AUTO-ISAC

The SMEP Attack

Friedrich Wiemer
Robert Bosch GmbH
XC-CE/ECS1

July 29th, 2025

The SMEP Attack

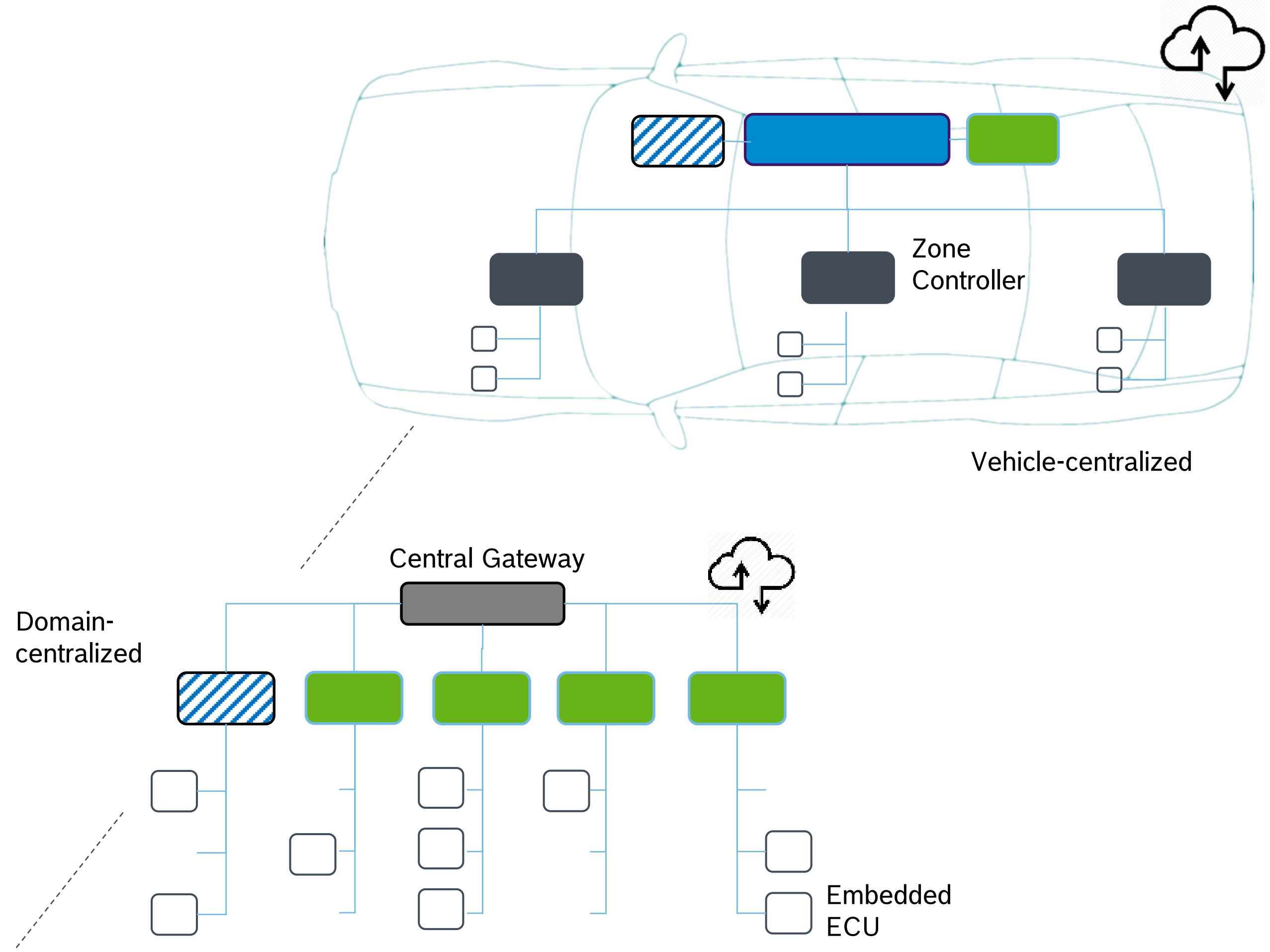
RSA

Public key	$e, n = pq$
Private key	d with $x^{(e \cdot d)} \bmod n = x$
Signing of m	$s = m^d \bmod n$
Verification of s	$m =?= s^e \bmod n$
Encryption of m	$c = m^e \bmod n$
Decryption of c	$m = c^d \bmod n$

Promise: its not getting worse 🙃

The SMEP Attack

EE Architecture, Gateways or Zone Controllers and Switches

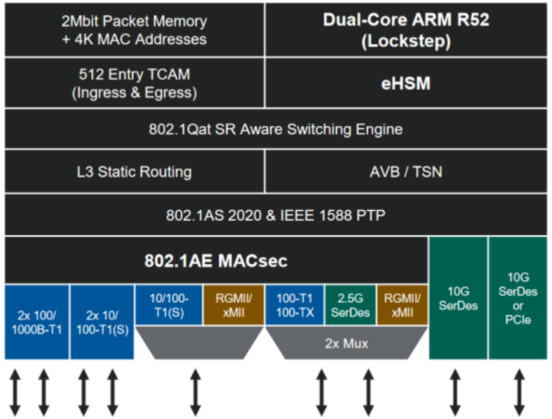


- In-vehicle network (IVN) connected by
  - Central Gateway
  - Zone Controllers & Vehicle Computers

Central Gateway

Zone Controller

Automotive Switch

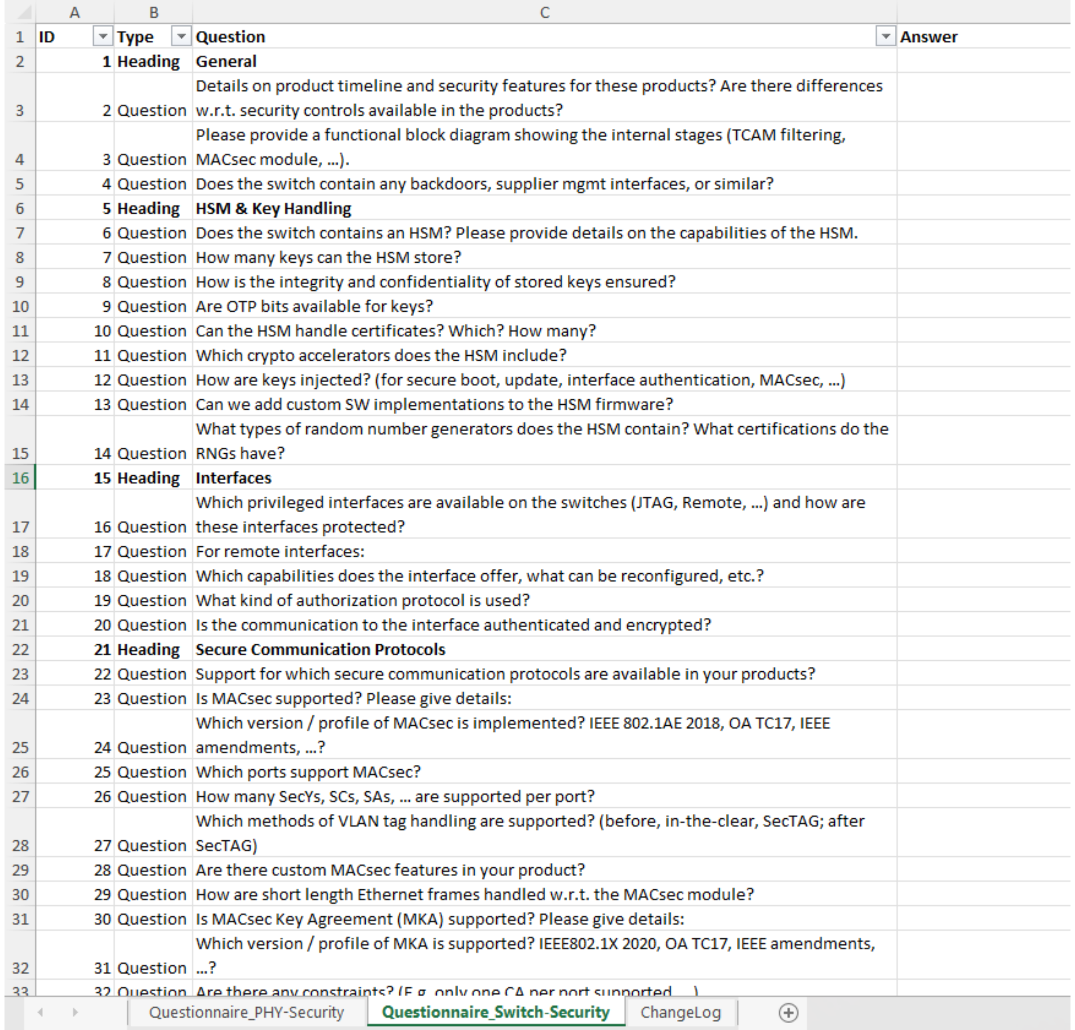


2Mbit Packet Memory + 4K MAC Addresses					Dual-Core ARM R52 (Lockstep)			
512 Entry TCAM (Ingress & Egress)					eHSM			
802.1Qat SR Aware Switching Engine								
L3 Static Routing					AVB / TSN			
802.1AS 2020 & IEEE 1588 PTP								
802.1AE MACsec							10G SerDes	10G SerDes or PCIe
2x 100/ 1000B-T1	2x 10/ 100-T1(S)	10/100- T1(S)	RGMII/ xMII	100-T1 100-TX	2.5G SerDes	RGMII/ xMII		
				2x Mux				

The SMEP Attack

Supplier Mgmt: Switch Evaluation

- As part of pre-development: analyze and evaluate switch vendors
- Set of topics that we discuss with vendor
- Discussions done with several switch vendors



	A	B	C	
1	**ID**	**Type**	**Question**	**Answer**
2	**1**	**Heading**	**General**	
3	2	Question	Details on product timeline and security features for these products? Are there differences w.r.t. security controls available in the products?	
4	3	Question	Please provide a functional block diagram showing the internal stages (TCAM filtering, MACsec module, …).	
5	4	Question	Does the switch contain any backdoors, supplier mgmt interfaces, or similar?	
6	**5**	**Heading**	**HSM & Key Handling**	
7	6	Question	Does the switch contains an HSM? Please provide details on the capabilities of the HSM.	
8	7	Question	How many keys can the HSM store?	
9	8	Question	How is the integrity and confidentiality of stored keys ensured?	
10	9	Question	Are OTP bits available for keys?	
11	10	Question	Can the HSM handle certificates? Which? How many?	
12	11	Question	Which crypto accelerators does the HSM include?	
13	12	Question	How are keys injected? (for secure boot, update, interface authentication, MACsec, …)	
14	13	Question	Can we add custom SW implementations to the HSM firmware?	
15	14	Question	What types of random number generators does the HSM contain? What certifications do the RNGs have?	
16	**15**	**Heading**	**Interfaces**	
17	16	Question	Which privileged interfaces are available on the switches (JTAG, Remote, …) and how are these interfaces protected?	
18	17	Question	For remote interfaces:	
19	18	Question	Which capabilities does the interface offer, what can be reconfigured, etc.?	
20	19	Question	What kind of authorization protocol is used?	
21	20	Question	Is the communication to the interface authenticated and encrypted?	
22	**21**	**Heading**	**Secure Communication Protocols**	
23	22	Question	Support for which secure communication protocols are available in your products?	
24	23	Question	Is MACsec supported? Please give details:	
25	24	Question	Which version / profile of MACsec is implemented? IEEE 802.1AE 2018, OA TC17, IEEE amendments, …?	
26	25	Question	Which ports support MACsec?	
27	26	Question	How many SecYs, SCs, SAs, … are supported per port?	
28	27	Question	Which methods of VLAN tag handling are supported? (before, in-the-clear, SecTAG; after SecTAG)	
29	28	Question	Are there custom MACsec features in your product?	
30	29	Question	How are short length Ethernet frames handled w.r.t. the MACsec module?	
31	30	Question	Is MACsec Key Agreement (MKA) supported? Please give details:	
32	31	Question	Which version / profile of MKA is supported? IEEE802.1X 2020, OA TC17, IEEE amendments, …?	
33	32	Question	Are there any constraints? (E.g. only one CA per port supported, …)	

The SMEP Attack
Disclaimer

- The following slides contain Realtek proprietary information
- We have responsibly disclosed the following vulnerability to Realtek and suggested improvements
- Realtek fixed the protocol and published according updates to the switch firmware

- Realtek approved these slides and this talk at AUTO-ISAC

We appreciate Realtek’s collaboration and the very serious and constructive handling of this topic!

REALTEK

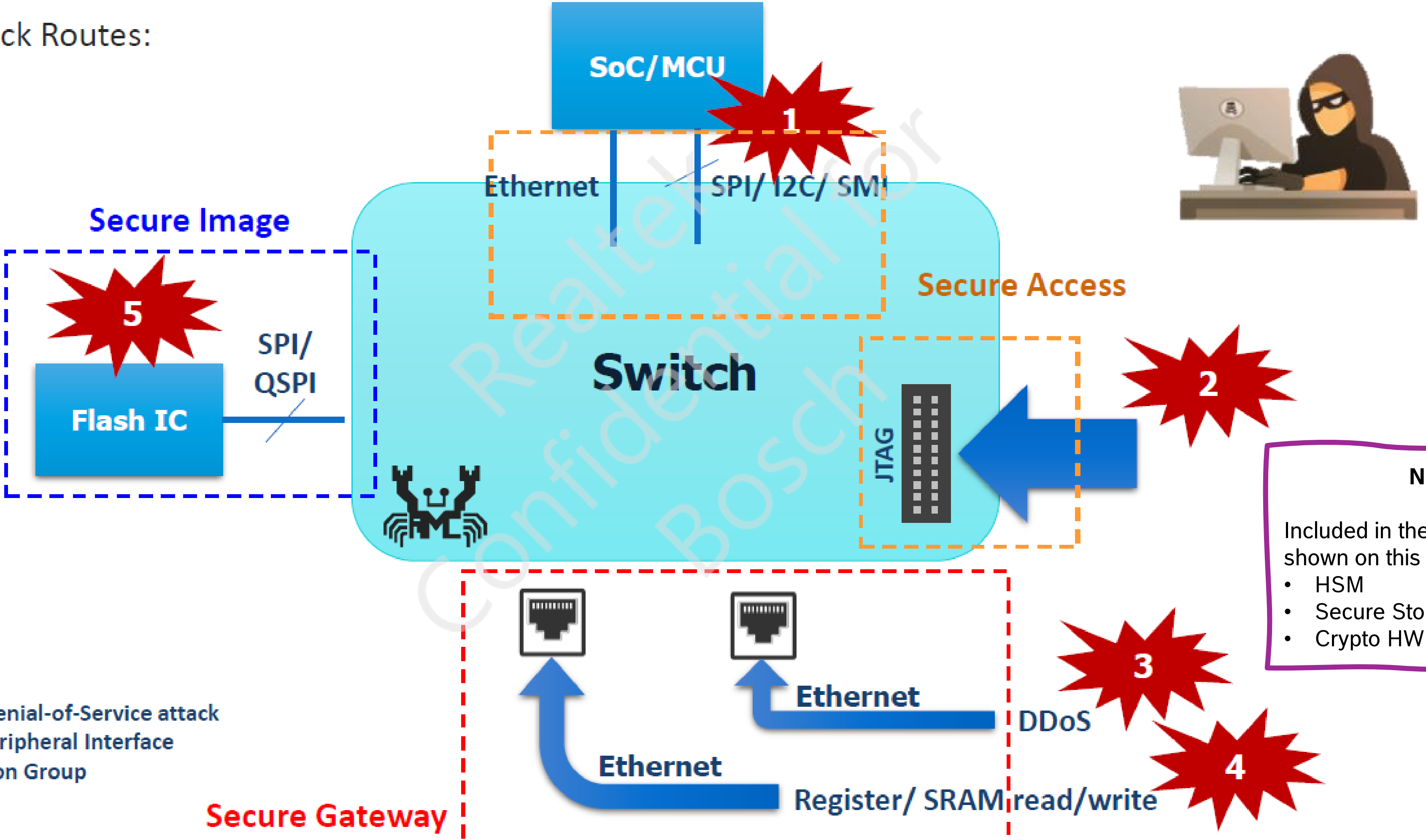
Security

23rd March. 2022

CNBG
REALTEK COMMUNICATIONS NETWORK BG

Security Consideration

- Possible Attack Routes:



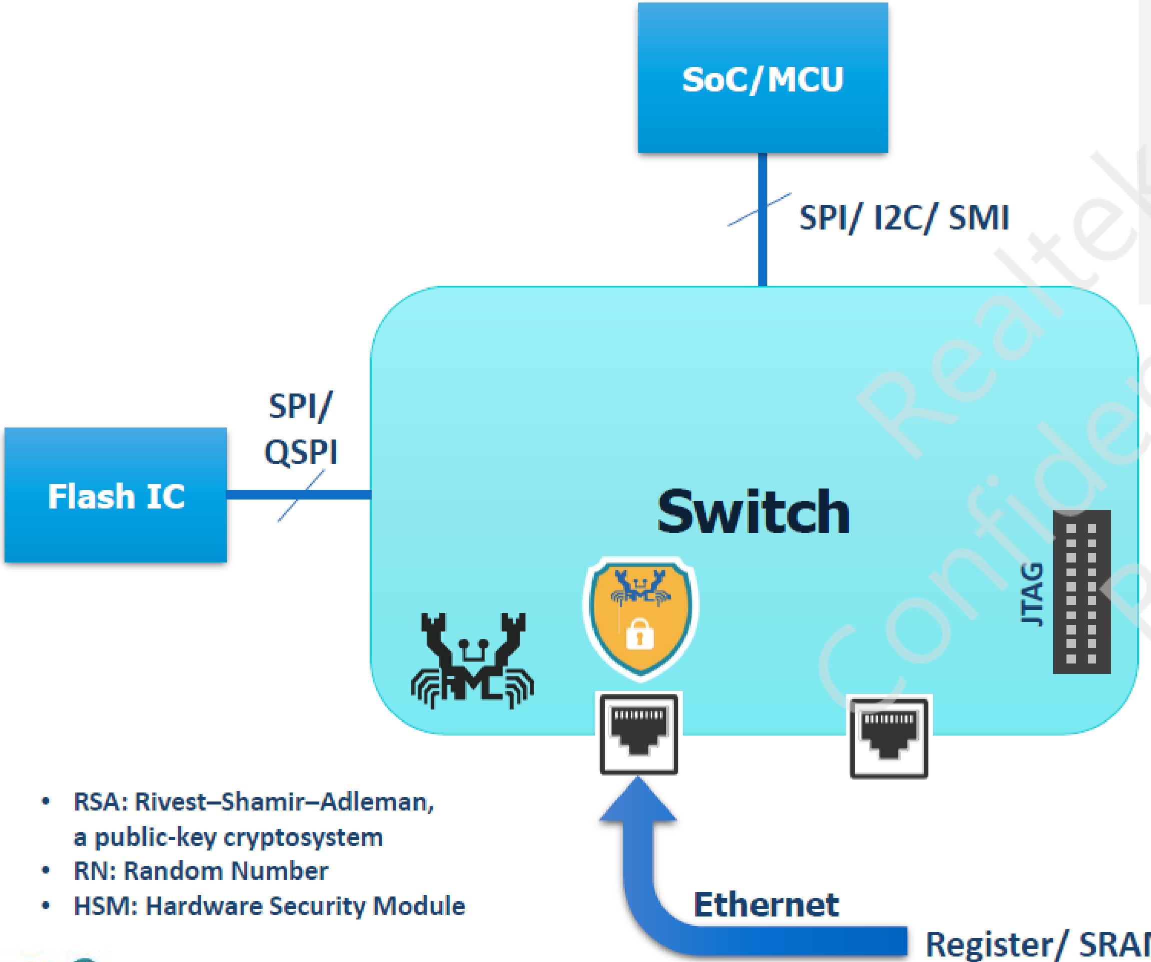
Note

Included in the switch but not shown on this slide:

- HSM
- Secure Storage
- Crypto HW Accelerators

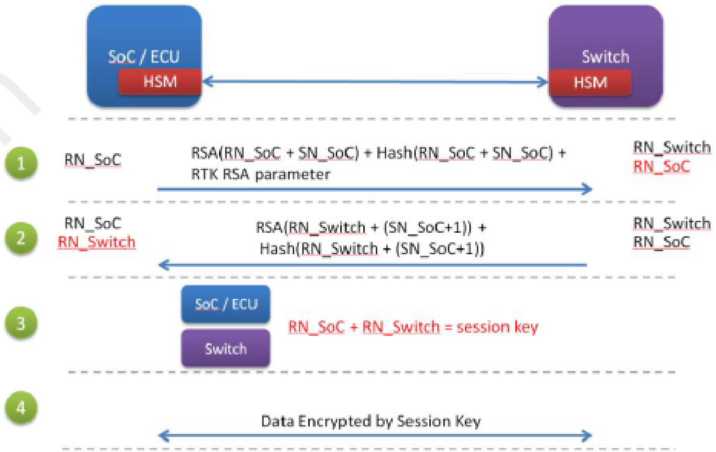
- DDoS: Distributed Denial-of-Service attack
- QSPI: Quad Serial Peripheral Interface
- JTAG: Joint Test Action Group

CASE_IV: Attack from Ethernet...



Realtek LOCK function & Secure Access

1. SRAM/Register access via Ethernet: Disabled by OTP
2. Secure Access: Using AES key to encrypt message.

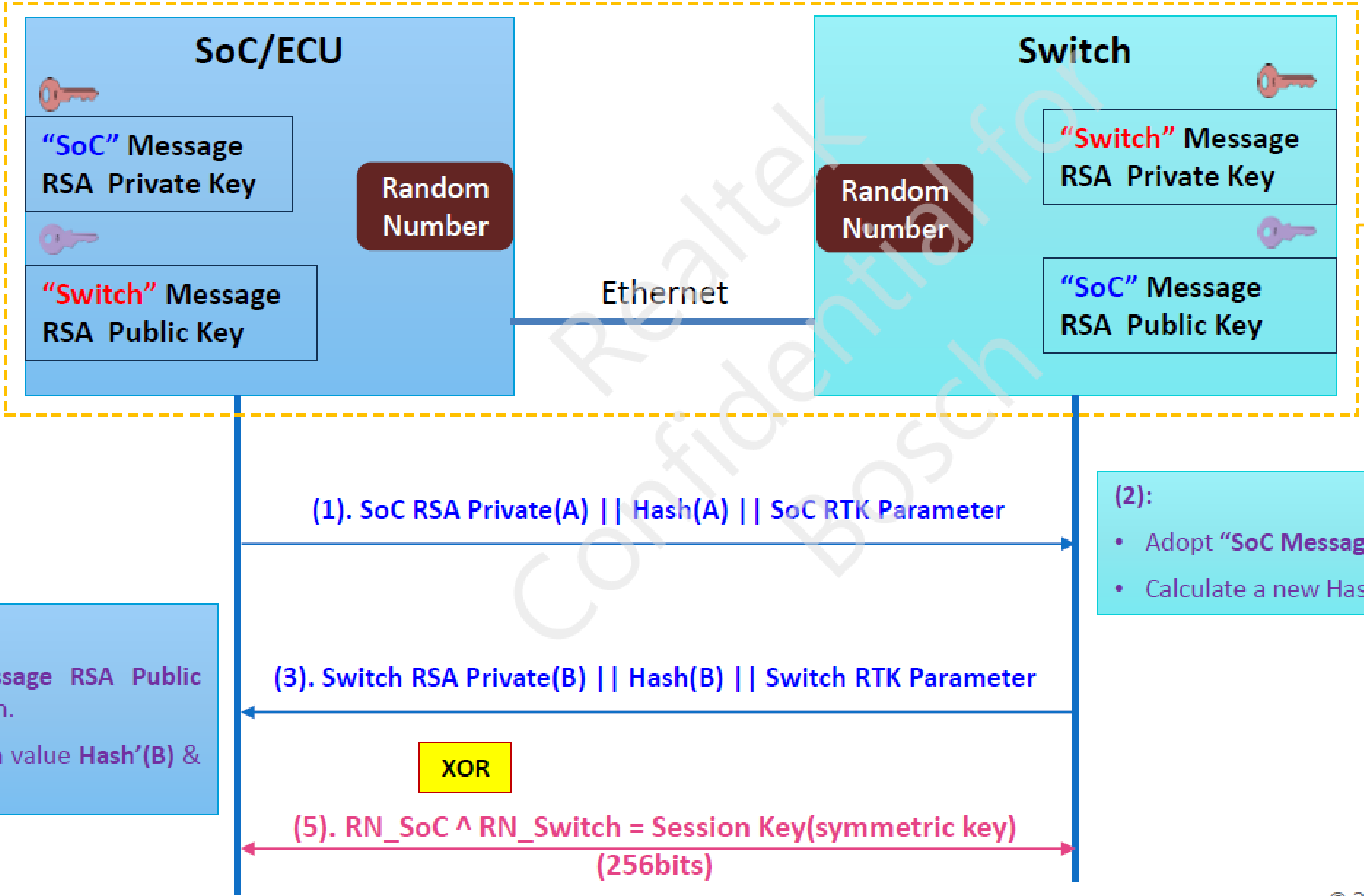

SoC / ECU
HSM
Switch
HSM
1
RN_SoC
RSA(RN_SoC + SN_SoC) + Hash(RN_SoC + SN_SoC) + RTK RSA parameter
RN_Switch
RN_SoC
2
RN_SoC
RN_Switch
RSA(RN_Switch + (SN_SoC+1)) + Hash(RN_Switch + (SN_SoC+1))
RN_Switch
RN_SoC
3
SoC / ECU
Switch
RN_SoC + RN_Switch = session key
4
Data Encrypted by Session Key

- RSA: Rivest–Shamir–Adleman, a public-key cryptosystem
- RN: Random Number
- HSM: Hardware Security Module

Secure Access: Session Key & Secure Message

- Generate "Session Key" and adopt it to do "Data Encryption(secure message)".

- Requirement for SoC/ECU:
  1. RSA2048 (no padding scheme)
  2. AES256-CBC
  3. SHA1
  4. RNG



SoC/ECU

"SoC" Message RSA Private Key

"Switch" Message RSA Public Key

Random Number

Ethernet

Switch

"Switch" Message RSA Private Key

"SoC" Message RSA Public Key

Random Number

(1). SoC RSA Private(A) || Hash(A) || SoC RTK Parameter

(3). Switch RSA Private(B) || Hash(B) || Switch RTK Parameter

XOR

(5). RN_SoC ^ RN_Switch = Session Key(symmetric key) (256bits)

SOC and Switch must pre-store the counterpart's RSA public key, and its own RSA private key (we call them TLS keys).

(2):

- Adopt **"SoC Message RSA Public Key"** to do decryption.
- Calculate a new Hash value **Hash'(A)** & compare to **Hash(A)**

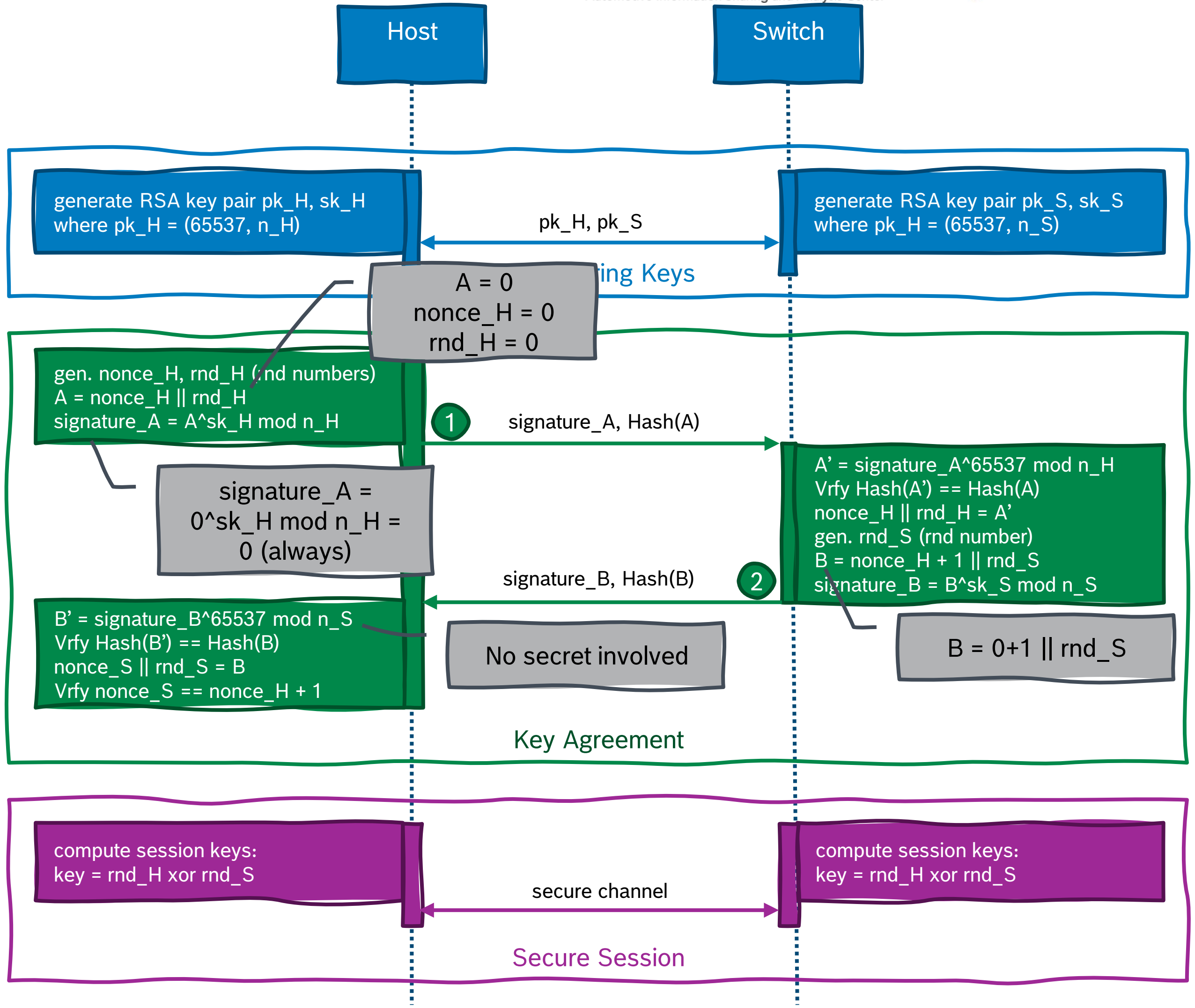
(4):

- Adopt **"Switch Message RSA Public Key"** to do decryption.
- Calculate a new Hash value **Hash'(B)** & compare to **Hash(B)**

- Definition:
  1. A = SN_SoC || RN_SoC
  2. B = SN_SoC+1 || RN_Switch

The SMEP Attack

- The key agreement
  1. The host generates a challenge (nonce_H) as well as his randomness for the session key (rnd_H) and “signs” it (or encrypts it?)
  2. The switch verifies the “signature”, computes the response (nonce_H + 1), his randomness for the session key (rnd_S) and “signs” it
- The actual attack
  - Attacker chooses A = nonce_H || rnd_H = 0
  - signature_A is then always 0
  - Switch will verify signature and accepts the challenge = 0
  - Attacker can “verify”/“decrypt” the signature and learns rnd_S
  - Attacker can compute key = 0 xor rnd_S



Host

Switch

generate RSA key pair pk_H, sk_H
where pk_H = (65537, n_H)

pk_H, pk_S

generate RSA key pair pk_S, sk_S
where pk_H = (65537, n_S)

ing Keys

A = 0
nonce_H = 0
rnd_H = 0

gen. nonce_H, rnd_H (rnd numbers)
A = nonce_H || rnd_H
signature_A = A^sk_H mod n_H

1 signature_A, Hash(A)

A’ = signature_A^65537 mod n_H
Vrfy Hash(A’) == Hash(A)
nonce_H || rnd_H = A’
gen. rnd_S (rnd number)
B = nonce_H + 1 || rnd_S
signature_B = B^sk_S mod n_S

signature_A =
0^sk_H mod n_H =
0 (always)

signature_B, Hash(B) 2

B’ = signature_B^65537 mod n_S
Vrfy Hash(B’) == Hash(B)
nonce_S || rnd_S = B
Vrfy nonce_S == nonce_H + 1

No secret involved

B = 0+1 || rnd_S

Key Agreement

compute session keys:
key = rnd_H xor rnd_S

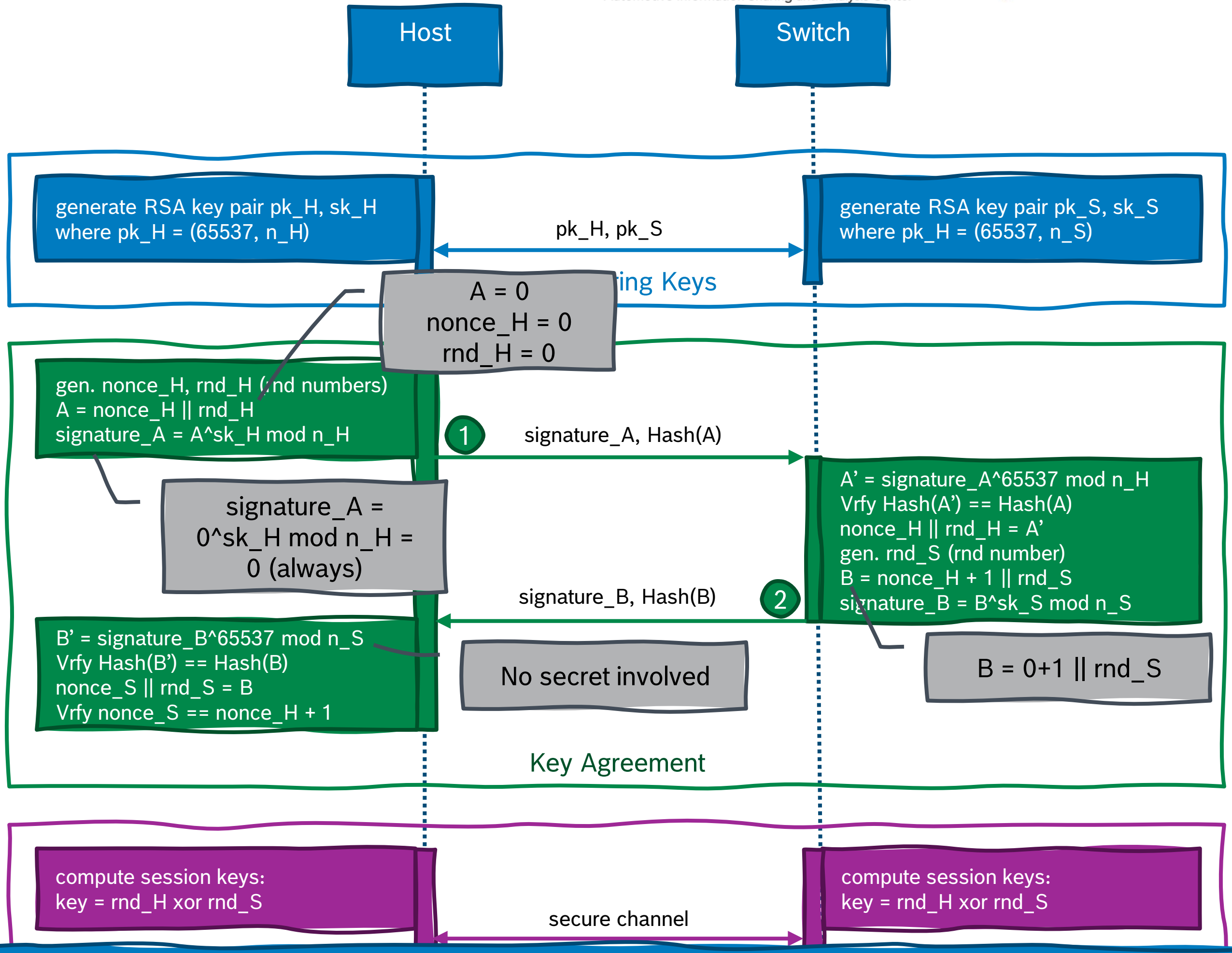
secure channel

compute session keys:
key = rnd_H xor rnd_S

Secure Session

The SMEP Attack

- The key agreement
  1. The host generates a challenge (nonce_H) as well as his randomness for the session key (rnd_H) and "signs" it (or encrypts it?)
  2. The switch verifies the "signature", computes the response (nonce_H + 1), his randomness for the session key (rnd_S) and "signs" it
- The actual attack
  - Attacker chooses A = nonce_H || rnd_H = 0
  - signature_A is then always 0
  - Switch will verify signature and accepts the challenge = 0
  - Attacker can "verify"/"decrypt" the signature and learns rnd_S



Host

Switch

generate RSA key pair pk_H, sk_H
where pk_H = (65537, n_H)

pk_H, pk_S

generate RSA key pair pk_S, sk_S
where pk_H = (65537, n_S)

ing Keys

A = 0
nonce_H = 0
rnd_H = 0

gen. nonce_H, rnd_H (rnd numbers)
A = nonce_H || rnd_H
signature_A = A^sk_H mod n_H

1 signature_A, Hash(A)

A' = signature_A^65537 mod n_H
Vrfy Hash(A') == Hash(A)
nonce_H || rnd_H = A'
gen. rnd_S (rnd number)
B = nonce_H + 1 || rnd_S
signature_B = B^sk_S mod n_S

signature_A =
0^sk_H mod n_H =
0 (always)

signature_B, Hash(B) 2

B' = signature_B^65537 mod n_S
Vrfy Hash(B') == Hash(B)
nonce_S || rnd_S = B
Vrfy nonce_S == nonce_H + 1

No secret involved

B = 0+1 || rnd_S

Key Agreement

compute session keys:
key = rnd_H xor rnd_S

secure channel

compute session keys:
key = rnd_H xor rnd_S

Actually, the shown attack is only one possibility to break this protocol – can you find more?

The SMEP Attack

Extracting the public key

- Side note: how hard is it to learn the public key?
  - Realtek stated the public key cannot be exported from the switch’s memory
- However, we can use the switch as an oracle for this:
  1. Guess public exponent $e = 65537$
  2. Use the switch’ implementation to generate k many message / signature pairs (m_i, s_i)
  3. Compute

  $\gcd(m_1 - s_1^e, m_2 - s_2^e, \ldots, m_k - s_k^e) = 1$ or n_S

I promised no more math – so you have to trust me here 😉

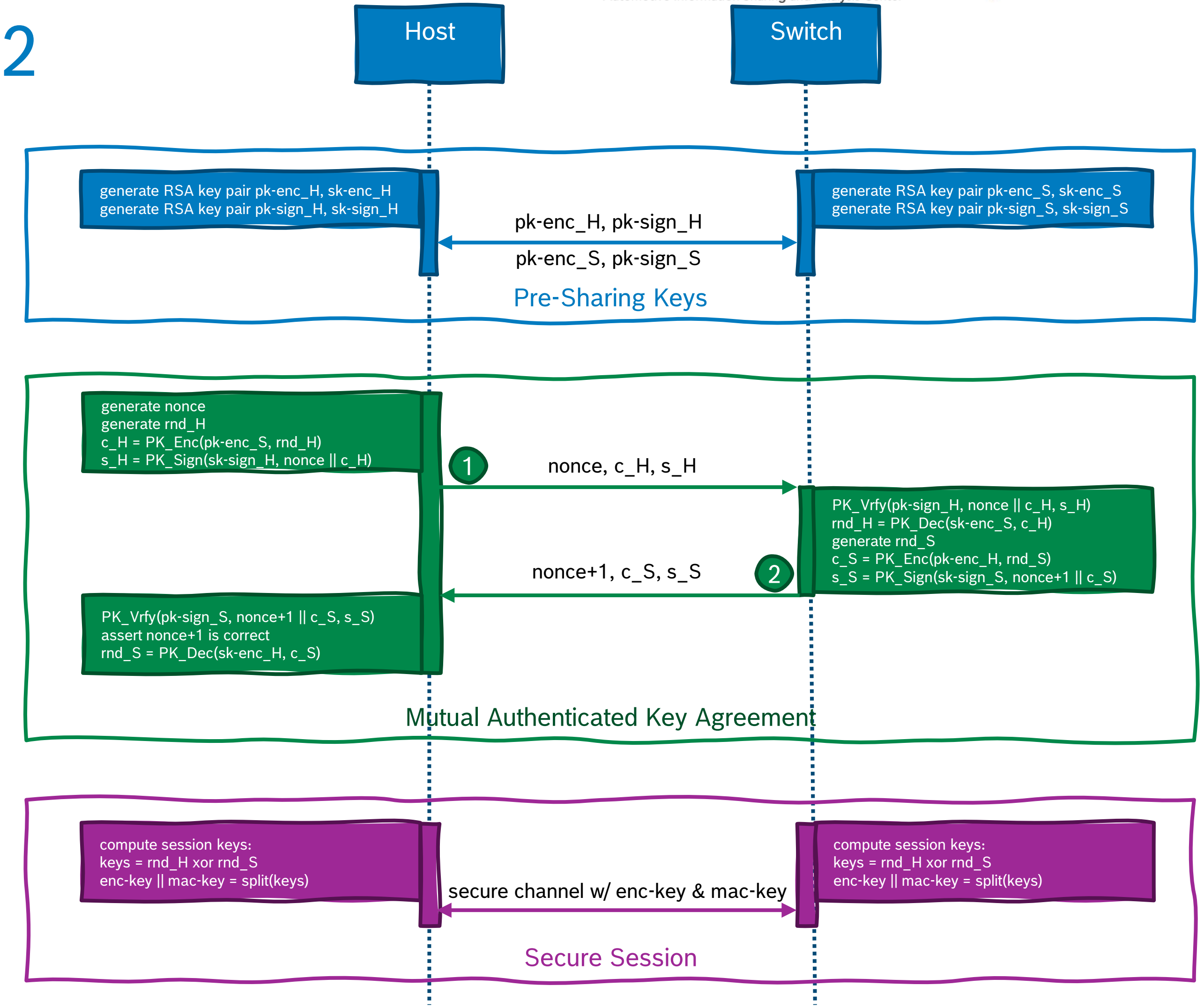
gcd = greatest common divisor

The SMEP Attack

Joint Development: SMEPv2

- Three parts of SMEP

1. Pre-sharing keys
   - Both host and switch generate two RSA 3072-bit key pairs, one for encryption, one for signing
   - These keys are pre-shared in a trusted environment
2. Mutual authenticated key agreement
   1. The host initiates the SMEP run by generating a random *nonce* and random number *rnd_H*. The random number *rnd_H* forms the host's contribution to the session key. For transmission, it therefore has to be confidential and is encrypted under the switch's public key *pk-enc_S* (resulting in the ciphertext *c_H*). To assert the authenticity of this random number, the ciphertext is then signed with the host's private signing key *sk-sign_H* (resulting in the signature *s_H*). Eventually, the host sends *nonce*, *c_H*, and *s_H* to the switch (1).
   2. The switch verifies the signature *s_H* using the host's public signature key *pk-sign_H*, and decrypt the random *rnd_H* using its own private decryption key *sk-enc_S*. The switch then generates a random number *rnd_S* that forms its contribution to the session key. As the host, the switch encrypts *rnd_S* with the help of *pk-enc_H* to the ciphertext *c_S* and signs it with his private signature key *sk-sign_S* to get the signature *s_S*. Finally, the switch responds with *nonce+1*, *c_S*, and *s_S* to the host's initial message (2).
   3. The host finally verifies the signature *s_S* with the switch's public signature key *pk-sign_S* and checks that the received *nonce+1* is correct (i.e., corresponds to his initial chosen *nonce+1*). Afterwards, the host decrypts the switch's random number *rnd_S* from the ciphertext *c_S* using his private decryption key *sk-enc_H*.
3. Secure channel / session
   - Both parties can now derive the session keys from the xor sum of the two random numbers *rnd_H* and *rnd_S*.



Host

Switch

generate RSA key pair pk-enc_H, sk-enc_H
generate RSA key pair pk-sign_H, sk-sign_H

generate RSA key pair pk-enc_S, sk-enc_S
generate RSA key pair pk-sign_S, sk-sign_S

pk-enc_H, pk-sign_H

pk-enc_S, pk-sign_S

Pre-Sharing Keys

generate nonce
generate rnd_H
c_H = PK_Enc(pk-enc_S, rnd_H)
s_H = PK_Sign(sk-sign_H, nonce || c_H)

1 nonce, c_H, s_H

PK_Vrfy(pk-sign_H, nonce || c_H, s_H)
rnd_H = PK_Dec(sk-enc_S, c_H)
generate rnd_S
c_S = PK_Enc(pk-enc_H, rnd_S)
s_S = PK_Sign(sk-sign_S, nonce+1 || c_S)

nonce+1, c_S, s_S 2

PK_Vrfy(pk-sign_S, nonce+1 || c_S, s_S)
assert nonce+1 is correct
rnd_S = PK_Dec(sk-enc_H, c_S)

Mutual Authenticated Key Agreement

compute session keys:
keys = rnd_H xor rnd_S
enc-key || mac-key = split(keys)

secure channel w/ enc-key & mac-key

compute session keys:
keys = rnd_H xor rnd_S
enc-key || mac-key = split(keys)

Secure Session

The SMEP Attack

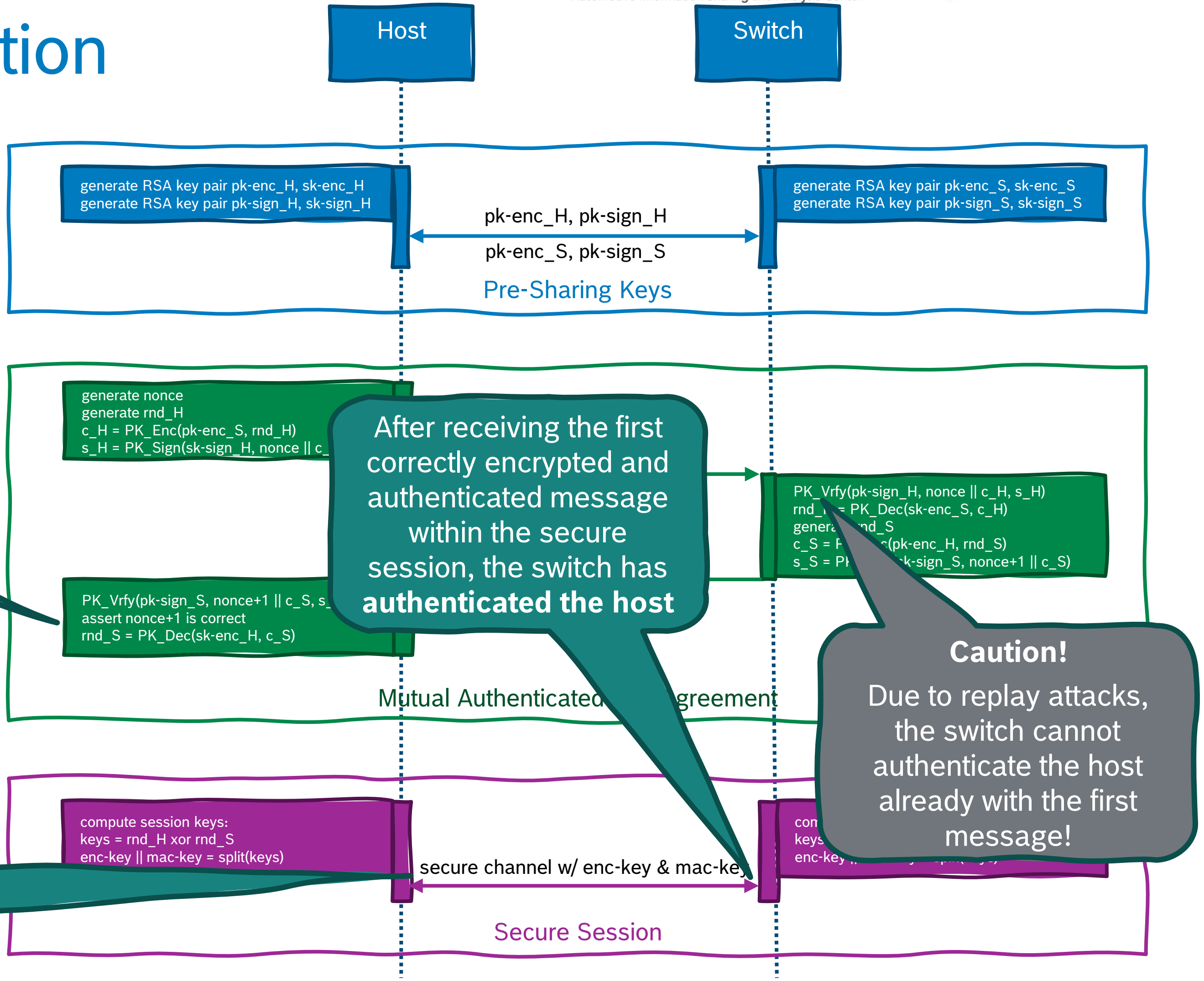
SMEPv2: Mutual Authentication

- Three parts of SMEP

1. Pre-sharing keys
   - Both host and switch generate two RSA 3072-bit key pairs, one for encryption, one for signing
   - These keys are pre-shared in a trusted environment
2. Mutual authenticated key agreement
   1. The host initiates th… d random
      number *rnd_H*. Th… tion to
      the session key. … and is
      encrypted under … iphertext
      c_H). To assert th… xt is then
      signed with the ho…
      signature *s_H*). Ev… the
      switch.
   2. The switch verifies the signature *s_H* using the host's public signa…
      pk-sign_H, and decrypt the random *rnd_H* using its own private decryptio…
      key *sk-enc_S*. The switch then generates a random number *rnd_S* that
      forms its contribution to the session key. As the host, the switch encrypts
      rnd_S with the help of *pk-enc_H* to the ciphertext *c_S* and signs it with his
      private signature key *sk-sign_S* to get the signature *s_S*. Finally, the switch
      responds with *nonce+1*, *c_S*, and *s_S* to the host's initial message.
   3. The host finally verifie… signature
      key *pk-sign_S* and ch… e.,
      corresponds to his ini… decrypts the
      switch's random num… private
      decryption key *sk-enc…*
3. Secure channel / se…
   - Both parties can now… n of the
     two random numbers…

After signature and nonce+1 verification, the host has **authenticated the switch**

The session keys are **confidential** and **authentic** and thus protect the established channel.


Host
Switch
generate RSA key pair pk-enc_H, sk-enc_H
generate RSA key pair pk-sign_H, sk-sign_H
pk-enc_H, pk-sign_H
pk-enc_S, pk-sign_S
generate RSA key pair pk-enc_S, sk-enc_S
generate RSA key pair pk-sign_S, sk-sign_S
Pre-Sharing Keys
generate nonce
generate rnd_H
c_H = PK_Enc(pk-enc_S, rnd_H)
s_H = PK_Sign(sk-sign_H, nonce || c_
PK_Vrfy(pk-sign_H, nonce || c_H, s_H)
rnd_ = PK_Dec(sk-enc_S, c_H)
genera nd_S
c_S = F c(pk-enc_H, rnd_S)
s_S = PK k-sign_S, nonce+1 || c_S)
PK_Vrfy(pk-sign_S, nonce+1 || c_S, s_
assert nonce+1 is correct
rnd_S = PK_Dec(sk-enc_H, c_S)
Mutual Authenticated greement
compute session keys:
keys = rnd_H xor rnd_S
enc-key || mac-key = split(keys)
com
keys
enc-key
secure channel w/ enc-key & mac-key
Secure Session

After receiving the first correctly encrypted and authenticated message within the secure session, the switch has **authenticated the host**

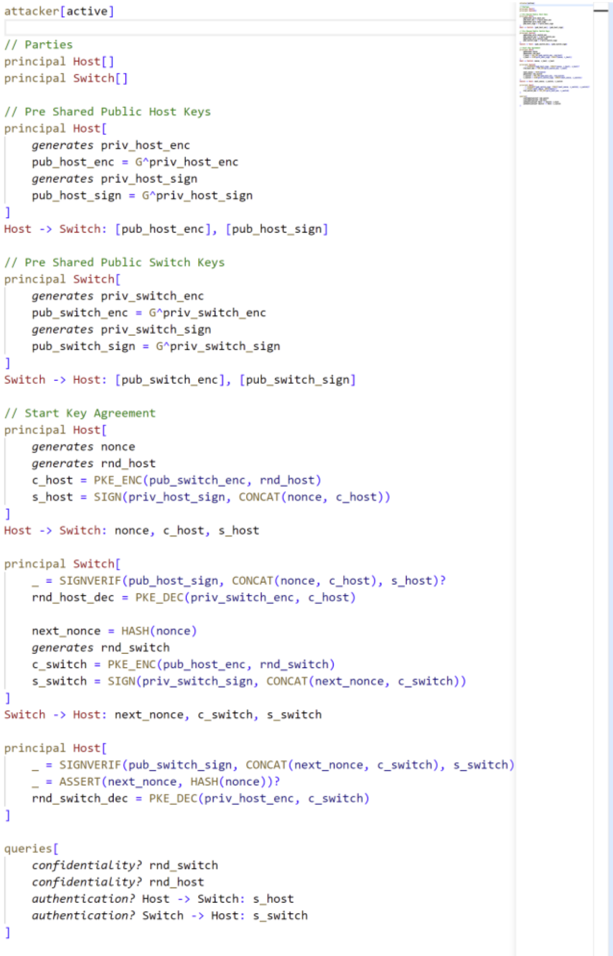
Caution!

Due to replay attacks, the switch cannot authenticate the host already with the first message!

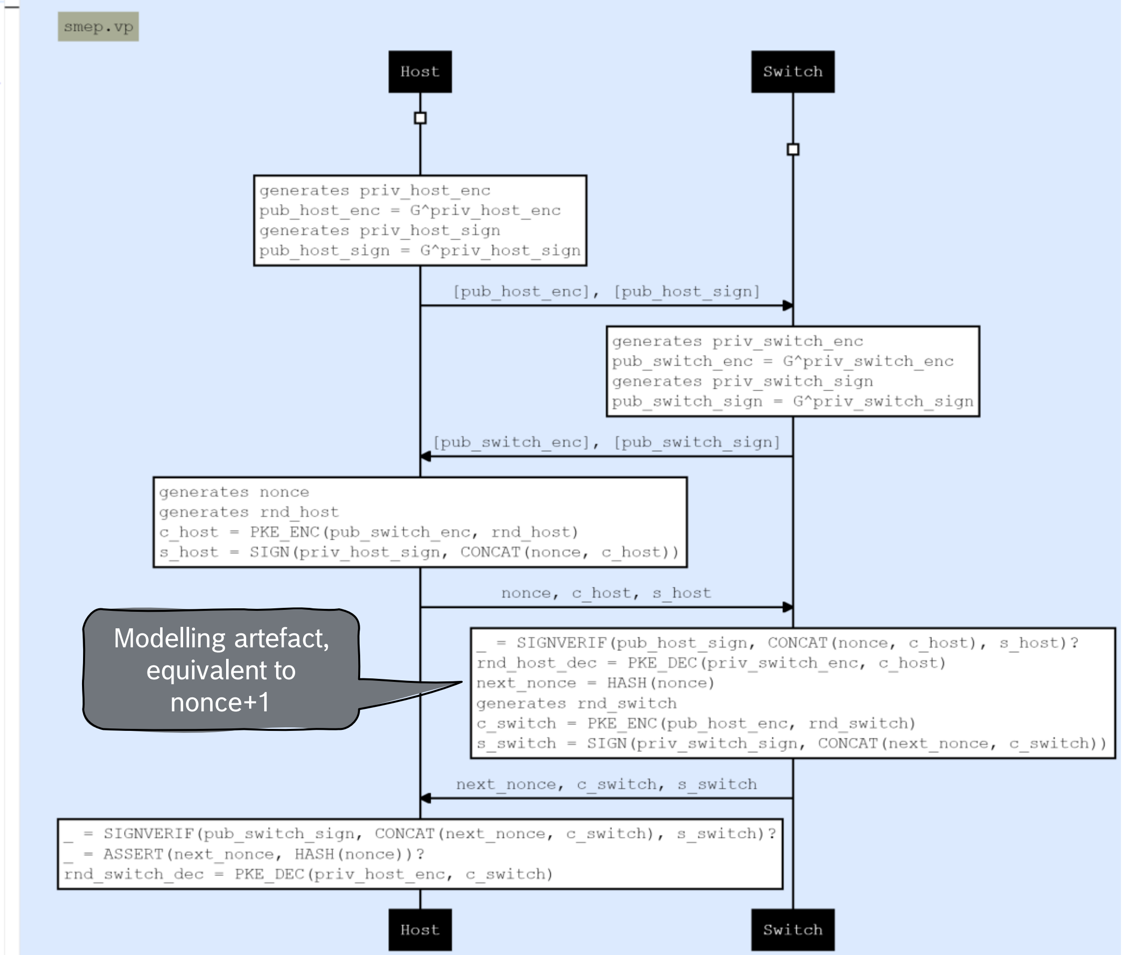
The SMEP Attack

SMEPv2: Formal Verification

- Verifpal (see https://verifpal.com/) model of the protocol
- Formally verifies the claimed security goals:
  - Confidentiality of *rnd_H*
  - Confidentiality of *rnd_S*
  - Authentication of host towards switch
  - Authentication of switch towards host
- Note:
  - Verifpal is currently beta software and not formally verified itself
  - This is not a mathematical proof
  - Due to modeling constraints, the switch does not send *nonce+1*, but *hash(nonce)*, which is semantically the same – however, due to efficiency reasons, *nonce+1* should be implemented



```
attacker[active]

// Parties
principal Host[]
principal Switch[]

// Pre Shared Public Host Keys
principal Host[
    generates priv_host_enc
    pub_host_enc = G^priv_host_enc
    generates priv_host_sign
    pub_host_sign = G^priv_host_sign
]
Host -> Switch: [pub_host_enc], [pub_host_sign]

// Pre Shared Public Switch Keys
principal Switch[
    generates priv_switch_enc
    pub_switch_enc = G^priv_switch_enc
    generates priv_switch_sign
    pub_switch_sign = G^priv_switch_sign
]
Switch -> Host: [pub_switch_enc], [pub_switch_sign]

// Start Key Agreement
principal Host[
    generates nonce
    generates rnd_host
    c_host = PKE_ENC(pub_switch_enc, rnd_host)
    s_host = SIGN(priv_host_sign, CONCAT(nonce, c_host))
]
Host -> Switch: nonce, c_host, s_host

principal Switch[
    _ = SIGNVERIF(pub_host_sign, CONCAT(nonce, c_host), s_host)?
    rnd_host_dec = PKE_DEC(priv_switch_enc, c_host)

    next_nonce = HASH(nonce)
    generates rnd_switch
    c_switch = PKE_ENC(pub_host_enc, rnd_switch)
    s_switch = SIGN(priv_switch_sign, CONCAT(next_nonce, c_switch))
]
Switch -> Host: next_nonce, c_switch, s_switch

principal Host[
    _ = SIGNVERIF(pub_switch_sign, CONCAT(next_nonce, c_switch), s_switch)
    _ = ASSERT(next_nonce, HASH(nonce))?
    rnd_switch_dec = PKE_DEC(priv_host_enc, c_switch)
]

queries[
    confidentiality? rnd_switch
    confidentiality? rnd_host
    authentication? Host -> Switch: s_host
    authentication? Switch -> Host: s_switch
]
```



```
generates priv_host_enc
pub_host_enc = G^priv_host_enc
generates priv_host_sign
pub_host_sign = G^priv_host_sign
```

Host → Switch: [pub_host_enc], [pub_host_sign]

```
generates priv_switch_enc
pub_switch_enc = G^priv_switch_enc
generates priv_switch_sign
pub_switch_sign = G^priv_switch_sign
```

Switch → Host: [pub_switch_enc], [pub_switch_sign]

```
generates nonce
generates rnd_host
c_host = PKE_ENC(pub_switch_enc, rnd_host)
s_host = SIGN(priv_host_sign, CONCAT(nonce, c_host))
```

Host → Switch: nonce, c_host, s_host

```
_ = SIGNVERIF(pub_host_sign, CONCAT(nonce, c_host), s_host)?
rnd_host_dec = PKE_DEC(priv_switch_enc, c_host)
next_nonce = HASH(nonce)
generates rnd_switch
c_switch = PKE_ENC(pub_host_enc, rnd_switch)
s_switch = SIGN(priv_switch_sign, CONCAT(next_nonce, c_switch))
```

Modelling artefact, equivalent to nonce+1

Switch → Host: next_nonce, c_switch, s_switch

```
_ = SIGNVERIF(pub_switch_sign, CONCAT(next_nonce, c_switch), s_switch)?
_ = ASSERT(next_nonce, HASH(nonce))?
rnd_switch_dec = PKE_DEC(priv_host_enc, c_switch)
```

Wrap up

Questions? Feedback?

> “Secure, or not secure,
> that is the question”

—freely adapted from Hamlet

Note

- Lesson learned: do not use textbook RSA, but properly padded versions of RSA (RSAES-OAEP, RSASSA-PSS, ...)
- Open Alliance TC19 is specifying a switch management framework, including secure access. Once this is available, this is hopefully the right solution to be used.