Verifying Cryptographic Protocols

Nikhil Swamy
CMSC 838F
April 9, 2008

Based on:
The Inductive Approach to Verifying Cryptographic Protocols, L. Paulson, JCS 1999
An Efficient Cryptographic Protocol Verifier Based on Prolog Rules, B. Blanchet, CSFW '01
Verified Interoperable Implementations of WS-Security Protocols, Bhargavan et al, CSFW '06
Verified Implementations of the Information Card Federated Identity-Management Protocol, Bhargavan et al, ASIACCS '08
Verifying Cryptographic Protocols

• New cryptographic communication protocols are still being designed

• With obvious security goals:
  • **Secrecy**: A spy cannot read the contents of a message intended for others
  • **Authenticity**: If a message appears to be from Alice, then it actually is
  • ...

• But ... protocols often go wrong
Protocol Models Vs. Protocol Implementations

- Verifying correctness of protocol models
  - A first step towards correctness
- But, models gloss over a lot of details
  - Actual message formats, gory implementation details
- So, we’d also like to be able to verify protocol implementations
  - Sometimes an implementation is the only specification
This Talk : Two approaches

- The **inductive approach** (Paulson, JCS ’99)
  - Focus on verifying protocol models
- The **logic-programming approach** (Blanchet, CSFW ’01 ...)
  - Applied to verifying implementations of web services security
    - Bhargavan et al. CSFW ’06, ASIACCS ‘08
Modeling a Protocol

• A protocol is formalized in terms of a trace of events (or messages)
  • Events are like: A sends X to B

• Given a trace T, an agent A can extend T with an event E in a manner prescribed by the protocol
  • E can depend on other events in T
  • But when receiving a message, A does not know the true identity of the sender

• One agent is the attacker, who can
  • Intercept a message, replay a message, generate new messages
The Structure of Messages

Messages are composed from:

- Agent names: $A, B, S, ...$
- Nonces: $Na, Nb, ...$
- Keys: $Ka, Kb, Kab, ...$
- Compound messages: $\{X, X'\}$
- Hashed messages: $Hash \; X$
- Encrypted messages: $Crypt \; K \; X$
The Otway-Rees Protocol (1)

Setup a session between $A$ and $B$ via an authentication server $S$

$A$ and $B$ have pre-registered symmetric keys $K_a$ and $K_b$ with $S$

1. Says $A B : \{Na, A, B, Crypt Ka \{Na, A, B\}\}$
2. Says $B S : \{Na, A, B, Crypt Ka \{Na, A, B\}, Nb, Crypt Kb \{Na, A, B\}\}$
3. Says $S B : \{Na, Crypt Ka \{Na, Kab\}, Crypt Kb \{Nb, Kab\}\}$
4. Says $B A : \{Na, Crypt Ka \{Na, Kab\}\}$
Specifying O-R Inductively (1)

A protocol is formalized as a set of traces

Says \( A \) \( B \) : \{Na, A, B, Crypt Ka \{Na, A, B\}\} # \( T \) in \( otway \)

How to make precise? We’ll come back to this

\( T \) in \( otway \)   \( A !\neq B \)   \( B !\neq S \)   \( Na \) fresh

\( T \) in \( otway \)   \( B !\neq S \)   \( Nb \) fresh  ...

(Says \( B \) \( S \) : ...) # \( T \) in \( otway \)
Why the inductive approach?

- Induction well understood; many existing tools to help with inductive reasoning
  - Isabelle, Coq etc. all come with specialized induction tactics
- Induction works well with nondeterminism
  - Concise specification
- Case-based reasoning can be lengthy, but modular
- Can handle infinite state spaces
- ...?
Modeling the Attacker

• Attacker can intercept any message
  • Easy to model in this inductive setting via nondeterminism

• A protocol is a set $T$ of traces derived according to the inductive rules
  • $T$ will include traces where $A$ sends a message to $B$
  • But this message can be ignored
    • i.e., Ignoring a message in a derivation of trace corresponds to the message being intercepted by the attacker
Modeling the Attacker

• Attacker can also inject arbitrary messages into a trace
  • Need to model attacker knowledge to define the set of messages that can be injected

• The function `synth` defines the set of messages that can be constructed from other message fragments observed by the attacker

• The function `analz` defines the set of fragments an attacker can derive by analyzing the network traffic using keys that are known to the attacker
Constructing messages: **synth**

Given a set $H$ of known message fragments

Attacker can include constants in a message

- Agent $A \in \text{synth} \ H$
- Number $N \in \text{synth} \ H$
- $X \in H \implies X \in \text{synth} \ H$
- $X \in \text{synth} \ H \implies \text{Hash} \ X \in \text{synth} \ H$
- $X \in \text{synth} \ H \implies \{X, Y\} \in \text{synth} \ H$
- $X \in \text{synth} \ H \implies K \in H \implies \text{Crypt} \ K X \in \text{synth} \ H$

Can hash arbitrary fragments

Can encrypt with known keys
Attacker’s analysis: \textbf{analz}

\begin{align*}
H \text{ is a set of messages} & \quad \frac{X \in H}{X \in \text{analz} H} \\
\text{From a message Crypt} K X, \text{ attacker knows} & \quad \frac{\text{Crypt } K X \in \text{analz } H}{X \in \text{analz } H} \\
X \text{ only if he knows the decryption key. } & \quad \frac{K^{-1} \in \text{analz } H}{X \in \text{analz } H} \\
\text{For a symmetric key } K^{-1} = K & \quad \frac{\{X, Y\} \in \text{analz } H}{X \in \text{analz } H} \\
\text{and } & \quad \frac{\{X, Y\} \in \text{analz } H}{Y \in \text{analz } H}
\end{align*}
Agents’ Knowledge

\[
\begin{align*}
\text{initState } S & \overset{\text{def}}{=} \text{all long-term keys} \\
\text{initState}(\text{Friend } i) & \overset{\text{def}}{=} \{\text{Key}(\text{shrK}(\text{Friend } i))\} \\
\text{initState Spy} & \overset{\text{def}}{=} \{\text{Key}(\text{shrK}(A)) \mid A \in \text{bad}\} \\
\text{spies } [] & \overset{\text{def}}{=} \text{initState Spy} \\
\text{spies } ((\text{Says } A B X) \# evs) & \overset{\text{def}}{=} \{X\} \cup \text{spies } evs \\
\text{spies } ((\text{Notes } A X) \# evs) & \overset{\text{def}}{=} \begin{cases} 
\{X\} \cup \text{spies } evs & \text{if } A \in \text{bad} \\
\text{spies } evs & \text{otherwise}
\end{cases}
\end{align*}
\]
Freshness

\[
\text{used } [] = \bigcup B. \text{parts}(\text{initState } B) \\
\text{used } ((\text{Says } A B X) \# evs) = \text{parts}\{X\} \cup \text{used } evs \\
\text{used } ((\text{Notes } A X) \# evs) = \text{parts}\{X\} \cup \text{used } evs
\]

Freshness will be wrt a set of previously used constants that have appeared in the trace
Otway-Rees for Real (in Isabelle)

Nil  [] ∈ otway

Fake  [| evs ∈ otway;  B ≠ Spy;  X ∈ synth (analz (spies evs)) |]
    ⇒  Says Spy B X  # evs ∈ otway

OR1  [| evs1 ∈ otway;  A ≠ B;  B ≠ Server;  Nonce NA ∉ used evs1 |]
    ⇒  Says A B { |Nonce NA, Agent A, Agent B,  
                  Crypt (shrK A) { |Nonce NA, Agent A, Agent B| } |}
        # evs1 ∈ otway

OR2  [| evs2 ∈ otway;  B ≠ Server;  Nonce NB ∉ used evs2;  
       Says A' B { |Nonce NA, Agent A, Agent B, X| } ∈ set evs2 |]
    ⇒  Says B Server  
        { |Nonce NA, Agent A, Agent B, X, Nonce NB,  
            Crypt (shrK B) { |Nonce NA, Agent A, Agent B| } |}
        # evs2 ∈ otway
Otway-Rees for Real (in Isabelle)

OR3 [\evs \in \text{otway}; \; B \neq \text{Server}; \; \text{Key} \; K_{AB} \notin \text{used evs3};
\text{Says B' Server}
\{\|\text{Nonce NA, Agent A, Agent B,}
\text{Crypt (shrK A)}\{\|\text{Nonce NA, Agent A, Agent B}|\},
\text{Nonce NB,}
\text{Crypt (shrK B)}\{\|\text{Nonce NA, Agent A, Agent B}|\}|\}
\in \text{set evs3} |]
\implies \text{Says Server B}
\{\|\text{Nonce NA,}
\text{Crypt (shrK A)}\{\|\text{Nonce NA, Key K}_{AB}|\},
\text{Crypt (shrK B)}\{\|\text{Nonce NB, Key K}_{AB}|\}|\}
# \evs \in \text{otway}
Otway-Rees for Real (in Isabelle)

OR4 [\mid evs4 \in \text{otway}; A \neq B; \\
\text{Says B Server } |\text{Nonce NA, Agent A, Agent B, X', Nonce NB,} \\
\text{Crypt (shrK B) } |\text{Nonce NA, Agent A, Agent B}| |] \\
\in \text{set evs4}; \\
\text{Says } S' \text{ B } |\text{Nonce NA, X, Crypt (shrK B) } |\text{Nonce NB, Key K}| |] \\
\in \text{set evs4 } |] \\
\implies \text{Says B A } |\text{Nonce NA, X}| \# evs4 \in \text{otway}

Oops [\mid evso \in \text{otway}; B \neq Spy; \\
\text{Says Server B } |\text{Nonce NA, X, Crypt (shrK B) } |\text{Nonce NB, Key K}| |] \\
\in \text{set evso } |] \\
\implies \text{Notes Spy } |\text{Nonce NA, Nonce NB, Key K}| \# evso \in \text{otway}
Session Key Secrecy

- Let $evs$ in $otway$ and $A, B$, not in $bad$
- Suppose $S$ issues a key $Kab$ to $A$ and $B$
  - Says $S$ $B : \{Na, Crypt Ka \{Na, Kab\}, Crypt Kb \{Nb, Kab\}\}$ in $evs$
  - And $Kab$ is not dropped by an Oops event
  - Notes Spy $\{Na, Nb, Kab\}$ not in $evs$
- Then: $Kab$ not in $(analz (spies evs))$
Authenticity?

1. $A \rightarrow C_B : Na, A, B, \{Na, A, B\}_{Ka}$
1'. $C \rightarrow A : Nc, C, A, \{Nc, C, A\}_{Kc}$
2'. $A \rightarrow C_S : Nc, C, A, \{Nc, C, A\}_{Kc}, Na', \{Nc, C, A\}_{Ka}$
2''. $C_A \rightarrow S : Nc, C, A, \{Nc, C, A\}_{Kc}, Na, \{Nc, C, A\}_{Ka}$
3'. $S \rightarrow C_A : Nc, \{Nc, Kca\}_{Kc}, \{Na, Kca\}_{Ka}$
4. $C_B \rightarrow A : Na, \{Na, Kca\}_{Ka}$
Fixing Otway-Rees

1. \( A \rightarrow B : Na, A, B, \{Na, A, B\}_{Ka} \)
2. \( B \rightarrow S : Na, A, B, \{Na, A, B\}_{Ka}, \{Na, Nb, A, B\}_{Kb} \)
3. \( S \rightarrow B : Na, \{Na, Kab\}_{Ka}, \{Nb, Kab\}_{Kb} \)
4. \( B \rightarrow A : Na, \{Na, Kab\}_{Ka} \)
Authenticity Theorem

- Let evs in \textbf{otway} and B not in \textbf{bad}
- Suppose S’ issues a key Kab to B
  - Says S’ B : \{Na, X, Crypt Kb \{Nb, Kab\}\} in evs
- Then B has requested a key from S
  - Says B S : \{Na, A, B, X’, Crypt Kb \{Na, Nb, A, B\}\} in evs
Verified Implementations of Web Services Security

Karthik Bhargavan, Cédric Fournet, Andy Gordon,
Nikhil Swamy, Stephen Tse

CSFW ’06, ASIACCS ’08
Web Services Security

• What is a web service?
  - Common case: RPC over HTTP using XML
  - Other message patterns and transports are possible

• Security intended to establish the:
  1. identity of the service requester and provider
  2. authenticity and correlation of messages
  3. secrecy of messages
(Also to preserve privacy in the context of CardSpace)
General Model
(from WS-Security)

A service publishes a policy stating

1. The kind of security token it expects from a requestor
2. Where the requestor can obtain such a token (STS)

Requestor approaches the STS to obtain a token

1. STS issues a token to the requestor
2. STS discloses the issued token to the service

Requestor then interacts with the service
Specs vs. Models vs. Implementations

- Protocol specifications (standards) are not models
  - Focus on message formats
  - Meant to guide implementations toward interoperability
- Formal 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)
- Implementations are written by developers
  - Often this is the only model you can get
- Specs, models, and implementations drift apart...
  - Even informal synchronization involves painful code reviews
  - How to keep track of implementation changes?
Verifying Implementations

- Our approach: automatically extract verifiable models from interoperable implementations
  - We consider reference implementations, not (yet) production code
  - Alternatives: generate implementations from models,..

- What you verify is what you run
  - The implementation is the model

- Write your model as code
  - Developers can write it (with help)
  - Executable specification, can be debugged

- Implementations have many non-security details
  - Choose what to abstract and what to verify
One Source, Three Tasks

Symbolic Model

My protocol

My code

Authz

Other Libraries

Application

Symbolic Crypto

Concrete Crypto

Platform (CLR)

Crypto Net

Source code (modules)

Security Goals

ProVerif

fs2pv

Symbolic verification

Symbolic testing & debugging

Interoperability (via SOAP)

Some other implementation

ProVerif

fs2pv
INFOCARD PROTOCOL
InfoCard: Information Card Profile v1.0

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

Client C (Windows Cardspace)
Client Application (A) (Web Browser)
Relying Party (RP) (Web Server)
Identity Provider (IP) (Security Token Server)

Selects card and provides password

Principal identities and protocol configured by policies and card database
**Authentication Goal [A1]**

- IP authenticates U before issuing token

- If IP issues a token that contains the secret card data of U and is meant for use at RP
  - then U must have selected this card and IP, and approved its use at RP.

- Protocol Design: IP requires that all token requests be authenticated using U’s credential
  - TLS: Request contains U’s username and password
  - WS-Security: Request is XML-signed using a key generated from U’s password, or using U’s private key
Authentication Goal [A2]

- RP authenticates U’s request (through IP)

- If RP accepts a message with a token issued by IP that contains the secret card data of U
  - then U must have selected the card and IP, and approved its use at RP,
  - and IP must have issued the token for the card,
  - and U must have approved the token,
  - and C must have sent the message to RP.

- Protocol Design: RP requires that the token is authenticated by IP and that the message is authenticated using the token
  - Token is XML-signed using IP’s private key
  - Message is XML-signed using a fresh symmetric key, and signature is counter-signed using issued token key
Authentication Goal [A3]

• C authenticates RP’s response

• If C accepts a message from RP
  – then RP must have sent this message in response to C’s request message.

• Protocol Design: C requires that the response message is authenticated by RP and correlated with the request
  – Message is XML–signed using the same symmetric key as request
  – Optionally, the signature value of the request is
Secrecy Goal [S1]

• U’s data is released only to RPs chosen by U

• If the attacker obtains the secret card data of U
  – then U must have selected the card and IP, and approved its use at a compromised RP,
  – and IP must have issued a token for the card,
  – and U must have approved the token.

• Protocol Design: IP authenticates U and then encrypts the token for RP
  – If token is not specialized to one RP, then the token is sent in the clear
Secrecy Goal [S2]

• RPs cannot reconstruct U’s browsing history

• Two colluding RP’s cannot correlate use of a card
  – A protocol run where U presents the same card to RP and RP’ is observationally equivalent to one where U presents different cards to them, even if RP and RP’ are compromised.

• Protocol Design: IP computes a pseudonym for U and inserts it into each issued token
  – the pseudonym is specialized to the receiving RP
  – two RPs get tokens with different pseudonyms.
Secrecy Goal [S3]

- IP only knows U’s browsing history if U tells it

- The IP cannot discover which RP the user U is interacting with, unless U requests a token with limited scope

- Protocol Design: The token request contains no information about RP if the token scope is unlimited
  - C computes the pseudonym in this case and sends
Protocol Narration (Self-Issued Card)

Initially C has: Card\([\text{cardId, claims}_U, PK(k_{RP})]\) \(\text{RP has: } k_{RP}\)

| C: | Request \((RP, M_{req})\) |
| U: | Select InfoCard \((\text{cardId}, C, RP, \text{claim-ty}_R)\) |
| C(\(IP\)): | Issue Token \((U, \text{cardId}, \text{claims}_U, RP, \text{display-tok})\) |
| U: | U : Approve Token \((\text{display-tok})\) |

\(C\rightarrow RP:\)  
let \(M_{ek} = \text{RSAEnc}(PK(k_{RP}), k)\) in  
let \(k_{sig} = \text{PSHA1}(k, \eta_1)\) in  
let \(k_{enc} = \text{PSHA1}(k, \eta_2)\) in  
let \(ppid_{\text{cardId},RP} = H_4(\text{cardId}, RP)\) in  
let \(k_{\text{cardId},RP} = K(\text{cardId}, RP)\) in  
let \(M_{tok} = \text{Assertion}(\text{Self}, PK(k_{proof}), \text{claims}_U, RP, ppid_{\text{cardId},RP})\) in  
let \(M_{toksig} = \text{RSASHA1}(k_{\text{cardId},RP}, M_{tok})\) in  
let \(M_{\text{saml}} = \text{SAML}(M_{tok}, M_{toksig})\) in  
let \(M_{\text{mac}} = \text{HMACSHA1}(k_{sig}, M_{req})\) in  
let \(M_{proof} = \text{RSASHA1}(k_{proof}, M_{mac})\) in  
Service Request \((M_{ek}, \eta_1, \eta_2, PK(k_{\text{cardId},RP}),\) \(\text{AESEnc}(k_{enc}, M_{saml}), \text{AESEnc}(k_{enc}, M_{mac}),\) \(\text{AESEnc}(k_{enc}, M_{proof})\), AESEnc\((k_{enc}, M_{req})\))

| RP: | Accept Request \((C, \text{claims}_U, M_{req}, M_{resp})\) |
| RP: | generate fresh \(\eta_3, \eta_4\) |
| RP \(\rightarrow C:\) | let \(k_{sig} = \text{PSHA1}(k, \eta_3)\) in  
| | let \(k_{enc} = \text{PSHA1}(k, \eta_4)\) in  
| | let \(M_{\text{mac}} = \text{HMACSHA1}(k_{sig}, M_{resp})\) in  
| | Service Response \((\eta_3, \eta_4, \text{AESEnc}(k_{enc}, M_{mac}), \text{AESEnc}(k_{enc}, M_{resp}))\) |

\(C:\)  
Response \((M_{resp})\)

- \(C\) receives an application request
- \(U\) selects card
- \(C\) generates a self-issued token
- \(U\) approves token
- \(C\) has fresh session key, two nonces, and asymmetric key-pairs
- Encrypt session key for \(RP\)
- Derive message signing key
- Derive message encryption key
- Compute PPID using card identifier, \(RP\)'s identity
- Compute token signing key using card, \(RP\)'s identity
- SAML assertion with public key, claims, and PPID
- Self-signed SAML assertion
- Issued token
- Message signature
- Endorsing signature proving possession of \(k_{proof}\)
- Request, with encrypted token, signatures, and body

- \(RP\) accepts request and authorizes a response
- \(\)Fresh nonces
- Derive message signing key
- Derive message encryption key
- Message Signature
- Service Response, with encrypted signatures and body
- \(C\) accepts response and sends it to application
<table>
<thead>
<tr>
<th>Initially</th>
<th>$C$ has: cardId, $PK(k_iP)$, $PK(k_RP)$</th>
<th>$IP$ has: $k_iP$, $PK(k_RP)$, Card(cardId, claimsU, pwdU, $IP_k_iP$)</th>
<th>$RP$ has: $k_RP$</th>
<th>$PK(k_iP)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$C$ :</td>
<td>Request ($RP$, $m_{req}$)</td>
<td>$C$ receives an application request</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$U$ :</td>
<td>Select InfoCard(cardId, $C$, $RP$, pwdU, $IP$, claim-tyRp)</td>
<td>User selects card and provides password</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$C$ → $IP$ :</td>
<td>generate fresh $k_1, \eta_1, \eta_2, \eta_c$</td>
<td>Fresh session key, two nonces, and client entropy for token key</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>let $M_{ok} = RSAEnc(PK(k_iP), k_1)$ in</td>
<td>Encrypt session key for $IP$</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>let $kSig = PSHA1(k_1, \eta_1)$ in</td>
<td>Derive message signing key</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>let $kEnc = PSHA1(k_1, \eta_2)$ in</td>
<td>Derive message encryption key</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>let $M_{rg} = RST(cardId, claim-tyRp, $RP$, \eta_c)$ in</td>
<td>Token request message body</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>let $M_{user} = (U, pwdU)$ in</td>
<td>User authentication token</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>let $M_{mac} = HMACSHA1(k_{Sig}, (M_{rg}, M_{user})$) in</td>
<td>Message signature</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Request Token ($M_{ok}, \eta_1, \eta_2$,</td>
<td>Token Request, with encrypted signatures, token and body</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>AESEnc(k_{Enc}, M_{mac}), AESEnc(k_{Enc}, M_{user}),</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>AESEnc(k_{Enc}, M_{rg})$)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$IP$ :</td>
<td>Issue Token (cardId, claimsU, $RP$, display-tok)</td>
<td>$IP$ issues token for $U$ to use at $RP$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$IP$ :</td>
<td>generate fresh $\eta_3, \eta_4, \eta_{se}, k_2$</td>
<td>Fresh nonces, server entropy, token encryption key</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$IP$ :</td>
<td>let $kSig = PSHA1(k_1, \eta_3)$ in</td>
<td>Derive message signing key</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$IP$ :</td>
<td>let $kEnc = PSHA1(k_1, \eta_4)$ in</td>
<td>Derive message encryption key</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$IP$ :</td>
<td>let $M_{tokkey} = RSAEnc(PK(k_iP), PSHA1(\eta_{se}, \eta_{se})$) in</td>
<td>Compute token key from entropies, encrypt for $RP$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$IP$ :</td>
<td>let $ppid_{cardId,RP} = H_1(k_{cardId,RP})$ in</td>
<td>Compute PPID using card master key,RP's identity</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$IP$ :</td>
<td>let $M_{tok} = Assertion(IP, M_{tokkey}, claimsU, $RP$, ppid_{cardId,RP})$ in</td>
<td>SAML assertion with token key, claims, and PPID</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$IP$ :</td>
<td>let $M_{toksig} = RSAEnc(M_{tok}$, $M_{tok}$) in</td>
<td>SAML assertion signed by issuer</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$IP$ :</td>
<td>let $M_{ek} = RSAEnc(PK(k_iP), k_2)$ in</td>
<td>Token encryption key, encrypted for $RP$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$IP$ :</td>
<td>let $M_{entok} = (M_{ek}, AESEnc(k_r, SAML(M_{tok}, M_{toksig})$) in</td>
<td>Encrypted issued token</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$IP$ :</td>
<td>let $M_{rst} = RST(M_{entok}, \eta_{sr})$ in</td>
<td>Token response message body</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$IP$ :</td>
<td>let $M_{mac} = HMACSHA1(k_{Sig}, M_{rst})$ in</td>
<td>Message Signature</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$IP$ :</td>
<td>Token Response ($\eta_3, \eta_4$, AESEnc(k_{Enc}, M_{mac}), AESEnc(k_{Enc}, M_{rst})$)</td>
<td>Token Response, with encrypted signature and body</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$U$ :</td>
<td>Approve Token (display-tok)</td>
<td>User approves token</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$C$ :</td>
<td>generate fresh $k_2, \eta_5, \eta_6, \eta_7$</td>
<td>Fresh session key, three nonces</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$C$ → $RP$ :</td>
<td>let $M_{ek} = RSAEnc(PK(k_iP), k_2)$ in</td>
<td>Encrypt session key for $RP$</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>let $kSig = PSHA1(k_2, \eta_5)$ in</td>
<td>Derive message signing key</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>let $kEnc = PSHA1(k_2, \eta_6)$ in</td>
<td>Derive message encryption key</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>let $kProof = PSHA1(\eta_{se}, \eta_{se})$ in</td>
<td>Compute token key from entropies</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>let $M_{mac} = HMACSHA1(k_{Sig}, M_{req})$ in</td>
<td>Message signature</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>let $kEndorse = PSHA1(k_{proof}, \eta_7)$ in</td>
<td>Derive a signing key from the issued token key</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>let $M_{proof} = HMACSHA1(k_{Endorse}, M_{mac})$ in</td>
<td>Endorsing signature proving possession of token key</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Service Request($M_{ek}, \eta_5, \eta_6, \eta_7, M_{entok}$, AESEnc(k_{Enc}, M_{mac}), AESEnc(k_{Enc}, M_{proof}), AESEnc(k_{Enc}, M_{req}))</td>
<td>Service Request, with issued token, encrypted signatures and body</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$RP$ :</td>
<td>Accept Request ($IP$, claimsU, $M_{req}$, $M_{resp}$)</td>
<td>$RP$ accepts request and authorizes a response</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$RP$ :</td>
<td>generate fresh $\eta_8, \eta_9$</td>
<td>Fresh nonces</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$RP$ :</td>
<td>let $kSig = PSHA1(k_2, \eta_8)$ in</td>
<td>Derive message signing key</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$RP$ :</td>
<td>let $kEnc = PSHA1(k_2, \eta_9)$ in</td>
<td>Derive message encryption key</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$RP$ :</td>
<td>let $M_{mac} = HMACSHA1(k_{Sig}, M_{resp})$ in</td>
<td>Message signature</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$RP$ :</td>
<td>Service Response ($\eta_8, \eta_9$, AESEnc(k_{Enc}, M_{mac}), AESEnc(k_{Enc}, M_{resp})$)</td>
<td>Service Response, with encrypted signatures and body</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$C$ :</td>
<td>Response ($M_{resp}$)</td>
<td>$C$ accepts response and sends it to application</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Wssec.fsi (interface)

**type** bytes
**val** payload2body : body → body payload

**type** α payload
**val** body2payload : body payload → body

**type** α enc
**val** aes_encrypt : symkey → α payload → α enc

**type** α dsig
**val** aes_decrypt : symkey → α enc → α payload

**type** envelope
**val** rsa_sign : privkey → α payload → α dsig
**val** rsa_verify : pubkey → α payload → α dsig → unit

...
1. Replace core libraries and platform with modules implementing a symbolic Dolev–Yao Abstraction

2. `fs2pv`

3. Verify

Fails security goals, here’s an attack!

Provably satisfies security goals
Formalizing a subset of F#

\[ M, N ::= \]
\[ x \quad \text{variable} \]
\[ a \quad \text{name} \]
\[ f(M_1, \ldots, M_n) \quad \text{constructor application} \]

\[ e ::= \]
\[ M \quad \text{value} \]
\[ \ell M_1 \ldots M_n \quad \text{function application} \]
\[ \text{fork}(\text{fun}() \to e) \quad \text{fork a parallel thread} \]
\[ \text{match } M \text{ with } (| M_i \to e_i | i \in 1..n) \quad \text{match: } M_i \text{ patterns, } n \geq 0 \]
\[ \text{let } x = e_1 \text{ in } e_2 \quad \text{sequential evaluation} \]

\[ d ::= \]
\[ \text{type } s = (| f_i \text{ of } s_{i1} \times \cdots \times s_{im_i} | i \in 1..n) \quad \text{datatype declaration} \]
\[ \text{let } x = e \quad \text{value declaration} \]
\[ \text{let } \ell x_1 \ldots x_n = e \quad \text{function declaration} \quad n > 0 \]

\[ S ::= d_1 \cdots d_n \quad \text{system: list of declarations} \]
Compiling F# to Pi Calculus

Consider the F# function

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

We can translate it as a process

```plaintext
\text{lin}(\text{mac}, (\text{nonce},\text{pwd},\text{text},k));
\text{out}(k,\text{Hmacsha1}(\text{nonce},\text{Concat(Utf8(pwd),Utf8(text)})))
```

We actually translate \text{mac} into a ProVerif reduction:

```fsharp
\text{reduc} \text{mac(nonce,pwd,text)} =
    \text{HmacSha1(nonce,Concat(Utf8(pwd),Utf8(text))})
```
What do we prove?

- Let \( L \) be a set of modules representing the symbolic libraries for cryptography, networking
- Let \( P \) be the protocol implementation
- Let \( I \) be the interface exported by \( L \) and \( P \)
  - We write \( L \ P :: I \)
- Let \( q \) be a desired security property
  - Authentication or secrecy property written as a correspondence between events in a trace
- Then, for all opponent programs \( O \) that respect \( I \), every trace of \( L \ P \ O \) satisfies \( q \)
  - Hence, using only the values and functions in \( I \), no opponent can break the property \( q \) of our
# Safety Results

<table>
<thead>
<tr>
<th>Name</th>
<th>LOC</th>
<th>Crypto Ops</th>
<th>Auth</th>
<th>Secrecry</th>
<th>Verif Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>SelfIssued–SOAP</td>
<td>1410(80)</td>
<td>9,3</td>
<td>A1–A3</td>
<td>S1,S2</td>
<td>38s</td>
</tr>
<tr>
<td>UserPassword–TLS</td>
<td>1426(96)</td>
<td>0,5,17,6</td>
<td>A1–A3</td>
<td>S1,S2</td>
<td>24m40s</td>
</tr>
<tr>
<td>UserPassword–SOAP</td>
<td>1429(99)</td>
<td>9,11,17,6</td>
<td>A1–A3</td>
<td>S1,S2</td>
<td>20m53s</td>
</tr>
<tr>
<td>UserCertificate–SOAP</td>
<td>1429(99)</td>
<td>13,7,11,6</td>
<td>A1–A3</td>
<td>S1–S3</td>
<td>66m21s</td>
</tr>
<tr>
<td>UserCertificate–SOAP–v</td>
<td>1429(99)</td>
<td>7,5,7,4</td>
<td>A3 Fails!</td>
<td>S1–S3</td>
<td>10s</td>
</tr>
</tbody>
</table>
Vulnerabilities

- If RP’s policy does not require signatures to be encrypted, we find a man-in-the-middle attack that breaks [A3]
  - The attacker can replace U’s token with his own, and recompute the message signature
  - The protocol terminates with inconsistent states at C and RP
- A similar attack is found if IP does not require encrypted signatures
- Fix: Use strong policies requiring encrypted signatures and/or signature confirmation
Vulnerabilities

• If the token is self-issued then the pseudonym does not provide enough privacy, breaking [S2]
  – Pseudonym = Hash (cardId + RP’s X.509 identifier)
  – Unless cardId is a cryptographically random secret, two colluding RP’s can guess it, confirm its value by computing the hash, and correlate the use of the same card at different RPs

• Fix: Use a strong random cardId, keep it secret
Conclusions

• Inductive approach for verifying protocol models
  • Concise specification, relatively easy interactive proofs
  • Possibly applicable to verifying implementations interactively

• Logic programming seems to be better suited to mostly automated analysis of implementations
  • But both approaches complementary: inductive approach when designing a protocol, fs2pv for code

• But: all of this is formal symbolic analysis
  • What about a computational model of cryptography?
Questions?
Summary

• We present verification results for the InfoCard protocol underlying Windows Cardspace
• We verify reference implementations in F#
• We found vulnerabilities in some configurations
• We present strong safety theorems for some configurations
  – Our results hold in the symbolic Dolev–Yao model

• Try it out on your favourite protocol!
Sample Configurations

• Self-issued card (created by user)
  – No IP, token is issued by client (Cardspace)
  – Token associated with asymmetric key-pair
  – All messages mac-ed and encrypted (WS-Security)

• Managed card (issued by IP to user)
  – User authenticates with username-password
  – IP issues token associated with symmetric key (encrypted for RP) and a user pseudonym
  – All messages mac-ed and encrypted (WS-Security)

• Or: C–IP exchanges over TLS, C–RP over WS–
Download, Compile, Verify,

Identity Provider (IP) → Get Policy → policy2fs → Generate → Verify
- Security goals: Yes → proxy.dll (Linked)
- Security goals: No → Concrete Libraries → Stop

Relying Party (RP) → References → policy → policy2fs

Symbolic Libraries: RP.fs, C.fs, IP.fs
InfoCard Configurations

Depending on IP’s policy, RP’s policy, Card DB:

• Who issues the token?
  – Self Issued, or IP issued

• How does the user authenticate at the IP?
  – Username/Password, X.509, Kerberos, or SAML

• Is the token’s scope limited to one RP?

• What type of key is associated with the token?
  – Symmetric, or Asymmetric

• What type of security is used for the messages?
# Protocol Narration (Self Issued Card)

**Initially** \( C \text{ has: } Card(\text{cardId}, \text{claims}_U), PK(k_{RP}) \) \[ \text{RP has: } k_{RP} \]

<table>
<thead>
<tr>
<th>C:</th>
<th>Request ((R_P, M_{req}))</th>
<th>C receives an application request</th>
</tr>
</thead>
<tbody>
<tr>
<td>U:</td>
<td>Select InfoCard ((\text{cardId}, C, R_P, \text{claim-ty}_R))</td>
<td>User selects card</td>
</tr>
<tr>
<td>C(IP):</td>
<td>Issue Token ((U, \text{cardId}, \text{claims}_U, R_P, \text{display-tok}))</td>
<td>C generates a self-issued token</td>
</tr>
<tr>
<td>U:</td>
<td>Approve Token ((\text{display-tok}))</td>
<td>User approves token</td>
</tr>
</tbody>
</table>

**C** → **R_P**:
- Let \( M_{ek} = \text{RSAEnc}(PK(k_{RP}), k) \) in
- Let \( k_{sig} = \text{PSHA1}(k, \eta_1) \) in
- Let \( k_{enc} = \text{PSHA1}(k, \eta_2) \) in
- Let \( ppid_{\text{cardId}, R_P} = H_4(\text{cardId}, R_P) \) in
- Let \( k_{\text{cardId}, R_P} = K(\text{cardId}, R_P) \) in
- Let \( M_{tok} = \text{Assertion}(\text{Self}, PK(k_{proof}), \text{claims}_U, R_P, ppid_{\text{cardId}, R_P}) \) in
- Let \( M_{toksig} = \text{RSASHA1}(k_{\text{cardId}, R_P}, M_{tok}) \) in
- Let \( M_{saml} = \text{SAML}(M_{tok}, M_{toksig}) \) in
- Let \( M_{mac} = \text{HMACSHA1}(k_{sig}, M_{req}) \) in
- Let \( M_{proof} = \text{RSASHA1}(k_{proof}, M_{mac}) \) in
- Service Request \((M_{ek}, \eta_1, \eta_2, PK(k_{\text{cardId}, R_P}), \text{AESEnc}(k_{enc}, M_{saml}), \text{AESEnc}(k_{enc}, M_{mac}), \text{AESEnc}(k_{enc}, M_{proof}), \text{AESEnc}(k_{enc}, M_{req}))\)

**R_P**: Accept Request \((C, \text{claims}_U, M_{req}, M_{resp})\)
- RP accepts request and authorizes a response
- Fresh nonces

**R_P** → **C**:
- Let \( k_{sig} = \text{PSHA1}(k, \eta_3) \) in
- Let \( k_{enc} = \text{PSHA1}(k, \eta_4) \) in
- Let \( M_{mac} = \text{HMACSHA1}(k_{sig}, M_{resp}) \) in
- Service Response \((\eta_3, \eta_4, \text{AESEnc}(k_{enc}, M_{mac}), \text{AESEnc}(k_{enc}, M_{resp}))\)

**C**: Response \((M_{resp})\)
- C accepts response and sends it to application
Soundness of our compiler

Theorem 1 (Reflection of Robust Safety)

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

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

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

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