An Executable Model for JFKr

An ACL2 approach to key-establishment protocol verification

Presented by: David Rager
February 1, 2006

Outline

- Derivation of JFKr
- Books developed for JFKr reasoning
- Demonstrate the JFKr executable model
- Presentation of properties
  - Identity
  - Session Key
- Wrap up

Design Objectives for a Key Exchange Protocol

- Shared secret
  - Create and agree on a secret which is known only to protocol participants
- Authentication
  - Participants need to verify each other’s identity
- Identity protection
  - Eavesdropper should not be able to infer participants’ identities by observing protocol execution
- Protection against denial of service
  - Malicious participant should not be able to exploit the protocol to cause the other party to waste resources
- Protection against replay attack
  - Malicious participant should not be able to reuse old data

Ingredient 1: Diffie-Hellman

$$A \rightarrow B: \quad g^a$$

$$B \rightarrow A: \quad g^b$$

- Shared secret: g^{ab}
  - Diffie-Hellman guarantees perfect forward secrecy
- Authentication
- Identity protection
- DoS protection

Ingredient 2: Challenge-Response

$$A \rightarrow B:\ m, A$$
$$B \rightarrow A:\ n, sig_B\{m, n, A\}$$
$$A \rightarrow B:\ sig_A\{m, n, B\}$$

Shared secret

- ☐ Authentication
  - A receives his own number m signed by B's private key and deduces that B is on the other end; similar for B
- ☐ Identity protection
- ☐ DoS protection

DH + Challenge-Response

ISO 9798-3 protocol:

$m := g^a$

$n := g^b$

$A \rightarrow B$: g^a, A

$B \rightarrow A$: g^b, $\mathrm{sig}_B\{g^a, g^b, A\}$

$A \rightarrow B$: $\mathrm{sig}_A\{g^a, g^b, B\}$

- Shared secret: g^{ab}
- Authentication
- Identity protection
- DoS protection

Ingredient 3: Encryption

Encrypt signatures to protect identities:

$A \rightarrow B$: g^a, A

$B \rightarrow A$: g^b, $E_K\{sig_B\{g^a, g^b, A\}\}$

$A \rightarrow B$: $E_K\{sig_A\{g^a, g^b, B\}\}$

- Shared secret: g^{ab}
- Authentication
- Identity protection (for responder only!)
- DoS protection

Anti-DoS Cookie

- Typical protocol:
  - Client sends request (message #1) to server
  - Server sets up connection, responds with message #2
  - Client may complete session or not (potential DoS)
- Cookie version:
  - Client sends request to server
  - Server sends hashed connection data back
    - Send message #2 later, after client confirms
  - Client confirms by returning hashed data
  - Need extra step to send postponed message

Ingredient 4: Anti-DoS Cookie

“Almost-JFK” protocol:

$A \rightarrow B$: g^a, A

$B \rightarrow A$: g^b, $\text{hash}_{Kb}\{g^b, g^a\}$

$A \rightarrow B$: g^a, g^b, $\text{hash}_{Kb}\{g^b, g^a\}$

$E_K\{\text{sig}_A\{g^a, g^b, B\}\}$

$B \rightarrow A$: g^b, $E_K\{\text{sig}_B\{g^a, g^b, A\}\}$

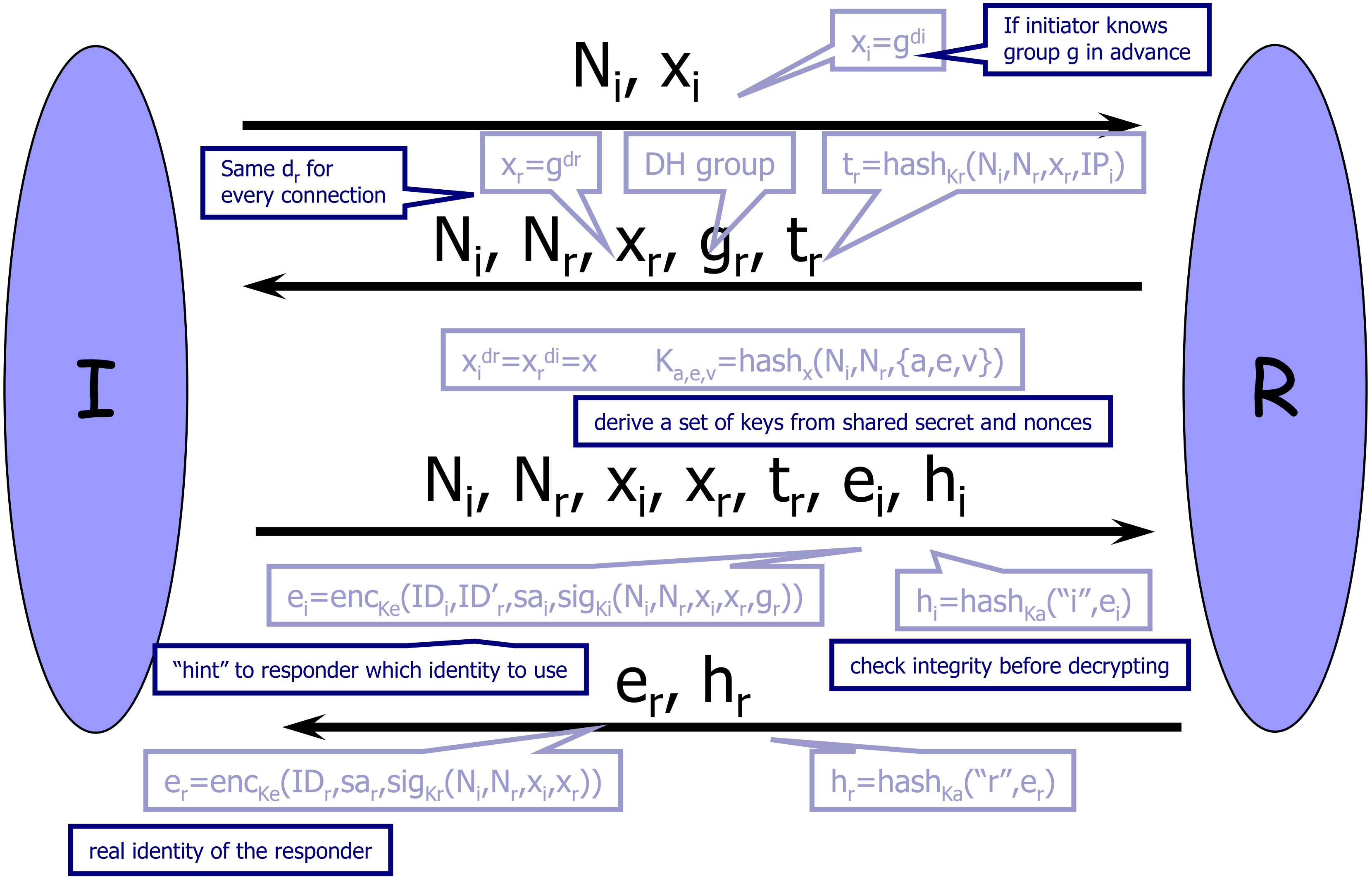
Doesn’t quite work: B must remember his DH exponential b for every connection

- Shared secret: g^{ab}
- Authentication
- Identity protection
- DoS protection?

Additional Features of JFK

- Keep g^a, g^b values medium-term, use (g^a,nonce)
  - Use same Diffie-Hellman value for every connection (helps against DoS), update every 10 minutes or so
  - Nonce guarantees freshness
  - More efficient, because computing g^a, g^b, g^{ab} is costly
- Two variants: JFKr and JFKi
  - JFKr protects identity of responder against active attacks and of initiator against passive attacks
  - JFKi protects only initiator's identity from active attack

JFKr



N_i, x_i

$x_i=g^{d_i}$

If initiator knows group g in advance

Same d_r for every connection

$x_r=g^{d_r}$

DH group

$t_r=\text{hash}_{K_r}(N_i,N_r,x_r,IP_i)$

N_i, N_r, x_r, g_r, t_r

$x_i^{d_r}=x_r^{d_i}=x$ $K_{a,e,v}=\text{hash}_x(N_i,N_r,\{a,e,v\})$

derive a set of keys from shared secret and nonces

$N_i, N_r, x_i, x_r, t_r, e_i, h_i$

$e_i=\text{enc}_{K_e}(ID_i,ID'_r,sa_i,sig_{K_i}(N_i,N_r,x_i,x_r,g_r))$

$h_i=\text{hash}_{K_a}(\text{"i"},e_i)$

"hint" to responder which identity to use

check integrity before decrypting

e_r, h_r

$e_r=\text{enc}_{K_e}(ID_r,sa_r,sig_{K_r}(N_i,N_r,x_i,x_r))$

$h_r=\text{hash}_{K_a}(\text{"r"},e_r)$

real identity of the responder

[Aiello et al.] and Shmatikov

Executing the Model

```
(defmacro run-5-steps-honest (network-s initiator-constants responder-constants
                                   public-constants initiator-s responder-s)

  `(mv-let
   (network-s-after-1 initiator-s-after-1)
   (initiator-step1 ,network-s ,initiator-s ,initiator-constants ,public-constants)

   (mv-let
    (network-s-after-2 responder-s-after-2)
    (responder-step1 network-s-after-1 ,responder-s ,responder-constants ,public-constants)

   (mv-let
    (network-s-after-3 initiator-s-after-3)
    (initiator-step2 network-s-after-2 initiator-s-after-1 ,initiator-constants ,public-constants)

   (mv-let
    (network-s-after-4 responder-s-after-4)
    (responder-step2 network-s-after-3 responder-s-after-2 ,responder-constants ,public-constants)

   (mv-let
    (network-s-after-5 initiator-s-after-5)
    (initiator-step3 network-s-after-4 initiator-s-after-3 ,initiator-constants ,public-constants)

    (mv network-s-after-5
        initiator-s-after-5
        responder-s-after-4)))))))
```

An Example Execution

```
;;; The below theorem illustrates an example of what a successful trace of the
;;; JFKr protocol looks like

(thm (mv-let (network-s initiator-s responder-s)
             (run-5-steps-honest nil
                                 *initiator-constant-list*
                                 *responder-constant-list*
                                 *public-constant-list*
                                 nil
                                 nil)
             (declare (ignore network-s))
             (and

              ;; responder stores the correct partner
              (equal (id *initiator-constant-list*)
                     (id-i responder-s))

              ;; initiator stores the correct partner
              (equal (id *responder-constant-list*)
                     (id-r initiator-s))

              ;; responder and initiator have the same session key
              (equal (session-key initiator-s)
                     (session-key responder-s)))))
```

Executable Model Demonstration

Notes:

1. Ld “jfkr.lisp”
2. Run-5-steps-honest with constants
    1. Notice both parties complete
    2. Same key
    3. Identities match up

Prerequisites to the Model

- Encryption book – we need:
  - Functions that do primitive hash/encrypt/signature operations
  - To prove that decrypting an encryption requires the key
  - To prove that duplicating a hash of something requires the key
  - To prove that verifying a signature requires the public key
  - To prove that creating a signature that can be verified with a public key requires the private key
  - To then disable the definitions of the hash/encrypt/signature functions, because we now have abstraction and no longer want to reason about the functions themselves.

Prerequisites to the Model

- Encryption book – we need symmetric encryption

```
(defun encrypt-symmetric-list (lst key)
  (if (atom lst)
      nil
    (cons (+ (car lst) key)
          (encrypt-symmetric-list (cdr lst) key))))

(defun decrypt-symmetric-list (lst key)
  (if (atom lst)
      nil
    (cons (- (car lst) key)
          (decrypt-symmetric-list (cdr lst) key))))
```

Prerequisites to the Model

- Encryption book – we need symmetric encryption

```
(defthm decrypt-of-encrypt-symmetric-equals-plaintext
  (implies (force (encryptable-listp lst))
   (equal (decrypt-symmetric-list (encrypt-symmetric-list lst key)
                                  key)
          lst)))

(defthm decrypt-of-encrypt-symmetric-needs-key
  (implies (and (encryptable-listp lst)
                (not (null lst))
                (keyp keyA)
                (keyp keyB)
                (not (equal keyA keyB)))
           (not (equal (decrypt-symmetric-list (encrypt-symmetric-list lst keyA)
                                               keyB)
                       lst))))
```

Prerequisites to the Model

- Encryption book – we need:
  - A similar model for asymmetric encryption and signature creation/verification
  - To then disable the definitions of the hash/encrypt/signature functions, because we now have abstraction and no longer want to reason about the functions themselves. So crucial to keep ACL2 from blowing up.

Prerequisites to the Model

- Diffie Helman book – we need:
  - ☐ A theorem that states that if each party derives the key using their own private value and the other party's public-DH-value, then the keys are equal
  - ☐ A way to state that either the x-exponent or y-exponent is necessary to derive the key.
    - Can probably exploit this to prove nil

Prerequesites to the Model

- Diffie Helman book – we need key equality

```
(defun compute-public-dh-value (g exponent-value b)
  (mod (expt g exponent-value) b))

(defun compute-dh-key (a-public-exponentiation a-private-value b)
  (mod (expt a-public-exponentiation a-private-value) b))


(defthm dh-computation-works
  (implies (and (integerp g)
                (<= 1 g)
                (integerp b)
                (<= 1 b)
                (integerp x-exponent)
                (<= 1 x-exponent)
                (integerp y-exponent)
                (<= 1 y-exponent))
           (equal (compute-dh-key (compute-public-dh-value g x-exponent b)
                                  y-exponent
                                  b)
                  (compute-dh-key (compute-public-dh-value g y-exponent b)
                                  x-exponent
                                  b)))))
```

Prerequesites to the Model

- Diffie Helman book – we need key secrecy

```
(defun session-key-requires-one-part-of-key
  (g b x-exponent y-exponent i-exponent)
  ;; we set the guards to nil to ensure that this function never executes and
  ;; is only used in the logical reasoning of the proof
  (declare (xargs :guard nil
                  :verify-guards nil))

  (implies (and (force (integerp g)) #| etc. |#
                (not (equal i-exponent x-exponent))
                (not (equal i-exponent y-exponent)))

           (let ((x-public-value (compute-public-dh-value g x-exponent b))
                 (y-public-value (compute-public-dh-value g y-exponent b))
                 (session-key
                  (compute-dh-key (compute-public-dh-value g x-exponent b)
                                  y-exponent
                                  b)))

             (and (not (equal (compute-dh-key x-public-value i-exponent b)
                              session-key))

                  (not (equal (compute-dh-key y-public-value i-exponent b)
                              session-key))))))
```

Model "Features"

- Party constants are abstract

```
(defthm run-5-steps-with-badly-forged-attacker-yields-both-failure

  (let ((initiator-constants (initiator-constants constants))
        (responder-constants (responder-constants constants))
        (public-constants (public-constants constants)))
 ; conclusion to come
)
```

Model “Features”

- Nondeterministic attacker

```
(defstub function-we-know-nothing-about1 (*) => *)

(defthm run-5-steps-with-badly-forged-attacker-yields-both-failure

  (mv-let
   (network-s-after-1 initiator-s-after-1)
   (initiator-step1 network-s initiator-s initiator-constants public-constants)


   (let ((network-s-after-1-munged (function-we-know-nothing-about1 network-s-after-1)))
     (mv-let
      (network-s-after-2 responder-s-after-2)
      (responder-step1 network-s-after-1-munged responder-s
                       responder-constants public-constants)
```

- ACL2 question – how do I hide the part inside of function-we-…?

Model “Features”

- Separation of concepts like a well-formed message versus a message that’s badly-forged

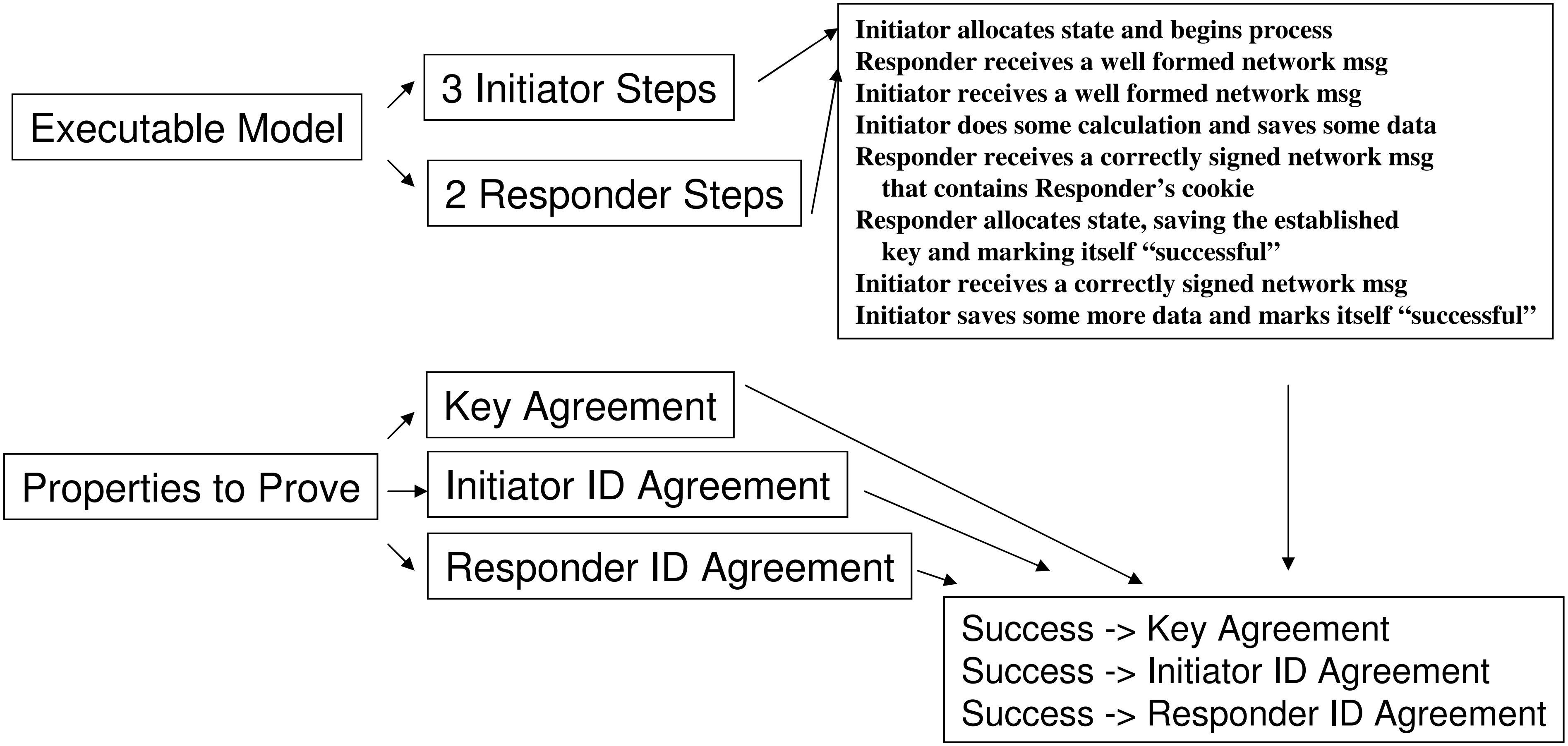
```
(defun well-formed-msg3p (msg)
  (declare (xargs :guard t))
  (and (alistp msg)
       (integerp (Ni-msg msg))
       (integerp (Nr-msg msg))
       (integerp (Xi-msg msg))
       (<= 0 (Xi-msg msg))
       (integerp (Xr-msg msg))
       (<= 0 (Xr-msg msg))
       (integerp (Tr-msg msg))
       (integer-listp (Er-msg msg))
       (integerp (Hi-msg msg))
       (integerp (Src-ip-msg msg))))
```

Model “Features”

- Separation of concepts like a well-formed message versus a message that’s badly-forged

```
(defun badly-forged-msg3p-old(msg responder-constants initiator-private-key)
  (let* ((dh-key (CRYPTO::compute-dh-key (xi-msg msg)
                                         (dh-exponent responder-constants)
                                         (b responder-constants)))
         (session-key (compute-session-key (Ni-msg msg)
                                           (Nr-msg msg)
                                          dh-key))
         (SigKi (compute-sig-Ki (Ni-msg msg)
                                (Nr-msg msg)
                                (Xi-msg msg)
                                (Xr-msg msg)
                                (g responder-constants)
                                (b responder-constants)
                               initiator-private-key))
         (Ei-decrypted (CRYPTO::decrypt-symmetric-list (Ei-msg msg) session-key)))
    (not (equal (nth 2 Ei-decrypted)
                SigKi))))
```

Game Plan



Executable Model

3 Initiator Steps

2 Responder Steps

Initiator allocates state and begins process
Responder receives a well formed network msg
Initiator receives a well formed network msg
Initiator does some calculation and saves some data
Responder receives a correctly signed network msg that contains Responder’s cookie
Responder allocates state, saving the established key and marking itself “successful”
Initiator receives a correctly signed network msg
Initiator saves some more data and marks itself “successful”

Properties to Prove

Key Agreement

Initiator ID Agreement

Responder ID Agreement

Success -> Key Agreement
Success -> Initiator ID Agreement
Success -> Responder ID Agreement

High Level Properties to Prove

- Identity Agreement
  - Wouldn’t it be lovely:

```
(implies (and (initiator-success initiator-s)
              (responder-success responder-s))
         (and (equal (id-I responder-s)
                     (id initiator-constants))
              (equal (id-r initiator-s)
                     (id responder-constants))))
```

Identity Agreement

- if they are not the id associated with a private key, then they do not have the private key
- if they do not have the private key, then they will not sign this message verifiable with the public key
- if they do not sign this message, then the protocol will not be successful

    - ☐ The last two are formalized in ACL2

Identity Agreement

- Translates by contra positive into:

- if they have the private key, then they are the id associated with that private key
- if they sign the message verifiable with the public key, then they have the private key
- if the protocol is successful, then they signed the message

Identity Agreement

- Reorders to:

- if the protocol is successful, then they signed the message
- if they sign the message verifiable with the public key, then they have the private key
- if they have the private key, then they are the id associated with that private key

Identity Agreement

- Gives us:

If the protocol is successful, then the "other" identity is the id associated with that private key

Identity Theorem

```
(defthm run-5-steps-with-badly-forged-attacker-yields-both-failure

  (let ((initiator-constants (initiator-constants constants))
        (responder-constants (responder-constants constants))
        (public-constants (public-constants constants)))

  (mv-let
   (network-s-after-1 initiator-s-after-1)
   (initiator-step1 network-s initiator-s initiator-constants public-constants)


   (let ((network-s-after-1-munged (function-we-know-nothing-about1 network-s-after-1)))
     (mv-let
      (network-s-after-2 responder-s-after-2)
      (responder-step1 network-s-after-1-munged responder-s
                       responder-constants public-constants)
; <snip>
            (let ((network-s-after-4-munged (function-we-know-nothing-about4 network-s-after-4)))

              (mv-let
               (network-s-after-5 initiator-s-after-5)
               (initiator-step3 network-s-after-4-munged initiator-s-after-3
                                initiator-constants public-constants)
                          (implies
                            (and (constantsp constants)
                                     (badly-forged-msg3p (msg3 network-s-after-3-munged)
                                                         (responder-constants constants)
                                                         (public-key-i public-constants))
                                     (badly-forged-msg4p (msg4 network-s-after-4-munged)
                                                         initiator-s-after-3
                                                         initiator-constants
                                                         (public-key-R public-constants)))
                            (and (protocol-failure responder-s-after-4)
                                     (protocol-failure initiator-s-after-5))))))))))))
```

High Level Properties to Prove

- Key Agreement
  - Wouldn't it be lovely:

```
(implies (and (initiator-success initiator-s)
              (responder-success responder-s))
         (equal (session-key initiator)
                (session-key responder)
```

Key Agreement

- ID proof is targeted towards safety while Key agreement proof is targeted towards liveness
- Say that when network messages check out as okay, the key derived in the intiator’s step 2 is equal to something (TDB)
- Say that when network messages check out as okay, the key derived in the responder’s step 2 is equal to something (TBD)
- Use the DH book to show that those two something’s are equal
- Prove that both parties show success only after all network messages they have received “check out”
- Conclude that if all parties have received valid network messages, then their keys must be equal (currently fuzzy)

Wrap-up

- Covered:
  - Derivation of JFKr
  - Books developed for JFKr reasoning
  - Demonstration of the JFKr executable model
  - Security Properties
    - Identity
    - Session Key
- Requires expertise in both ACL2 and security protocols
- Have more than a good start
- Original work so far as I know
  - But JFKr has been formally “verified” before
- Maybe it’s time to move onto wireless protocols, etc.

Resources

- Abadi, Blanchet, Fournet. Just Fast Keying in the Pi Calculus.
- Datta, Derek, Mitchell, and Pavlovic. A Derivation System and Compositional Logic for Security Protocols.
- Kaufmann, Matt and Moore, J Strother. ACL2 FAQ. 2004.
- Levy, Benjamin (translator). Diffie-Helman Method for Key Agreement. 1997.
- Paulson, Lawrence C. Proving Properties by Induction. 1997.
- Shmatikov, Vitaly. Just Fast Keying Slides. 2004.

Resources (cont’d)

Seriously. The derivation of JFKr slides are almost straight from Vitaly Shmatikov’s course.