MARTIN GUNNARSSON

Formal Verification of WirelessHART

Verifying old and finding new attacks

Presentation

- Short overview of WirelessHART
- Previous security analysis of WirelessHART
- Tamarin introduction
- Modeling WirelessHART
- Modeling results & Novel Attack

RI.
SE

Presentation & Paper Outline

- Short overview of WirelessHART
- Previous security analysis of WirelessHART
- Tamarin introduction
- Modeling WirelessHART
- Modeling results & Novel Attack

Contributions:

- Summary of all published WirelessHART attacks.
- Formal verification of Tamarin including previous attacks.
- Discovered a novel re-keying attack.

RI.
SE

WirelessHART - background

- Wireless HART.
- HART is a legacy (wired) protocol for industrial automation.
- Intended for communication between sensors and actuators.
- Controller sends *commands* to the device.
- Wired protocol – no security considerations or mechanisms.

RI.
SE

WirelessHART

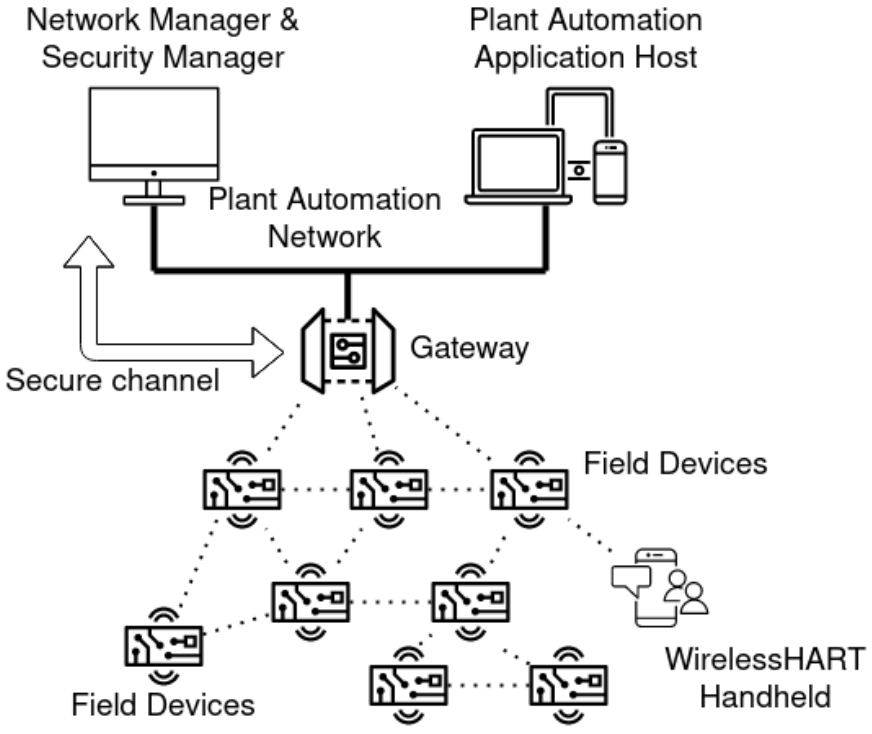
- Intended for communication between sensors and actuators.
- Battery powered wireless device has 30-60% reduced installation costs.
- Uses 7-layer OSI-model with “legacy HART” as *application layer*.
- 2.4GHz IEEE 802.15.4 mesh network as Physical layer.
- At lest 50 million devices installed 2022.
- Supported by major ICS vendors.

RI.
SE

Roles in WirelessHART

- *Field Devices*
- *Gateway*

- *Security & Network Manager*



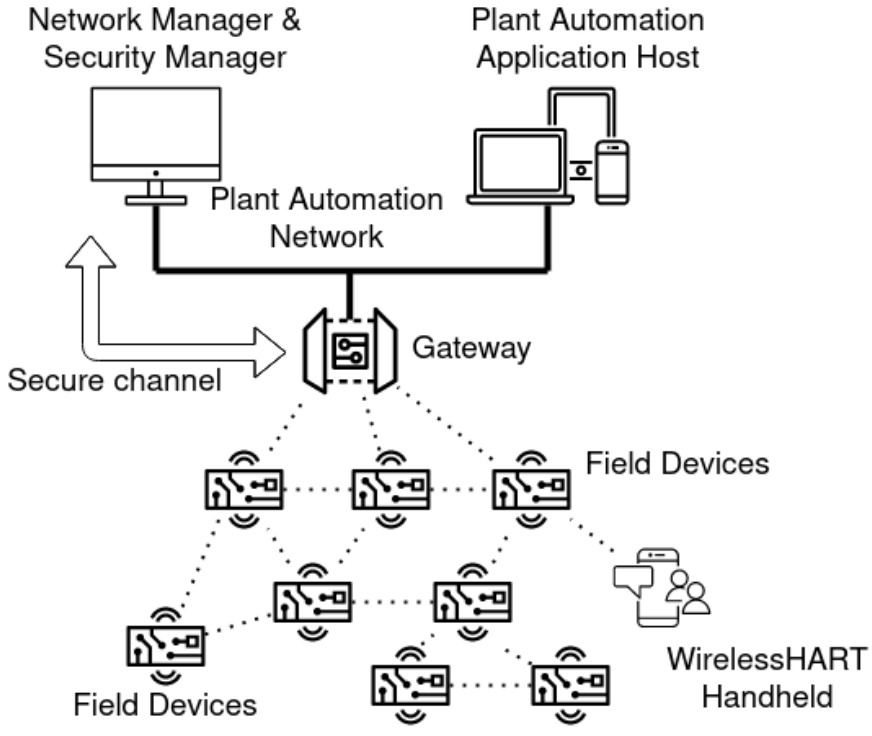
Roles in WirelessHART

- *Field Devices*
- *Gateway*

  - WirelessHART terminate at the *Gateway*

- *Security & Network Manager*

  - Manage the network



WirelessHART security

- Communication can be *unicast* or *broadcast* group communication.
- Messages are end-to-end encrypted and hop-by-hop authenticated.
- Only symmetric cryptography is used.
- AES-128-CCM AEAD.

RI.
SE

WirelessHART security

- Devices are provided with a symmetric Join_key before they join the network.
- Network_Key – Used for hop-by-hop authentication.
- Session Keys – Used for end-to-end encryption of messages.

RI.
SE

WirelessHART sub-protocols

- End-to-End security protocol – Used to communicate data and configurations to Field Devices through *sessions*.
- Join Sequence – Used to authenticate joining Field Device and provide keys and network information.
- Network Key Change Operation – Used to change the keys of Field Devices.

RI.
SE

Previous Security Analysis

RI.
SE

Attackers

- WirelessHART assume two types of attackers, an *Insider Attacker* and an *Outsider Attacker*.
- An *Outsider Attacker* knows nothing.
- An *Insider Attacker* know secrets of the network.
- The previous notion of *Insider Attacker* was too weak.

RI.
SE

Attackers

	Outsider Attacker	Insider Attacker (old)	Insider Attacker (new)
Join_Key	Unknown	Unknown	Known (Unique)
Network_Key	Unknown	Known	Known
BC Session Key	Unknown	Unknown	Known
Session Key	Unknown	Unknown	Known (Unique)
Well-Known-Key	Known	Known	Known

RI.
SE

Related Work

- Several attacks have been published:
  - Jamming attacks, message corruption attacks, disconnect Sybil-attacks, pre-join exhaustion attacks, command injection attacks
- A formal verification of WirelessHART in 2021, only covered the end-to-end security protocol.

RI.
SE

Jamming Attacks

- Blocking the Physical or MAC layer.
- Smart jamming that predict frequencies and slots to selectively jam one device.
- Several attacks published [1],[2]

[1] Raza, Shahid, et al. "Security considerations for the WirelessHART protocol." 2009 IEEE Conference on Emerging Technologies & Factory Automation. IEEE, 2009.

[2] Cheng, Xia, et al. "Launching smart selective jamming attacks in wirelesshart networks." IEEE INFOCOM 2021-IEEE Conference on Computer Communications. IEEE, 2021.

RI.
SE

Disconnect Sybil Attack

- Impersonating another device, telling neighbors the ‘device’ will leave the network.
- Lack of authentication on the Data Layer.

[3] Bayou, Lyes, et al. "Security issue of WirelessHART based SCADA systems." International Conference on Risks and Security of Internet and Systems. Springer, Cham, 2015.

RI.
SE

Command Injection Attack

- An *Insider Attacker* impersonate the Gateway and send ‘commands’ to a Field Device.
- Several variants exists. Direct Command Injection, Bounced Command Injection, and On-the-fly Command Injection.
- Lack of source-authentication due to compromised symmetric keys.

[4] Bayou, Lyes, et al. "Security analysis of WirelessHART communication scheme." International Symposium on Foundations and Practice of Security. Springer, Cham, 2016.

RI.
SE

Previously Published Attacks

Attack	STRIDE CLASSIFICATION	Attacker
Jamming Attack	DoS	Outsider
Disconnect Sybil Attack	Impersonation	Outsider
Pre-Join Exhaustion	DoS	Outsider
Direct Command Injection	Impersonation	Insider
Bounced Command Injection	Impersonation	Insider
Con-the-fly Command Injection	Tampering	Insider

RI.
SE

Security Analysis With Tamarin

RI.
SE

Protocol analysis with Tamarin

- Model the protocol using a protocol modeling language.
- Analyze protocols within a security-model.
- State the security properties as *lemmas*, to be proven.
- Automated reasoning gives proofs that lemmas *hold* or a *counterexample*.
- Intended as a *supplement* to traditional methods.

RI.
SE

Tamarin prover

- Protocols are represented using *rules* and work as a state-machine.
- An execution of a protocol is called a *Trace*.
- *Traces* are used to evaluate if a stated security lemma *holds*, or to provide a *counterexample*.
- Protocols are analyzed within the Dolev-Yao model.
  - The adversary *is* the network.

RI.
SE

Diffie-Hellman in Tamarin

```
rule r1:
  [Fr(~i_sk)]
   -->
  [!Initiator($I, ~i_sk), Out(‘g’^~i_sk]
```

RI.
SE

Diffie-Hellman in Tamarin

```
rule r1:
  [Fr(~i_sk)]
   -->
  [!Initiator($I, ~i_sk), Out('g'^~i_sk]

rule r2:
    let:
    session_key = i_pk^~r_sk
    [In(i_pk), Fr(~r_sk)]
    --[ Resp_S(session_key) ]->
    [!Responder($R,~r_sk, session_key), Out('g'^~r_sk)]
```

RI.
SE

Diffie-Hellman in Tamarin

```
rule r1:
  [Fr(~i_sk)]
   -->
  [!Initiator($I, ~i_sk), Out('g'^~i_sk]

rule r2:
    let:
    session_key = i_pk^~r_sk
    [In(i_pk), Fr(~r_sk)]
    --[ Resp_S(session_key) ]->
    [!Responder($R,~r_sk, session_key), Out('g'^~r_sk)]

rule r3:
    [In(r_pk), !Initiator($I, ~i_sk)]
    --[ Init_S(r_pk^~i_sk) ]->
    [!Initiator_1($I, ~i_sk, r_pk^~i_sk)]
```

RI. SE

Diffie-Hellman in Tamarin

```
lemma session_key_secrecy:
    “All Ses_K #i #j.
    Resp_S(Ses_K) @ #i
    & Init_S(Ses_K) @ #j &
    #j < #i &
    not(Ex #k. K(Ses_K) @ #k)”
```

RI.
SE

Diffie-Hellman in Tamarin

```
lemma session_key_secrecy:
    “All Ses_K #i #j.
    Resp_S(Ses_K) @ #i
    & Init_S(Ses_K) @ #j &
    #j < #i &
    not(Ex #k. K(Ses_K) @ #k)”
```

```
lemma session_key_secrecy:
    “All Ses_K1 Ses_K2 #i #j.
    Resp_S(Ses_K1) @ #i &
    Init_S(Ses_K2) @ #j &
    #j < #i &
    not(Ex #k1 #k2. K(Ses_K1) @ #k1 | K(Ses_K2) @ #k2 )”
```

RI. SE

Modeling and Verifying WirelessHART

RI.
SE

Formal verification of WirelessHART

- We have modeled all sub-protocols of WirelessHART relevant to security.
- Replay-protection was not modeled.

RI.
SE

Formal verification of WirelessHART

- We have modeled all sub-protocols of WirelessHART relevant to security.
- Replay-protection was not modeled.

- Split our model into four parts:
  - End-to-End Security Protocol
  - Join Sequence
  - Network key Change Operation
  - DLPDU advertisements

RI.
SE

Formal verification of WirelessHART

- Split our model into four parts:
  - End-to-End Security Protocol
  - Join Sequence
  - Network key Change Operation
  - DLPDU advertisements

<u>Properties tested:</u>

- Secrecy
- Authentication
- Integrity

All properties were tested against an Insider Attacker.

RI.
SE

End-to-End Security Protocol

- Sessions are used to communicate data and configurations to devices.
- Can be used in unicast or broadcast mode.
- Provides confidentiality, integrity protection, replay protection.
- A system wide Network_key is intended to protect the network from an *Outsider Attacker*.

RI.
SE

```
rule Gateway_send_command_UC:
        let
        NL_msg_s = ~command
        NL_header_s = <~NL_nonce_s, 'Command_pkt', 'NL_Ctrl', 'Sec_Ctrl'>
        NL_key_s = ~D_GW_UC

        DL_payload_s = aead_enc(NL_key_s, ~NL_nonce_s, NL_header_s, NL_msg_s)

        DL_key_s = ~NetworkKey
        DL_header_s = <~DL_nonce_s, $device_id, $Gateway_ID>
        DL_mac_s = mac( < DL_header_s, DL_payload_s>, DL_key_s)
        encrypted_message = <DL_header_s, NL_header_s, DL_payload_s, DL_mac_s>

        in
        [!Gateway_1($Gateway_ID, $NetworkID, ~NetworkKey, ~D_GW_BC, ~D_GW_UC),
         !Field_Device_0($device_id, $NetworkID, ~JoinKey, ~NetworkKey, ~D_GW_BC,
~D_NM_BC, ~D_GW_UC, ~D_NM_UC),
         Fr(~DL_nonce_s), Fr(~NL_nonce_s), Fr(~command)]
        --[SendCommand_UC($Gateway_ID, $device_id, ~command)]->
        [Out(encrypted_message) ]
```

RI.
SE

```
rule Device_receive_command_UC:
        let
        /* Incoming Message */
        DL_header_r = <DL_nonce_r, receiver, source>
        NL_header_r = <NL_nonce_r, a, b, c>
        incoming_message = < DL_header_r, NL_header_r, DL_payload_r, DL_MAC_r>
         NL_key_r = ~D_GW_UC
        decrypted_command = aead_dec(NL_key_r, NL_nonce_r, NL_header_r,
DL_payload_r)
        in
        [In(incoming_message),
         !Field_Device_0($device_id, $NetworkID, ~JoinKey, ~NetworkKey, ~D_GW_BC,
~D_NM_BC, ~D_GW_UC, ~D_NM_UC)
        ]
       --[
           Eq( verify_mac(DL_MAC_r, <DL_header_r, DL_payload_r>, ~NetworkKey),
true),
           Eq( aead_verify(NL_key_r, NL_nonce_r, NL_header_r, DL_payload_r), true),
           ReceivedCommand_UC(source, receiver, decrypted_command),
           Authentic(source, receiver, decrypted_command, ~NetworkKey)
        ]->
        []
```

RI. SE

End-to-End Protocol

- We have modeled end-to-end protocol over unicast and broadcast sessions.
- Rules for sending and receiving messages.

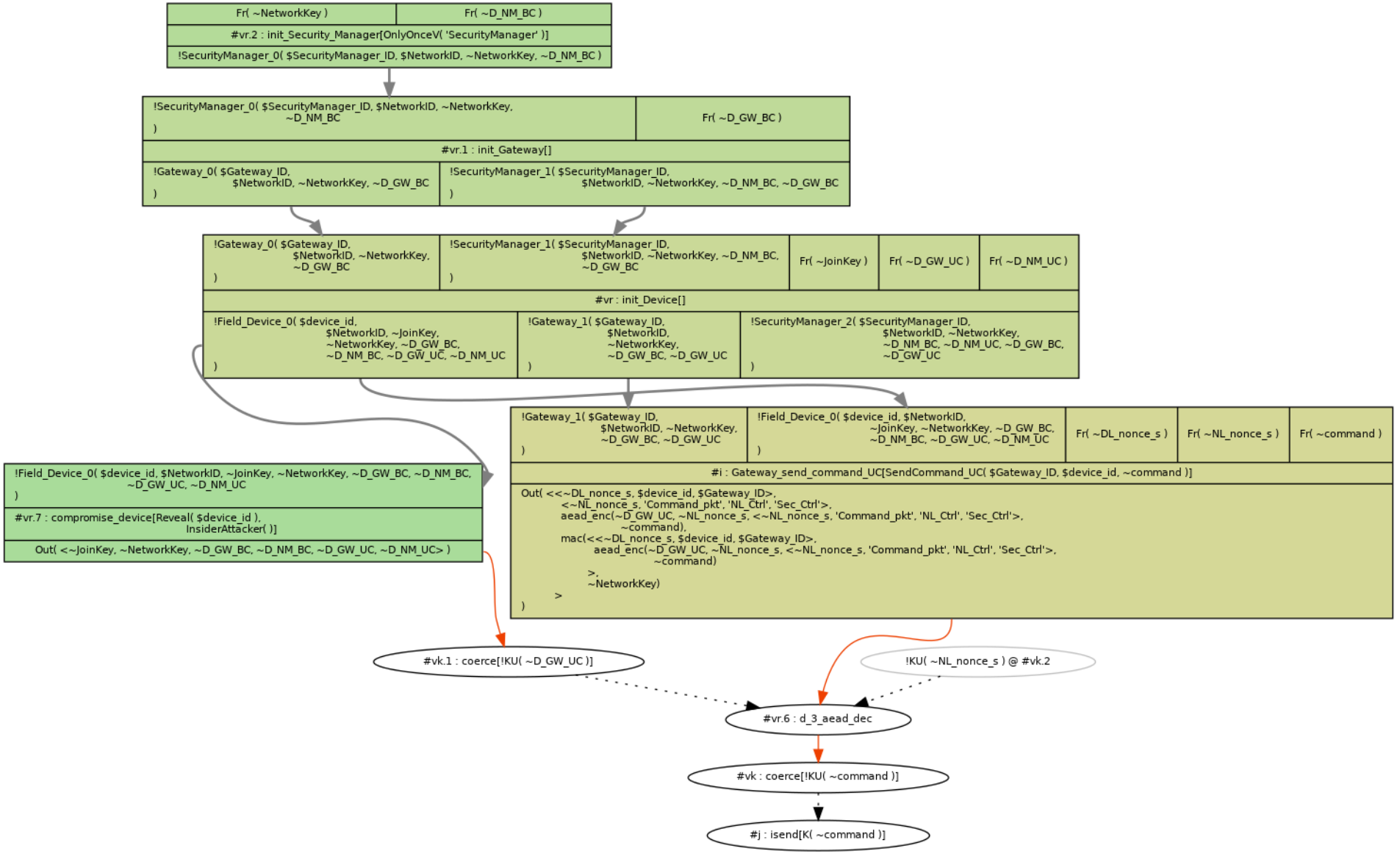
	Unicast	Broadcast
Command Secrecy	Hold	Does not hold
Command Authentication	Hold	Does not Hold

RI.
SE

Lemmas

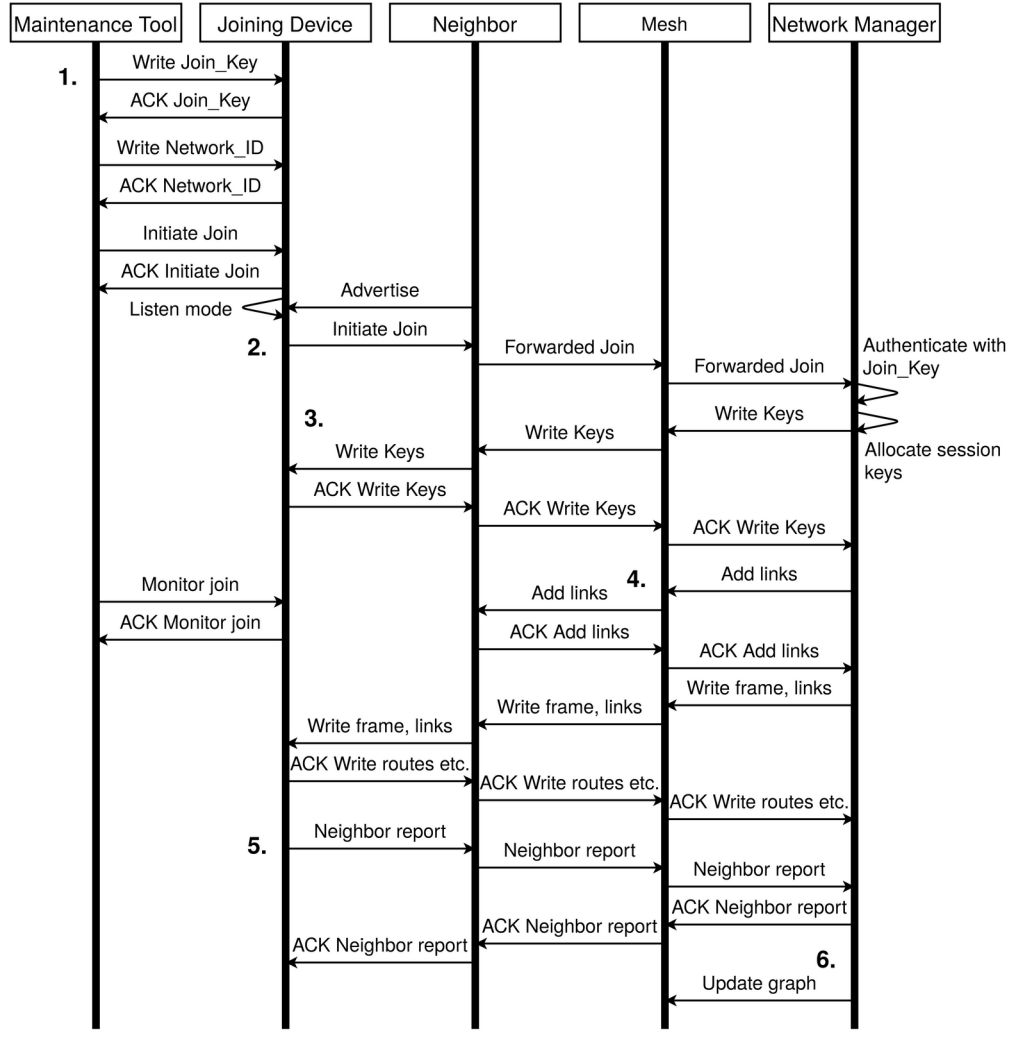
```
/* A Field Device D will believe a message coming from
Gateway GW over unicast to be authentic, or D's keys has been
revealed. */
lemma message_authentication_UC:
"All GW D cmd #i. ReceivedCommand_UC(GW, D, cmd) @i
==> (Ex #j. SendCommand_UC(GW, D, cmd) @j & j<i) | (Ex #k.
Reveal(D) @ #k) | (Ex c #m. Reveal(c) @ #m & not c = D)"

/* A 'command' cmd sent over unicast to a Field Device D that
is known by ADV implies D has been compromised. */
lemma command_secrecy_UC:
      "All GW D cmd #i #j.
       SendCommand_UC(GW, D, cmd) @ #i &
       K(cmd) @ #j
       ==>
         Ex #k. Reveal(D) @ #k"
```

RI. SE


Fr(~NetworkKey)
Fr(~D_NM_BC)
#vr.2 : init_Security_Manager[OnlyOnceV('SecurityManager')]
!SecurityManager_0($SecurityManager_ID, $NetworkID, ~NetworkKey, ~D_NM_BC)
!SecurityManager_0($SecurityManager_ID, $NetworkID, ~NetworkKey, ~D_NM_BC)
Fr(~D_GW_BC)
#vr.1 : init_Gateway[]
!Gateway_0($Gateway_ID, $NetworkID, ~NetworkKey, ~D_GW_BC)
!SecurityManager_1($SecurityManager_ID, $NetworkID, ~NetworkKey, ~D_NM_BC, ~D_GW_BC)
!Gateway_0($Gateway_ID, $NetworkID, ~NetworkKey, ~D_GW_BC)
!SecurityManager_1($SecurityManager_ID, $NetworkID, ~NetworkKey, ~D_NM_BC, ~D_GW_BC)
Fr(~JoinKey)
Fr(~D_GW_UC)
Fr(~D_NM_UC)
#vr : init_Device[]
!Field_Device_0($device_id, $NetworkID, ~JoinKey, ~NetworkKey, ~D_GW_BC, ~D_NM_BC, ~D_GW_UC, ~D_NM_UC)
!Gateway_1($Gateway_ID, $NetworkID, ~NetworkKey, ~D_GW_BC, ~D_GW_UC)
!SecurityManager_2($SecurityManager_ID, $NetworkID, ~NetworkKey, ~D_NM_BC, ~D_NM_UC, ~D_GW_BC, ~D_GW_UC)
!Field_Device_0($device_id, $NetworkID, ~JoinKey, ~NetworkKey, ~D_GW_BC, ~D_NM_BC, ~D_GW_UC, ~D_NM_UC)
#vr.7 : compromise_device[Reveal($device_id), InsiderAttacker( )]
Out(<~JoinKey, ~NetworkKey, ~D_GW_BC, ~D_NM_BC, ~D_GW_UC, ~D_NM_UC>)
!Gateway_1($Gateway_ID, $NetworkID, ~NetworkKey, ~D_GW_BC, ~D_GW_UC)
!Field_Device_0($device_id, $NetworkID, ~JoinKey, ~NetworkKey, ~D_GW_BC, ~D_NM_BC, ~D_GW_UC, ~D_NM_UC)
Fr(~DL_nonce_s)
Fr(~NL_nonce_s)
Fr(~command)
#i : Gateway_send_command_UC[SendCommand_UC($Gateway_ID, $device_id, ~command)]
Out(<<~DL_nonce_s, $device_id, $Gateway_ID>, <~NL_nonce_s, 'Command_pkt', 'NL_Ctrl', 'Sec_Ctrl'>, aead_enc(~D_GW_UC, ~NL_nonce_s, <~NL_nonce_s, 'Command_pkt', 'NL_Ctrl', 'Sec_Ctrl'>, ~command), mac(<<~DL_nonce_s, $device_id, $Gateway_ID>, aead_enc(~D_GW_UC, ~NL_nonce_s, <~NL_nonce_s, 'Command_pkt', 'NL_Ctrl', 'Sec_Ctrl'>, ~command) >, ~NetworkKey) >)
#vk.1 : coerce[!KU(~D_GW_UC)]
!KU(~NL_nonce_s) @ #vk.2
#vr.6 : d_3_aead_dec
#vk : coerce[!KU(~command)]
#j : isend[K(~command)]

Join Sequence

- Authenticate a device to the network.
- Provision of keys.
- Provision of network specific information.



Join procedure

- We have modeled Join procedure, excluding steps for network configuration.
- Rules for sending and receiving “join requests”, and “write keys” messages.
- Join_key *can* be unique.

	Unique Join_Key
Provisioned Key Secrecy	Hold
Provisioned Key Authentication	Hold

RI.
SE

Lemmas

```
/* For all protocol executions, a key K received at #m
implies that it was sent at #l and that #l happened before #m
*/
lemma key_agreement:
        all-traces
        "All a k #m.
           KeysReceived(a, k) @ #m
          ==>
           (Ex #l. KeysSent(a, k) @ #l) |
           (Ex #j. InsiderAttacker() @ #j)
        "
```

RI.
SE

Lemmas

```
/* For all protocol executions, a key K received at #m
implies that it was sent at #l and that #l happened before #m
*/
lemma key_secrecy:
        all-traces
        "All a k #m #n.
           KeysReceived(a, k) @ #m &
           K(k) @ #n
          ==>
           (Ex #l. KeysSent(a, k) @ #l) &
           (Ex #m. InsiderAttacker() @ #m)
        "
```

RI.
SE

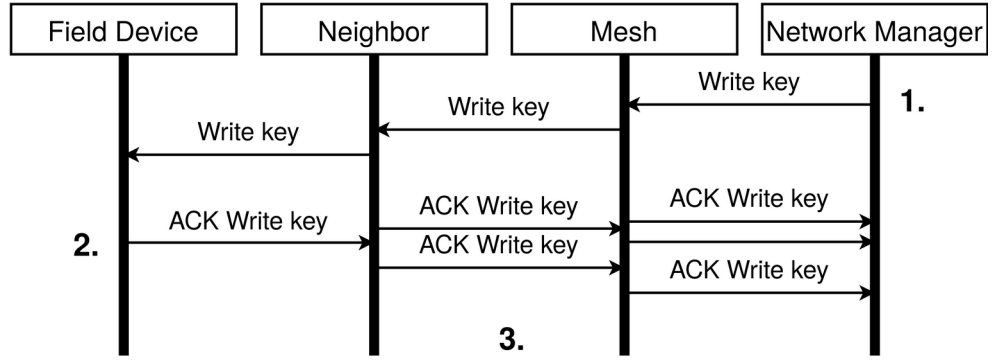
Previously Published Attacks

Attack	STRIDE	Attacker	Verified?
Jamming Attack	DoS	Outsider	-
Disconnect Sybil Attack	Impersonation	Outsider	Yes
Pre-Join Exhaustion	DoS	Outsider	Yes
Direct Command Injection	Impersonation	Insider	Yes
Bounced Command Injection	Impersonation	Insider	Yes
Con-the-fly Command Injection	Tampering	Insider	Yes

RI.
SE

Network Key Change Operation

- New keys are sent to a *Field Device* over a unicast or broadcast session.
- The receiving device(es) will switch to the new key.



Modeling Network key change

- We have modeled Network key change over unicast and broadcast sessions.
- Rules for sending and receiving messages.
- We found a novel attack!

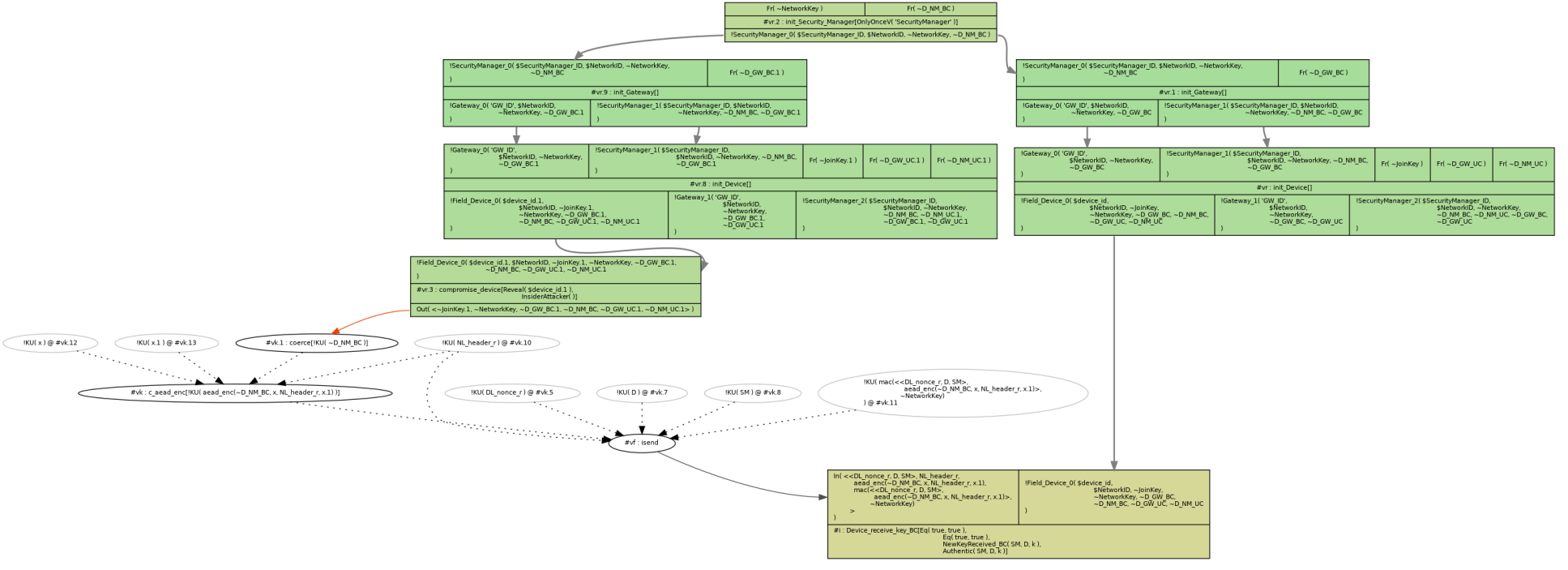
	Unicast	Broadcast
New Key Secrecy	Hold	Does not hold
New Key Authentication	Hold	Does not Hold

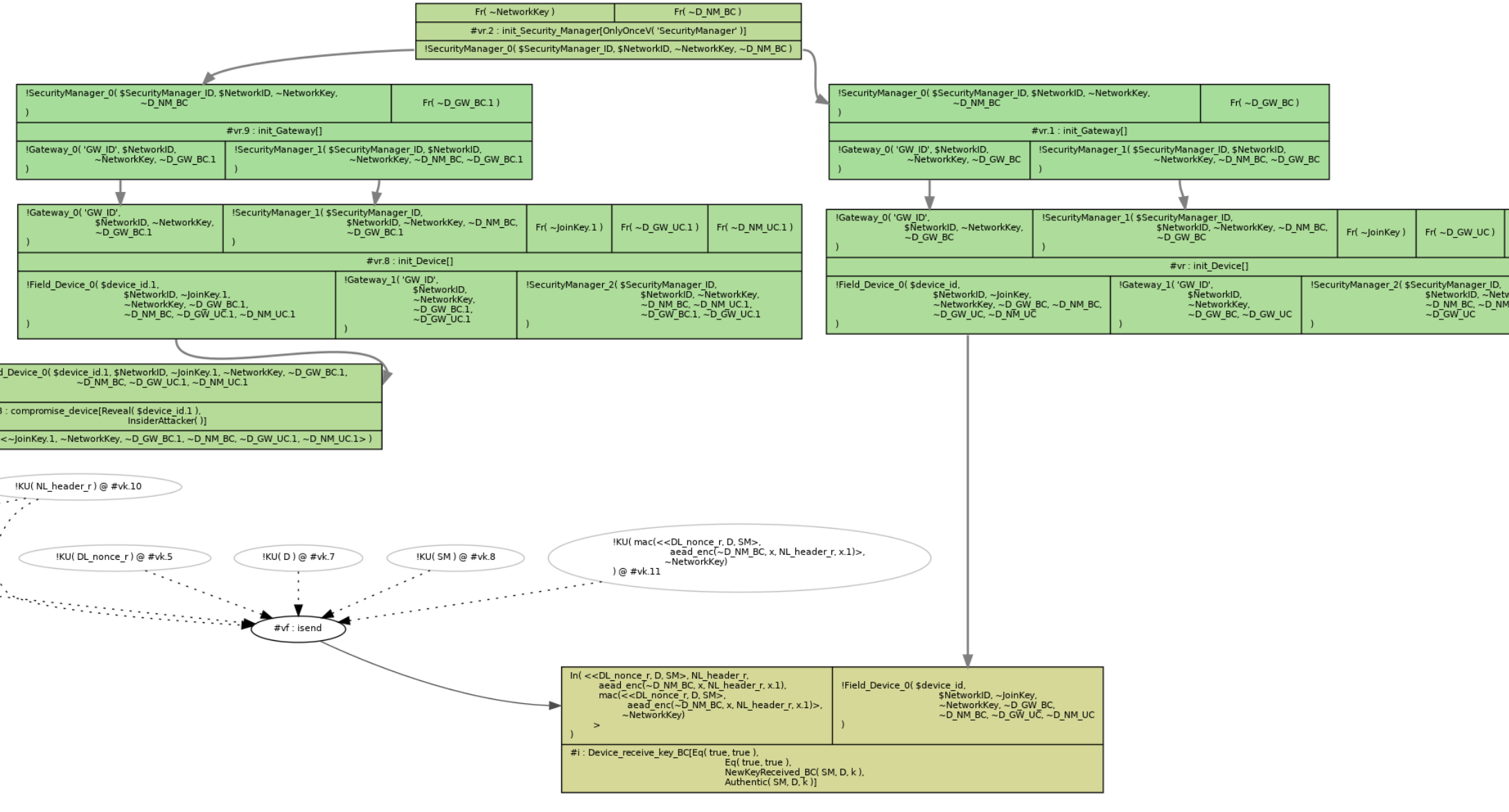
RI.
SE

Lemmas

```
/* A Device D will believe a message, sent over broadcast, to be
authentic, or any device's key has been revealed */
lemma key_authentication_BC:
"All SM D k #i. NewKeyReceived_BC(SM,D,k) @i
==> (Ex #j. SendKey_BC(SM,D,k) @j & j<i) | (Ex #k. InsiderAttacker() @ #k)
"

/* A key k sent to device D, over broadcast, that is known by ADV implies
D has been compromised. */
lemma key_secrecy_BC:
      "All SM D k #i #j.
       SendKey_BC(SM, D, k) @ #i &
       K(k) @ #j
       ==>
         Ex #k. InsiderAttacker() @ #k
       "
```

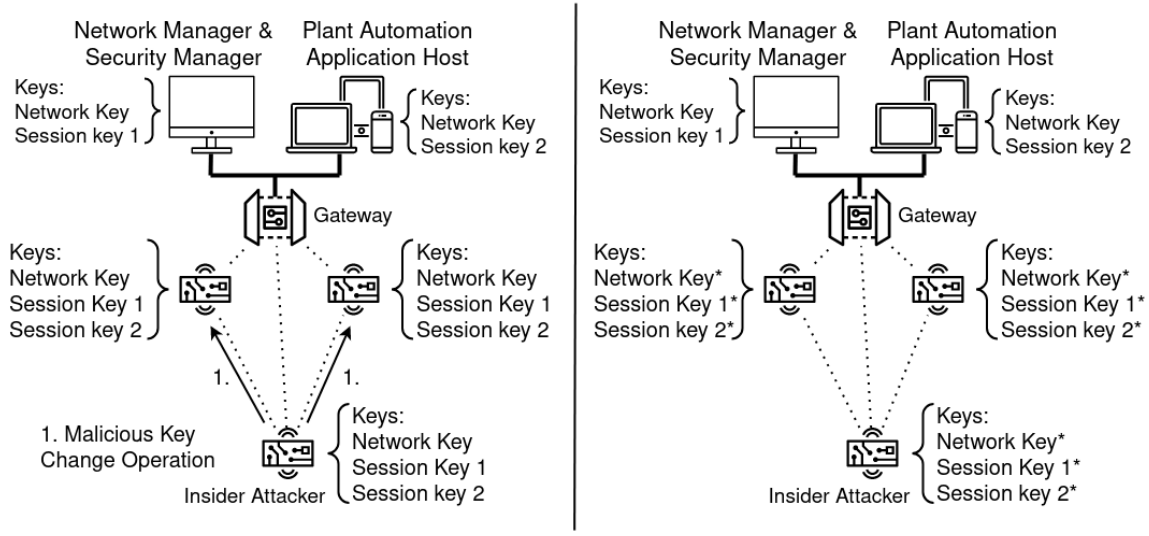
RI. SE

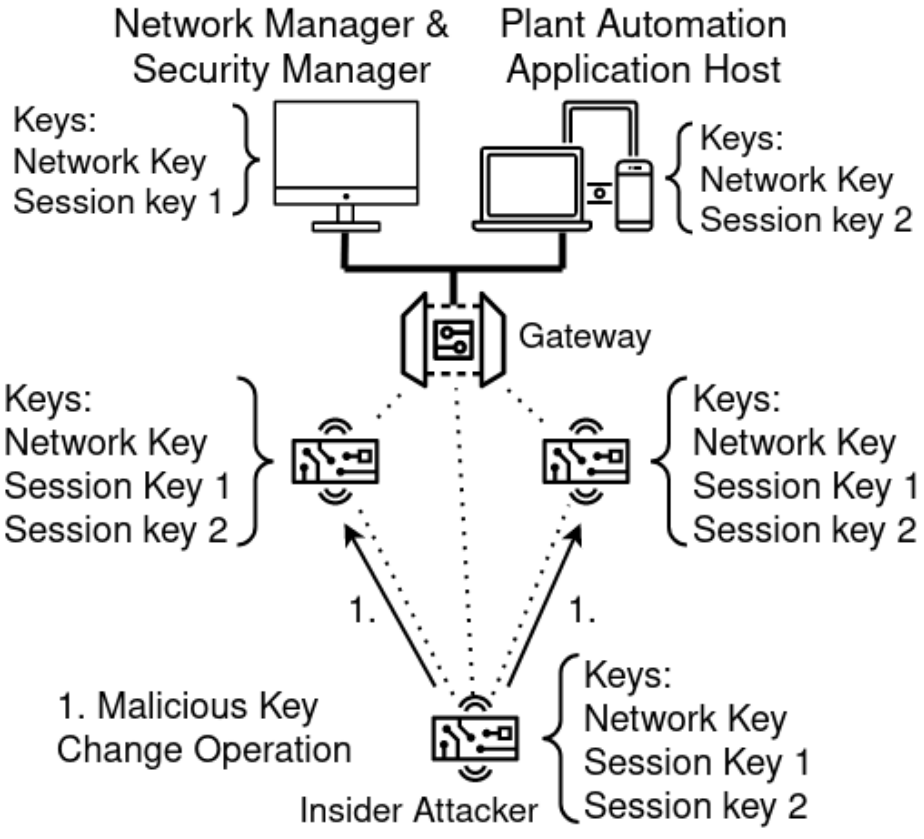

Fr(~NetworkKey)
Fr(~D_NM_BC)
#vr.2 : init_Security_Manager[OnlyOnceVl ‘SecurityManager’)]
!SecurityManager_0($SecurityManager_ID, $NetworkID, ~NetworkKey, ~D_NM_BC)
!SecurityManager_0($SecurityManager_ID, $NetworkID, ~NetworkKey, ~D_NM_BC)
Fr(~D_GW_BC.1)
#vr.9 : init_Gateway[]
!Gateway_0(‘GW_ID’, $NetworkID, ~NetworkKey, ~D_GW_BC.1)
!SecurityManager_1($SecurityManager_ID, $NetworkID, ~NetworkKey, ~D_NM_BC, ~D_GW_BC.1)
!SecurityManager_0($SecurityManager_ID, $NetworkID, ~NetworkKey, ~D_NM_BC)
Fr(~D_GW_BC)
#vr.1 : init_Gateway[]
!Gateway_0(‘GW_ID’, $NetworkID, ~NetworkKey, ~D_GW_BC)
!SecurityManager_1($SecurityManager_ID, $NetworkID, ~NetworkKey, ~D_NM_BC, ~D_GW_BC)
!Gateway_0(‘GW_ID’, $NetworkID, ~NetworkKey, ~D_GW_BC.1)
!SecurityManager_1($SecurityManager_ID, $NetworkID, ~NetworkKey, ~D_NM_BC, ~D_GW_BC.1)
Fr(~JoinKey.1)
Fr(~D_GW_UC.1)
Fr(~D_NM_UC.1)
#vr.8 : init_Device[]
!Field_Device_0($device_id.1, $NetworkID, ~JoinKey.1, ~NetworkKey, ~D_GW_BC.1, ~D_NM_BC, ~D_GW_UC.1, ~D_NM_UC.1)
!Gateway_1(‘GW_ID’, $NetworkID, ~NetworkKey, ~D_GW_BC.1, ~D_GW_UC.1)
!SecurityManager_2($SecurityManager_ID, $NetworkID, ~NetworkKey, ~D_NM_BC, ~D_NM_UC.1, ~D_GW_BC.1, ~D_GW_UC.1)
!Gateway_0(‘GW_ID’, $NetworkID, ~NetworkKey, ~D_GW_BC)
!SecurityManager_1($SecurityManager_ID, $NetworkID, ~NetworkKey, ~D_NM_BC, ~D_GW_BC)
Fr(~JoinKey)
Fr(~D_GW_UC)
Fr(~D_NM_UC)
#vr : init_Device[]
!Field_Device_0($device_id, $NetworkID, ~JoinKey, ~NetworkKey, ~D_GW_BC, ~D_NM_BC, ~D_GW_UC, ~D_NM_UC)
!Gateway_1(‘GW_ID’, $NetworkID, ~NetworkKey, ~D_GW_BC, ~D_GW_UC)
!SecurityManager_2($SecurityManager_ID, $NetworkID, ~NetworkKey, ~D_NM_BC, ~D_NM_UC, ~D_GW_BC, ~D_GW_UC)
!Field_Device_0($device_id.1, $NetworkID, ~JoinKey.1, ~NetworkKey, ~D_GW_BC.1, ~D_NM_BC, ~D_GW_UC.1, ~D_NM_UC.1)
#vr.3 : compromise_device[Reveal($device_id.1), InsiderAttacker( )]
Out(<~JoinKey.1, ~NetworkKey, ~D_GW_BC.1, ~D_NM_BC, ~D_GW_UC.1, ~D_NM_UC.1>)
!KU(x) @ #vk.12
!KU(x.1) @ #vk.13
#vk.1 : coerce[!KU(~D_NM_BC)]
!KU(NL_header_r) @ #vk.10
#vk : c_aead_enc[!KU(aead_enc(~D_NM_BC, x, NL_header_r, x.1))]
!KU(DL_nonce_r) @ #vk.5
!KU(D) @ #vk.7
!KU(SM) @ #vk.8
!KU(mac(<<DL_nonce_r, D, SM>, aead_enc(~D_NM_BC, x, NL_header_r, x.1)>, ~NetworkKey)) @ #vk.11
#vf : isend
In(<<DL_nonce_r, D, SM>, NL_header_r, aead_enc(~D_NM_BC, x, NL_header_r, x.1), mac(<<DL_nonce_r, D, SM>, aead_enc(~D_NM_BC, x, NL_header_r, x.1)>, ~NetworkKey) >)
!Field_Device_0($device_id, $NetworkID, ~JoinKey, ~NetworkKey, ~D_GW_BC, ~D_NM_BC, ~D_GW_UC, ~D_NM_UC)
#i : Device_receive_key_BC[Eq(true, true), Eq(true, true), NewKeyReceived_BC(SM, D, k), Authentic(SM, D, k)]

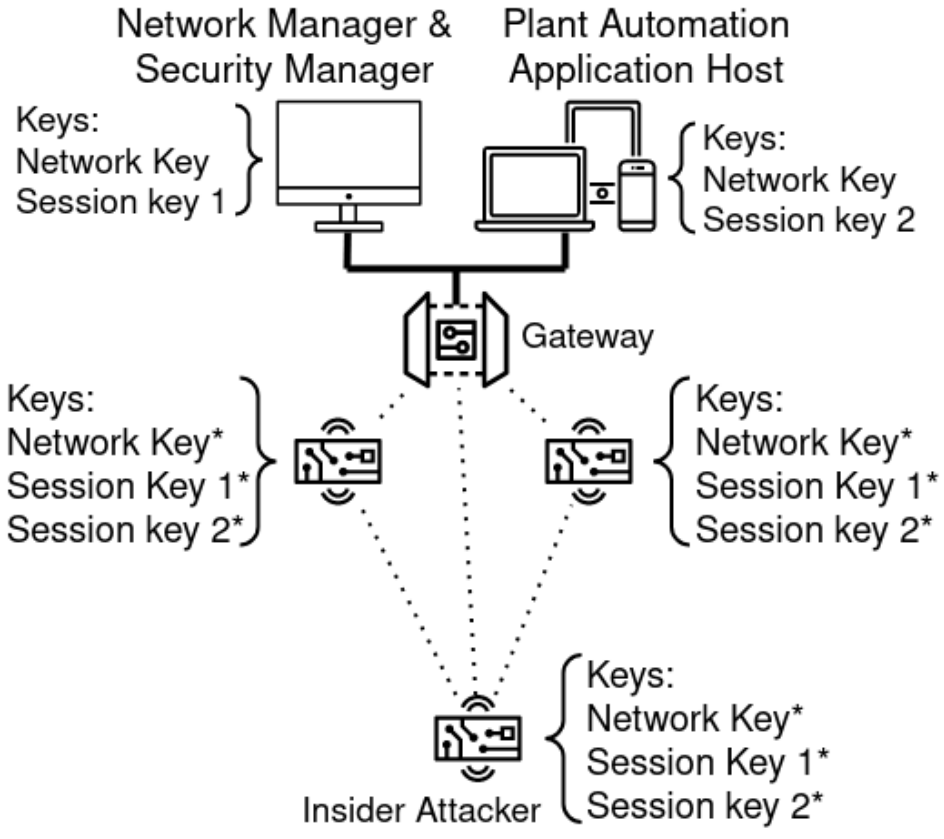

Fr(~NetworkKey)
Fr(~D_NM_BC)
#vr.2 : init_Security_Manager[OnlyOnceV('SecurityManager')]
!SecurityManager_0($SecurityManager_ID, $NetworkID, ~NetworkKey, ~D_NM_BC)
!SecurityManager_0($SecurityManager_ID, $NetworkID, ~NetworkKey, ~D_NM_BC)
Fr(~D_GW_BC.1)
#vr.9 : init_Gateway[]
!Gateway_0('GW_ID', $NetworkID, ~NetworkKey, ~D_GW_BC.1)
!SecurityManager_1($SecurityManager_ID, $NetworkID, ~NetworkKey, ~D_NM_BC, ~D_GW_BC.1)
!SecurityManager_0($SecurityManager_ID, $NetworkID, ~NetworkKey, ~D_NM_BC)
Fr(~D_GW_BC)
#vr.1 : init_Gateway[]
!Gateway_0('GW_ID', $NetworkID, ~NetworkKey, ~D_GW_BC)
!SecurityManager_1($SecurityManager_ID, $NetworkID, ~NetworkKey, ~D_NM_BC, ~D_GW_BC)
!Gateway_0('GW_ID', $NetworkID, ~NetworkKey, ~D_GW_BC.1)
!SecurityManager_1($SecurityManager_ID, $NetworkID, ~NetworkKey, ~D_NM_BC, ~D_GW_BC.1)
Fr(~JoinKey.1)
Fr(~D_GW_UC.1)
Fr(~D_NM_UC.1)
#vr.8 : init_Device[]
!Field_Device_0($device_id.1, $NetworkID, ~JoinKey.1, ~NetworkKey, ~D_GW_BC.1, ~D_NM_BC, ~D_GW_UC.1, ~D_NM_UC.1)
!Gateway_1('GW_ID', $NetworkID, ~NetworkKey, ~D_GW_BC.1, ~D_GW_UC.1)
!SecurityManager_2($SecurityManager_ID, $NetworkID, ~NetworkKey, ~D_NM_BC, ~D_NM_UC.1, ~D_GW_BC.1, ~D_GW_UC.1)
!Gateway_0('GW_ID', $NetworkID, ~NetworkKey, ~D_GW_BC)
!SecurityManager_1($SecurityManager_ID, $NetworkID, ~NetworkKey, ~D_NM_BC, ~D_GW_BC)
Fr(~JoinKey)
Fr(~D_GW_UC)
#vr : init_Device[]
!Field_Device_0($device_id, $NetworkID, ~JoinKey, ~NetworkKey, ~D_GW_BC, ~D_NM_BC, ~D_GW_UC, ~D_NM_UC)
!Gateway_1('GW_ID', $NetworkID, ~NetworkKey, ~D_GW_BC, ~D_GW_UC)
!KU(NL_header_r) @ #vk.10
!KU(DL_nonce_r) @ #vk.5
!KU(D) @ #vk.7
!KU(SM) @ #vk.8
!KU(mac(<<DL_nonce_r, D, SM>, aead_enc(~D_NM_BC, x, NL_header_r, x.1)>, ~NetworkKey)) @ #vk.11
#vf : isend
In(<<DL_nonce_r, D, SM>, NL_header_r, aead_enc(~D_NM_BC, x, NL_header_r, x.1), mac(<<DL_nonce_r, D, SM>, aead_enc(~D_NM_BC, x, NL_header_r, x.1)>, ~NetworkKey) >)
!Field_Device_0($device_id, $NetworkID, ~JoinKey, ~NetworkKey, ~D_GW_BC, ~D_NM_BC, ~D_GW_UC, ~D_NM_UC)
#i : Device_receive_key_BC[Eq(true, true), Eq(true, true), NewKeyReceived_BC(SM, D, k), Authentic(SM, D, k)]

Malicious re-keying attack

- An *Insider Attacker* can spoof Network Key Change Messages.
- An attacker can then change the keys of uncompromised devices.
- Changing keys will isolate devices from the *Gateway* and *Security Manager*.




Network Manager &
Security Manager
Plant Automation
Application Host
Keys:
Network Key
Session key 1
Keys:
Network Key
Session key 2
Gateway
Keys:
Network Key
Session Key 1
Session key 2
Keys:
Network Key
Session Key 1
Session key 2
1.
1.
1. Malicious Key
Change Operation
Insider Attacker
Keys:
Network Key
Session Key 1
Session key 2


Network Manager &
Security Manager
Plant Automation
Application Host
Keys:
Network Key
Session key 1
Keys:
Network Key
Session key 2
Gateway
Keys:
Network Key*
Session Key 1*
Session key 2*
Keys:
Network Key*
Session Key 1*
Session key 2*
Insider Attacker
Keys:
Network Key*
Session Key 1*
Session key 2*

Re-keying attack impact

- The Network Manager and Plant Automation Host will not be able to contact victims.
- Changing the Network_Key cause devices with the new key to drop all legitimate messages.
- Compromise of one device can lead to DoS of the entire network.
- Devices have to be manually recovered.

RI.
SE

Conclusion

- WirelessHART security suffer from a inherent vulnerabilities.
- A more rigorous process when developing protocols is needed.
- Yet another attack against WirelessHART.
- Formal verification is a valuable tool when designing and evaluating protocols.
- A re-design of the protocol, with new considerations, is probably necessary.

RI.
SE

THANK YOU!

Questions?

RI.SE

THANK YOU!

Questions?