Modular Verification of Security Protocol Code

Cédric Fournet
Microsoft Research

Joint work with Karthik Bhargavan and Andy Gordon

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

Protocol Verification 1978—2008

- In **Using encryption for authentication in large networks of computers (CACM 1978)**, Needham and Schroeder set up a verification challenge:

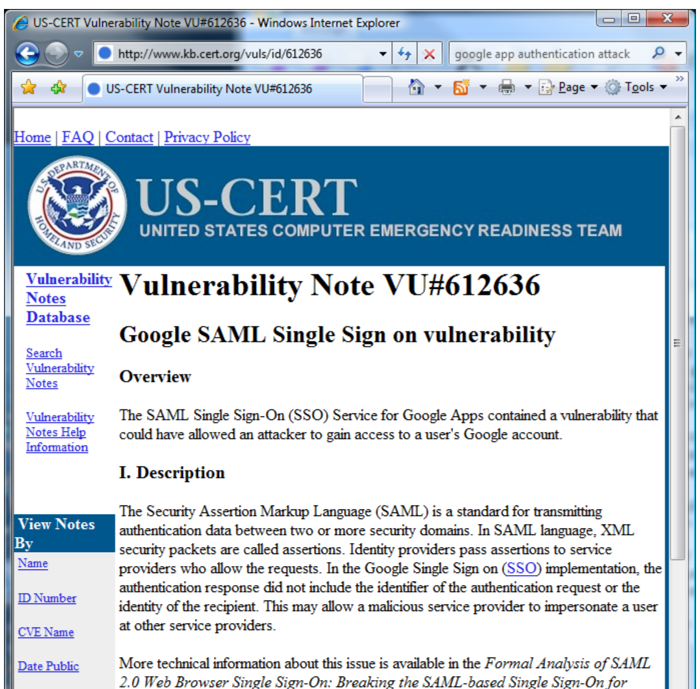
> "Protocols such as those developed here are prone to extremely subtle errors that are unlikely to be detected in normal operation.
>
> The need for techniques to verify the correctness of such protocols is great, and we encourage those interested in such problems to consider this area."

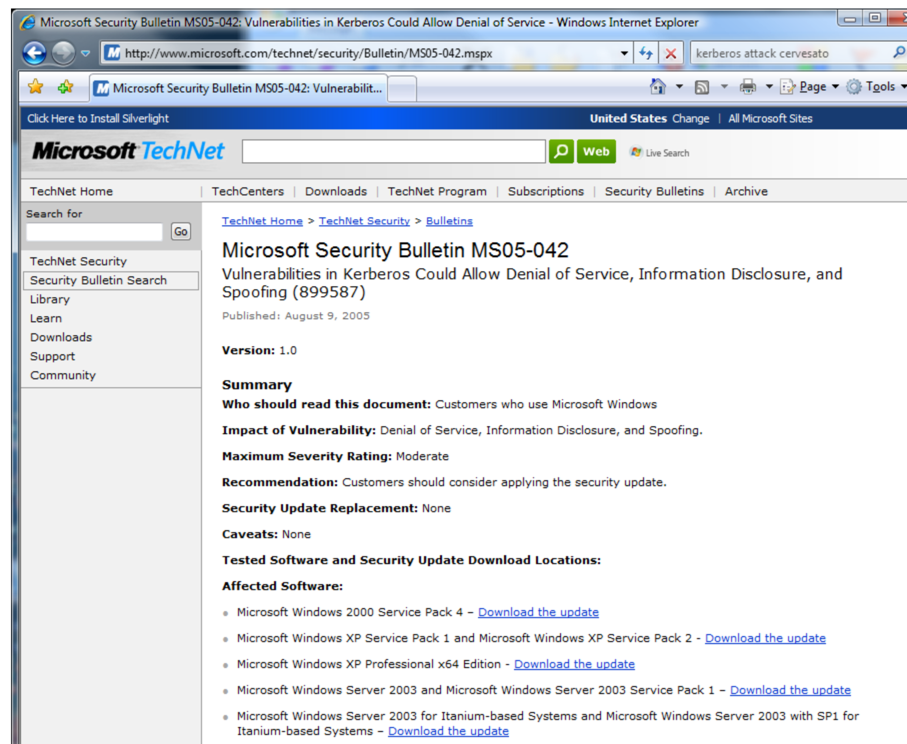
Nowadays,

- Authentication and secrecy properties for basic protocols have been thoroughly studied
- After intense effort on symbolic reasoning, techniques and tools are available for automatically proving these properties
  - Athena, TAPS, ProVerif, CryptoVerif, FDR, AVISPA, etc
- We can automatically verify most security properties for detailed models of crypto protocols
  - IPSEC, Kerberos, Web Services, Infocard, TLS, ...

Cryptographic Protocols (Still) Go Wrong

- Both design and implementations
  - Most standards got it wrong once or twice (SSL, SSH, IPSEC, 802.11)
  - Implementation details matter!
- For example, recent flaws in Google single-sign-on, in Kerberos

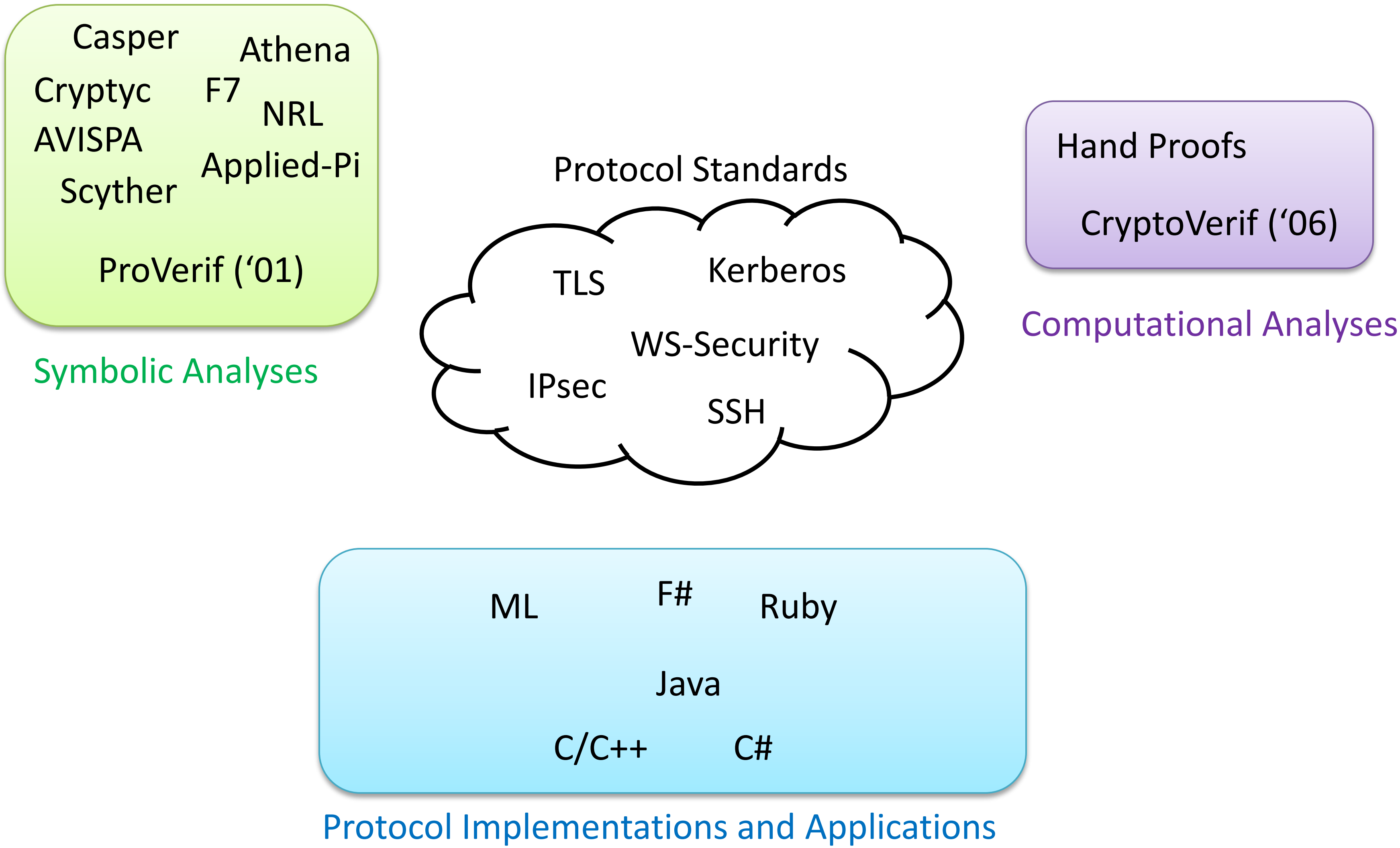




Security Verification?

- Best practice: apply formal methods and tools throughout the protocol design & review process
- Not so easy
  - Specifying a protocol is a lot of work
  - Most practitioners don't understand formal models
- Protocols go wrong because...
  - they are logically flawed, or
  - they are used wrongly, or
  - they are wrongly implemented
- Some troublesome questions
  1. How to relate crypto protocols to application security?
  2. How to relate formal models to protocol implementations?

Specs, Code, and Formal Tools



Goal: Crypto Verification Kit

MICROSOFT SECURITY
DEVELOPMENT LIFECYCLE

SECURITY ENGINEERING & COMMUNITY

OCTOBER 1, 2008

Version 4.1

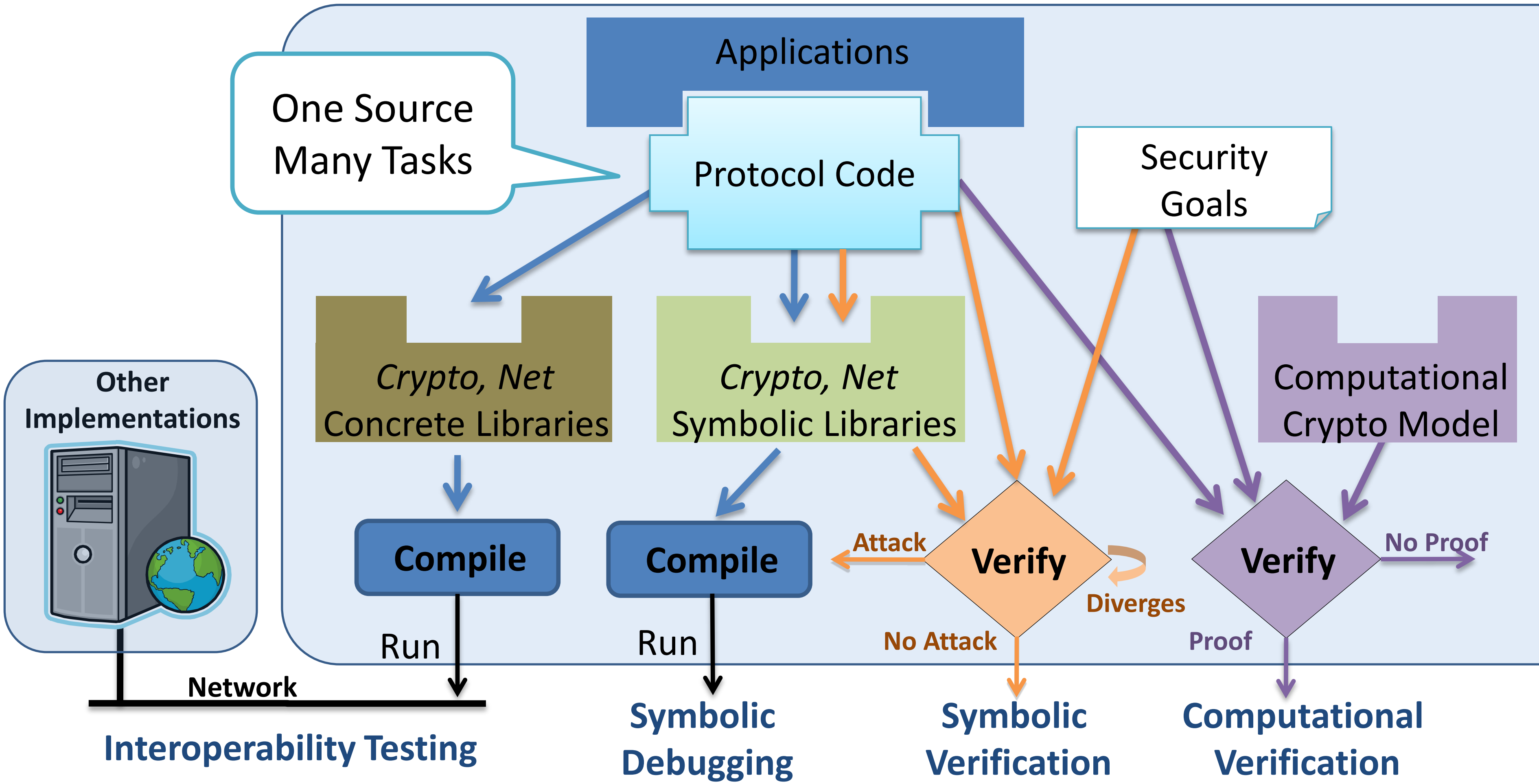
> Innovative use of cryptographic constructs often results in subtle (or not so subtle) mistakes. Using standard algorithms in standard ways, or getting expert advice from the crypto board greatly reduces the odds of a problem.

Expert review by the Crypto Board of non-standard crypto helps

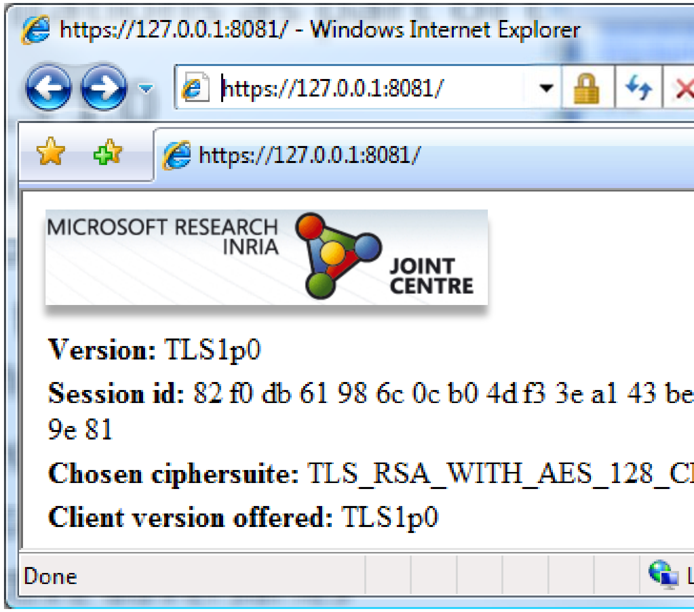
- but even experts miss bugs, and standards may be wrong
- reviewers can't check all implementation details and changes

We develop **automated tools** to verify protocols
as part of their design and development

Verifying Protocol Code (not just specs)



TLS in F# [CCS'08]



We implemented a subset of TLS (10 kLOC)

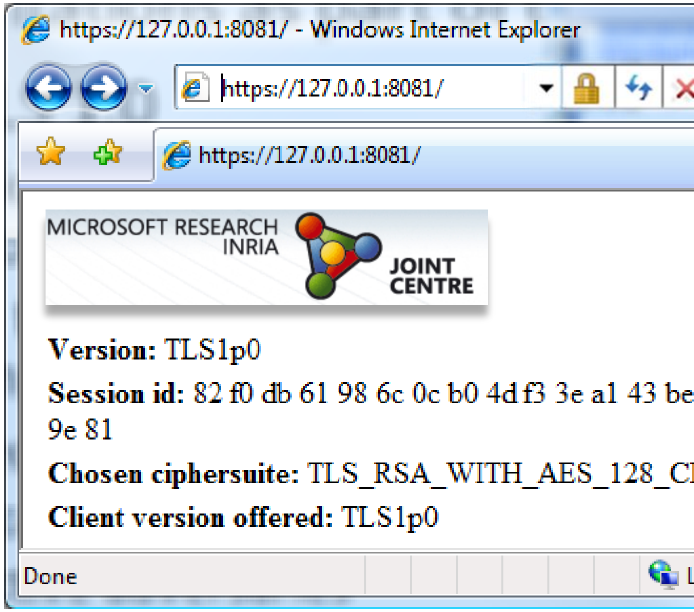
- Supports SSL3.0, TLS1.0, TLS1.1 with session resumption
- Supports any ciphersuite using DES, AES, RC4, SHA1, MD5

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 formally verified our implementation (symbolically & computationally)

TLS in F# [CCS'08]



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

OUR LATEST VERIFICATION METHOD

LOGICAL INVARIANTS FOR CRYPTOGRAPHY

Invariants for Cryptographic Structures

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

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

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

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

(3) We define logical invariants on cryptographic structures.

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

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

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

(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 $hash\ k_{ab}\ x$

OUR TOOL FOR AUTOMATED VERIFICATION

F7: REFINEMENT TYPES FOR F#

Refinement Types

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

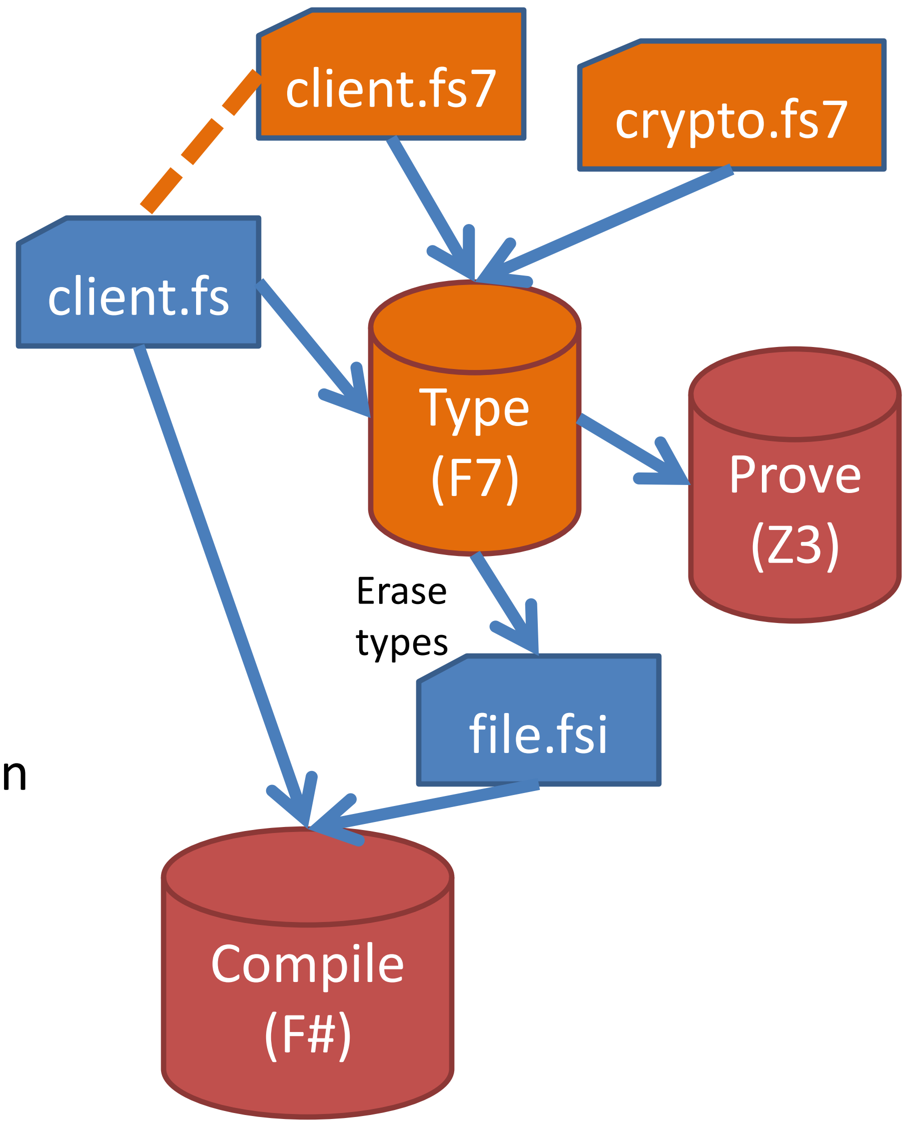
Refinement types are types of the form $x : T\{C\}$ where
– T is the base type,
– x refers to the result of the expression, and
– C is a logical formula

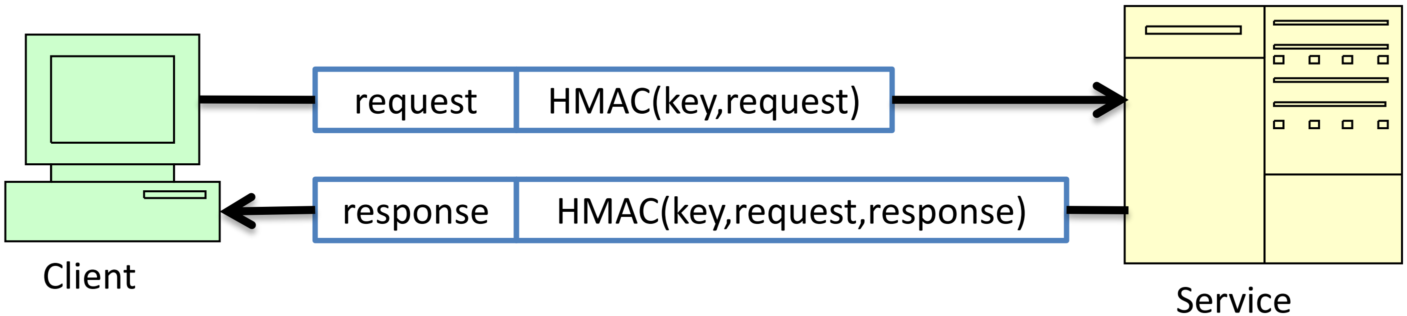
The values of this type are the values M of type T such that $C\{M/x\}$ holds.

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
– $x : \text{str}\{Request(a,b,x)\}$ is the type of strings sent as requests from a to b

F7: Refinements Types for F#

- We use extended interfaces
  - We typecheck implementations
  - We generate .fsi interfaces by erasure from .fs7
- We support a large subset of F#
  - ADTs, records, patterns, refs
  - Value- and type-polymorphism
  - Concurrency
- We call Z3, an SMT prover, on each non-trivial proof obligation





SAMPLE PROTOCOL

AUTHENTICATED RPC

Informal Description

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

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

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

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

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

$$
\begin{array}{l}
\forall a,b,s,t.\ RecvResponse(a,b,s,t) \Leftrightarrow \\
\quad (Request(a,b,s) \wedge Response(a,b,s,t)) \vee Bad(a) \vee Bad(b)
\end{array}
$$

F# Implementation

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

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

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

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 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 l**east models** for inductive definitions (not all models)
- After proving our theorems (by hand, or using other tools),
  we can **assume** them so that they can be used for typechecking

Refined Modules (Theory)

- 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.

Refined Modules (Sample)

- **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 Pattern Example:
Hybrid Encryption

Hybrid encryption implements public-key encryption for large plaintexts:

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

Hybrid Encryption:

$a \rightarrow b$: $\quad$ $\mathit{rsa\_oaep}\ pk_b\ k_{ab} \mid \mathit{aes}\ k_{ab}\ t$

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

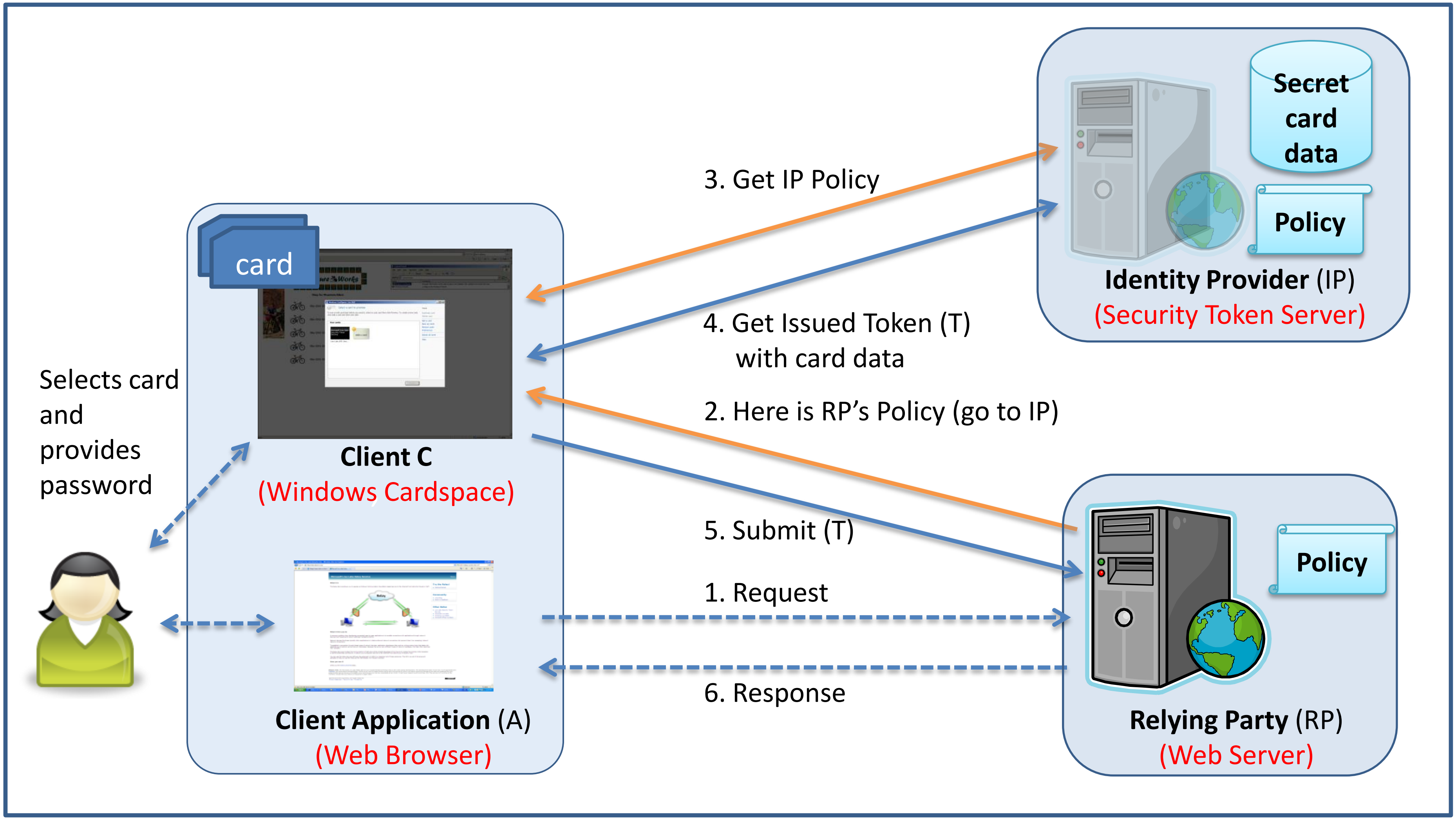
Verification relies on the single usage of the symmetric key.

CASE STUDY

CARDSPACE

& WEB SERVICES SECURITY

InfoCard: Information Card Profile v1.0

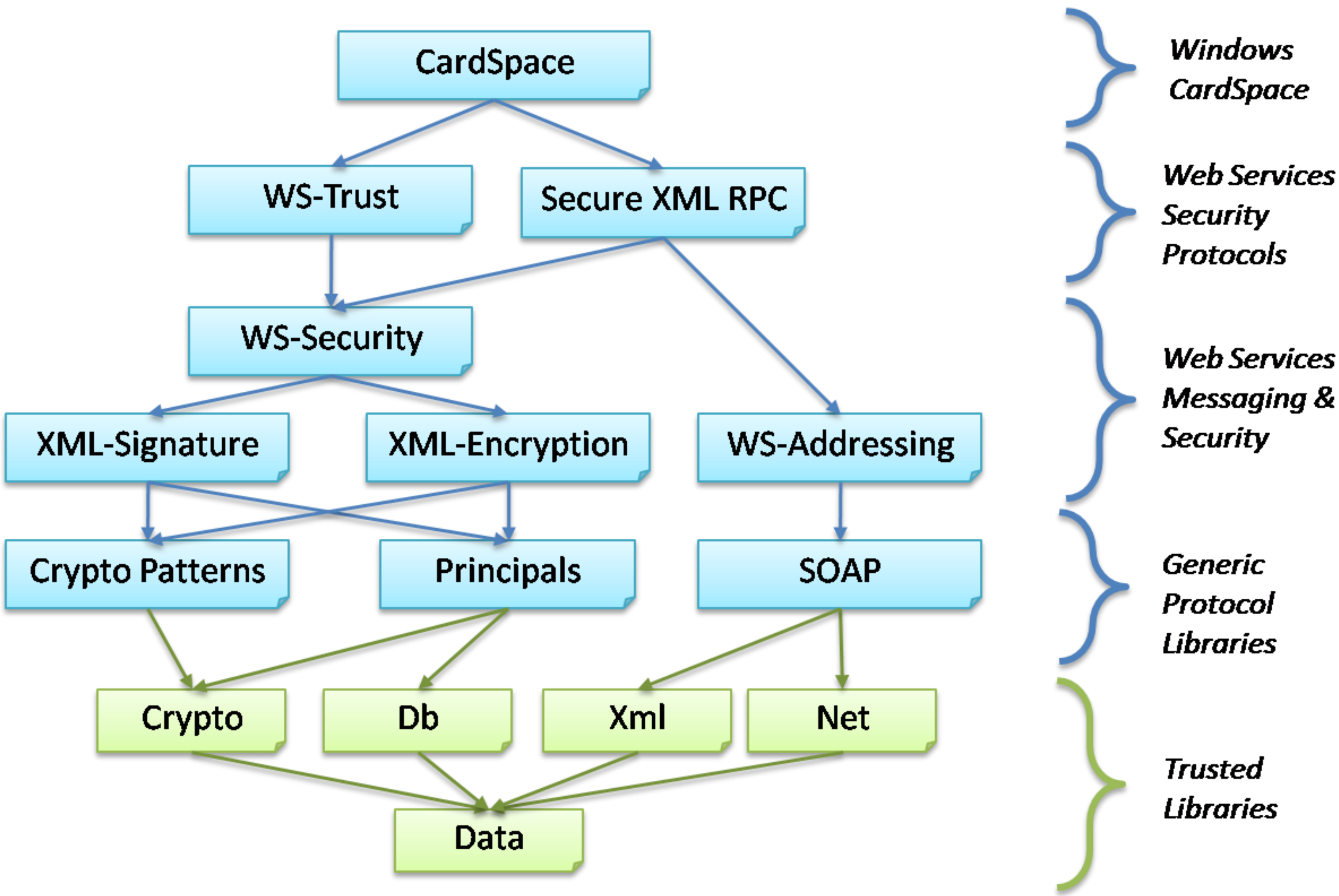


Protocol Narration (Managed Card)

Initially, **C has:** $\mathit{cardId}, \mathtt{PK}(k_{\mathrm{IP}}), \mathtt{PK}(k_{\mathrm{RP}})$; **IP has:** $k_{\mathrm{IP}}, \mathtt{PK}(k_{\mathrm{RP}}), \mathit{Card}(\mathit{cardId}, \mathit{claims}_{\mathrm{U}}, \mathit{pwd}_{\mathrm{U,IP}}, k_{\mathit{cardId}})$; **RP has:** $k_{\mathrm{RP}}, \mathtt{PK}(k_{\mathrm{IP}})$

C :	*Request* (RP, M_{req})	C *receives an application request*
U :	*Select InfoCard* (cardId, C, RP, $\mathit{pwd}_{\mathrm{U,IP}}$, $\mathit{types}_{\mathrm{RP}}$)	*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 $M_{rst} = \mathtt{RST}(\mathit{cardId}, \mathit{types}_{\mathrm{RP}}, \mathrm{RP}, \eta_{ce})$ in	*Token request message body*
	let $M_{user} = (\mathrm{U}, \mathit{pwd}_{\mathrm{U}})$ 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, $\mathit{claims}_{\mathrm{U}}$, 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 $\mathit{ppid}_{\mathit{cardId},\mathrm{RP}} = H_1(k_{\mathit{cardId}}, \mathrm{RP})$ in	*Compute PPID using card master key,* RP*'s identity*
	let $M_{tok} = \mathtt{Assertion}(\mathrm{IP}, M_{tokkey}, \mathit{claims}_{\mathrm{U}}, \mathrm{RP}, \mathit{ppid}_{\mathit{cardId},\mathrm{RP}})$ 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, $\mathit{claims}_{\mathrm{U}}$, 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: Information Card Profile v1.0



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)

Performance
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*

A new security API for protecting application data

DISTRIBUTED KEY MANAGER [2009]

Verifying DKM

(ongoing work with T. Acar, D. Shumow)

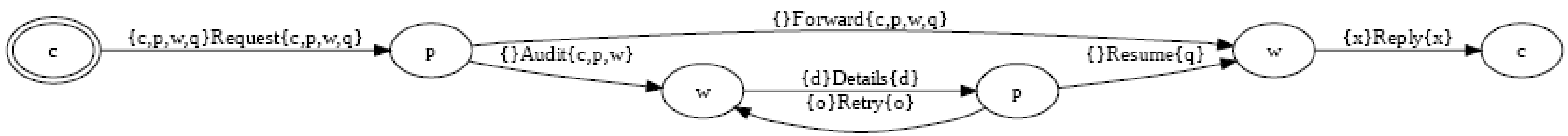
language	DKM source
C# code	~ 20,000
F# code	1,654
F7 model	300

- We supplement production code (in C#) with verified reference code (in F#)
  - We re-code three internal interfaces in F#
  - We re-use their test suites to validate our code against theirs
  - We develop (and verify) a precise security model
- We identified several security issues in their (high-quality) design and implementation
  - (...)
- **F7 can help with new security code, as part of the development process, at a reasonable cost**

VERIFYING “SECURITY COMPILERS” [2009]

F7-Based Verifying Compilers

For many applications, it is easier to **generate** security protocols
from high-level specifications than to **verify** low-level handcrafted code



1. We compile **distributed workflows** to custom crypto protocols [CSF07,09]
   - The compiler generates efficient F# code (low cryptographic overhead) with compact message formats, embedded key-establishment, etc
   - The compiler is not trusted, but also generates detailed F7 type annotations, “dumping” the invariants used for protocol synthesis
2. F7 typechecks the resulting protocol code (might report compiler bugs)

   Maybe the largest verified cryptographic protocols
   10 roles, 30 messages, 5 000 LOCs of F# code + 5 000 LOCs of F7 types

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

Summary:

Modular Verification of Protocol Code

- We develop **automated verification tools** for verifying the security of cryptographic protocol implementations
  - Against realistic threat models (crypto, active attackers)
  - Using security models closely related to executable code
  - As part of their design and development cycle
- We build **cryptographic structures with logical invariants**. We enforce them for F# code, using F7 refinement types (other verification tools may usefully apply)
- Ongoing work:
  - **Tools:** F7 v2.0 (Dec'09)
  - **Applications**: WS*, CardSpace, TLS, DKM, …
  - **Computational Soundness** of F7 typechecking under standard cryptographic assumptions
  - **Synthesis** of cryptographic protocol implementations