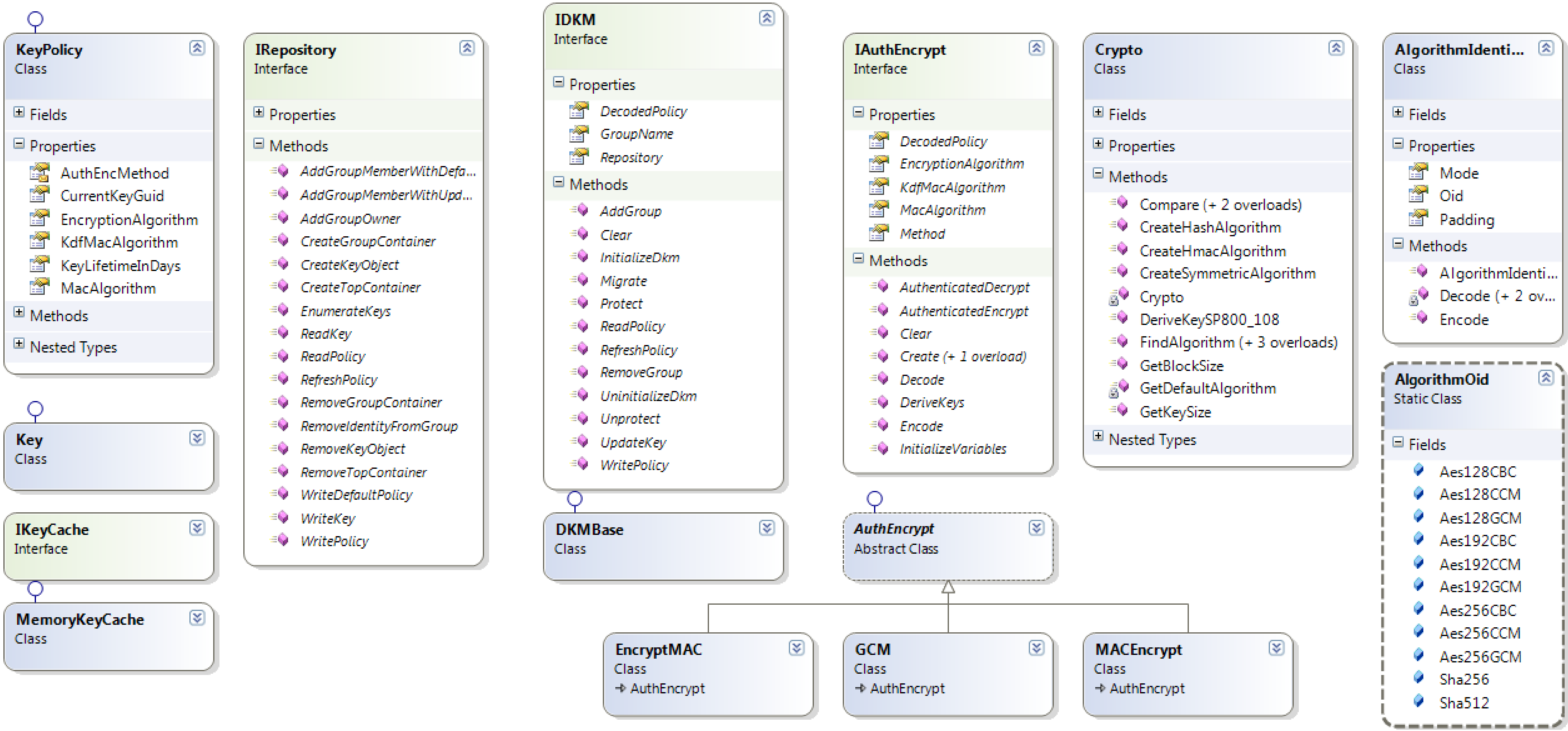
```
Verifying DKM
time ../../../../../../../inria/lang-sec/msrc/cvk/bin/f7.exe -nokindcheck --define f7
-pervasives ../../../../../../../inria/lang-sec/msrc/cvk/samples/lib/fs7-interfaces/pervasives.fs7
-tuples ../../../../../../../inria/lang-sec/msrc/cvk/samples/lib/fs7-interfaces/tuples.fs7
../../../../../../../inria/lang-sec/msrc/cvk/samples/lib/fs7-interfaces/pi.fs7
../../../../../../../inria/lang-sec/msrc/cvk/samples/lib/fs7-interfaces/list.fs7
db.fs7 var.fs7 CryptoModel.fs7 Acls.fs7 KeyPolicyModel.fs7 RepositoryModel.fs7 AuthEncModel.fs7
DKM.fs7 DKM.fs -scripts DKM | tee DKM.tc7 | egrep --color "ERROR|WARNING"
ERROR: failed type checking val DKM.DkmUnprotect

0m9.863s

  65590 2011-02-10 11:42 DKM.tc7
 111421 2011-02-10 11:42 DKM.smp
   8331 2011-02-10 11:41 DKM.fs
   2940 2011-01-12 14:26 DKM.fs7
```

```
Verifying DKM
time ../../../../../../../inria/lang-sec/msrc/cvk/bin/f7.exe -nokindcheck --define f7
-pervasives ../../../../../../../inria/lang-sec/msrc/cvk/samples/lib/fs7-interfaces/pervasives.fs7
-tuples ../../../../../../../inria/lang-sec/msrc/cvk/samples/lib/fs7-interfaces/tuples.fs7
../../../../../../../inria/lang-sec/msrc/cvk/samples/lib/fs7-interfaces/pi.fs7
../../../../../../../inria/lang-sec/msrc/cvk/samples/lib/fs7-interfaces/list.fs7
db.fs7 var.fs7 CryptoModel.fs7 Acls.fs7 KeyPolicyModel.fs7 RepositoryModel.fs7 AuthEncModel.fs7
DKM.fs7 DKM.fs -scripts DKM | tee DKM.tc7 | egrep --color "ERROR|WARNING"

0m8.058s

    819 2011-02-10 11:42 DKM.tc7
 107561 2011-02-10 11:42 DKM.smp
   8329 2011-02-10 11:42 DKM.fs
   2940 2011-01-12 14:26 DKM.fs7
```

Joint Development & Verification



The DKM Codebase



Crypto Agility?

Encrypt(Policy, key)

- Legacy systems:
  data must be accessible in 10+ years
  even if protected by weak algorithms!
- Plug-and-play cryptography:
  algorithms get broken & replaced
    - Encrypt: ~~DES~~; 3DES; AES-128; AES-256 …
    - Hash: ~~MD5~~; SHA-1; SHA-256; SHA-512; SHA3 …

Production code relying on crypto can be verified

- New tasks:
  - Write reference code in F#
  - When using non-standard crypto, adapt F7 verification libraries
- Verification is fast & automated
  - Part of the build/test process

language	LOCs
C#	~ 20,000
F#	1,447
F7	855

task	time
Build + unit tests	3 m
Verify (F7)	17 s

Summary

- We verify crypto protocol implementations by **refinement typechecking**
  - Verification is modular
  - We use abstract types and refinements to control the usage of cryptography
  - Except for the crypto libraries, proofs are automated & fast
  - Applied to full-fledged implementations of industrial standards and protocols
- This talk: **integrity properties**
  - Active adversaries range over programs with access to specially-crafted interfaces that account for potential partial compromise
  - Verification is cryptographically sound, both symbolically and computationally
- Omitted: **secrecy properties**
- Our approach and libraries are language-independent (in principle)
  - So far we use mostly F# & F7

Questions?

- We verify crypto protocol implementations
  by **refinement typechecking**
  - Verification is modular
  - We use abstract types and refinements to control the usage of cryptography
  - Except for the crypto libraries, proofs are automated & fast
  - Applied to full-fledged implementations
    of industrial standards and protocols
- This talk: **integrity properties**
  - Active adversaries range over programs with access to specially-crafted interfaces that account for potential partial compromise
  - Verification is cryptographically sound, both symbolically and computationally
- Omitted: **secrecy properties**
- Our approach and libraries are language-independent (in principle)
  - So far we use mostly F# & F7