FLP Security

Franziska Obergfell

April 27, 2021

Contents

1 Introduction **3**

2 Sequence Diagrams **4**
2.1 Key Activation 4
2.2 Alarm Flag Reset 4
2.3 Dump Log 5
2.4 Erase Log 5
2.5 Expire SA 6
2.6 Key Deactivation 6
2.7 Key Inventory 7
2.8 Key Verification 7
2.9 Ping 7
2.10 Read ARSN 8
2.11 Rekey SA 9
2.12 SA Status Request 9
2.13 Set ARSN Window 10
2.14 Set ARSN 10
2.15 Start SA 11
2.16 Stop SA 11
2.17 TC Process Security 12
2.18 TM Apply Security 13

3 Class Diagrams **13**
3.1 Basic Functionality 13
3.2 External Interfaces 14
3.3 Key Management 15
3.4 SA Management 17
3.5 Security Log 20
3.6 Frame Processing 21
3.7 Security Management 23

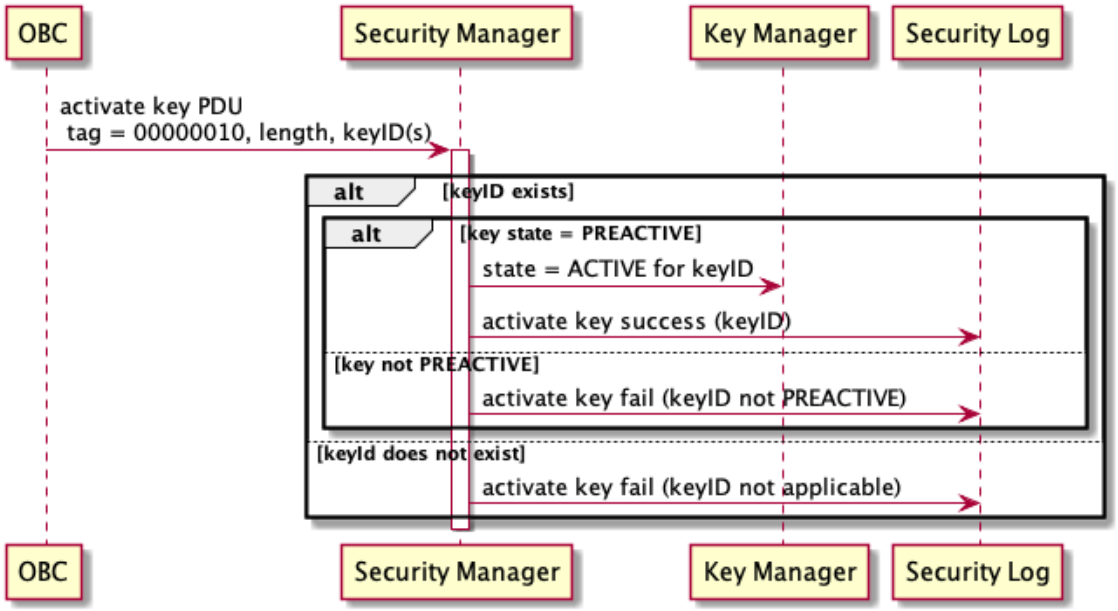
1 Introduction

In this document a possible structure and algorithmic sketch for an implementation of a security plugin for the FLP2 is proposed. It is to be noticed that some aspects might not be covered in full and some ambiguities might still be undiscovered. Also it is to be seen as a rather rough modeling of the software as I find it hard to model a lot of the details in my head on paper. Furthermore it is to be noted that the actual implementation might differ from the structure and modeling here and should be seen rather as an idea to group classes and procedures.

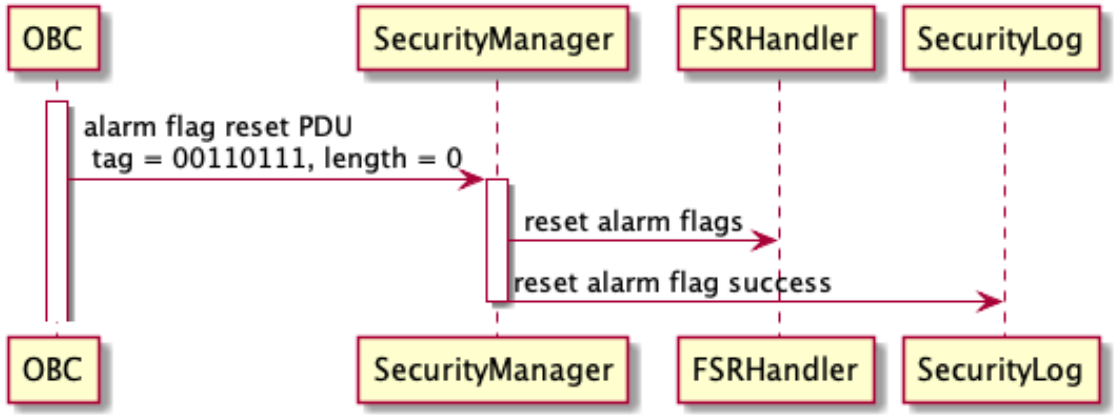
2 Sequence Diagrams

2.1 Key Activation

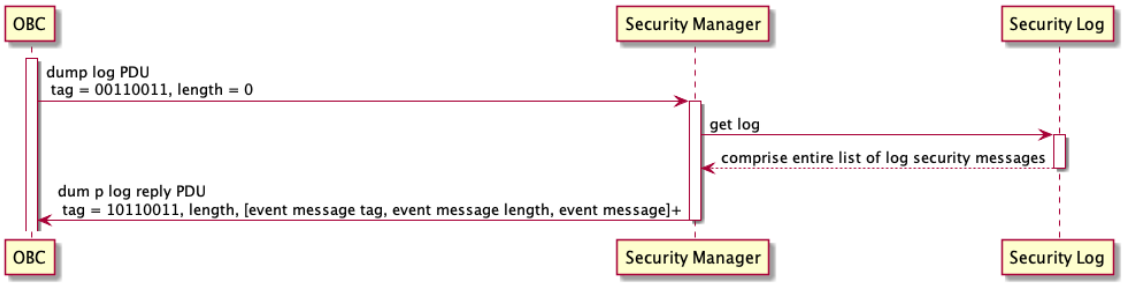
Software Components are shown as yellow boxes and all communication and action is shown via arrows. On reception of an activate key PDU from the Onboard Computer the Security Manager verifies if the key to be activated exists and is in the state PRE_ACTIVE as otherwise it can't be transitioned to ACTIVE. In this case the keys state is set to ACTIVE and a log message is wrote into the Security Log indicating the successful key activation. If one of the necessary conditions does not hold this is also logged in the Security Log indicating the error occurred.



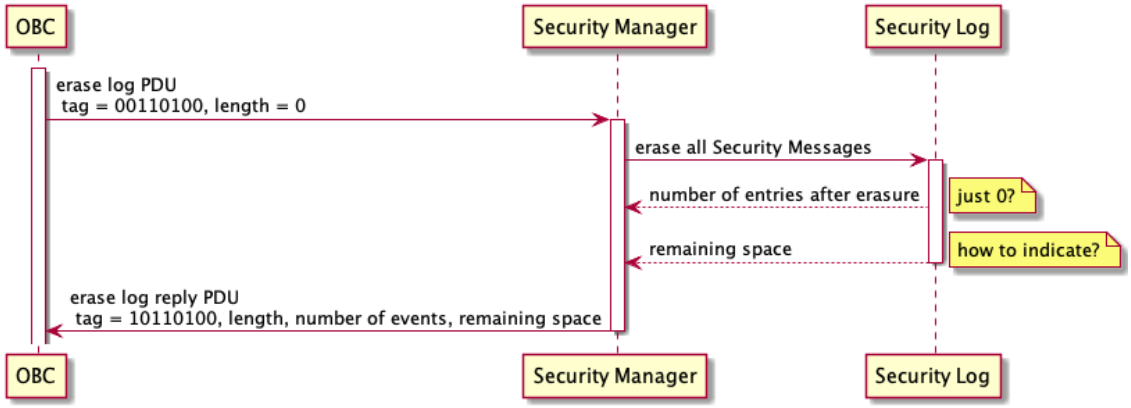
2.2 Alarm Flag Reset



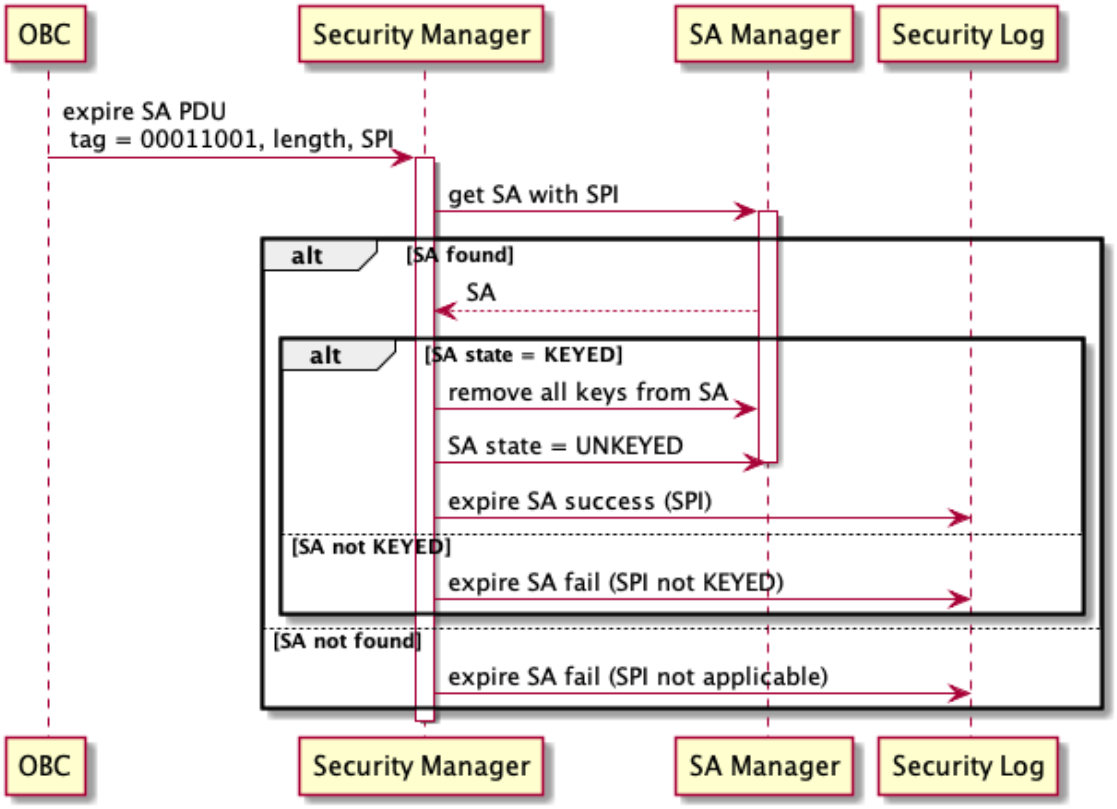
2.3 Dump Log



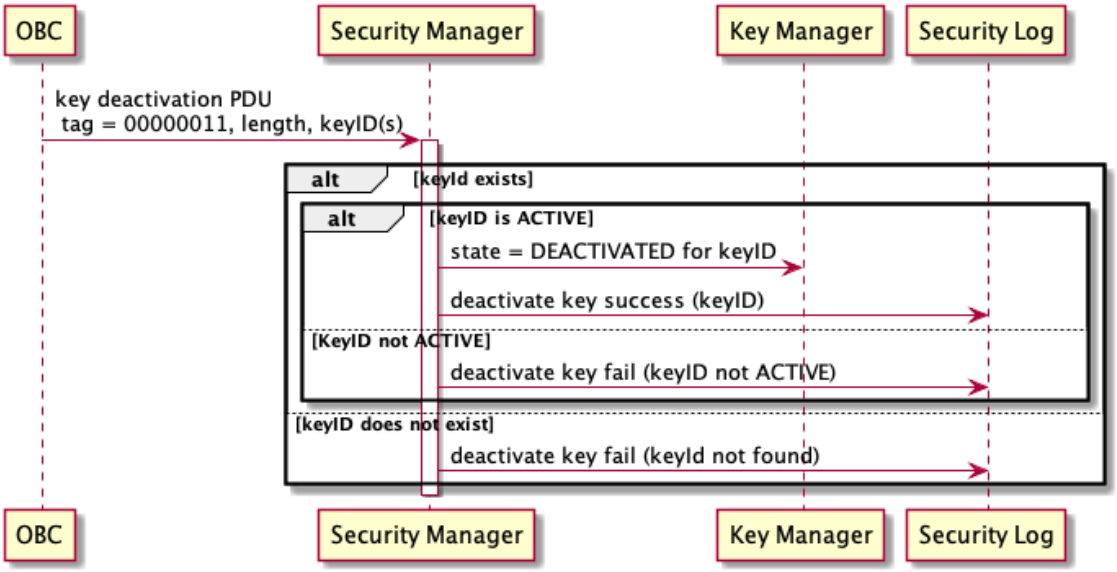
2.4 Erase Log



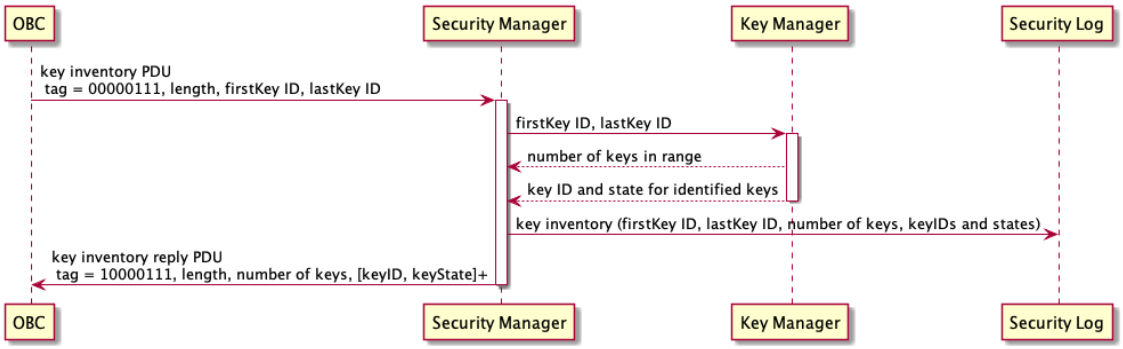
2.5 Expire SA



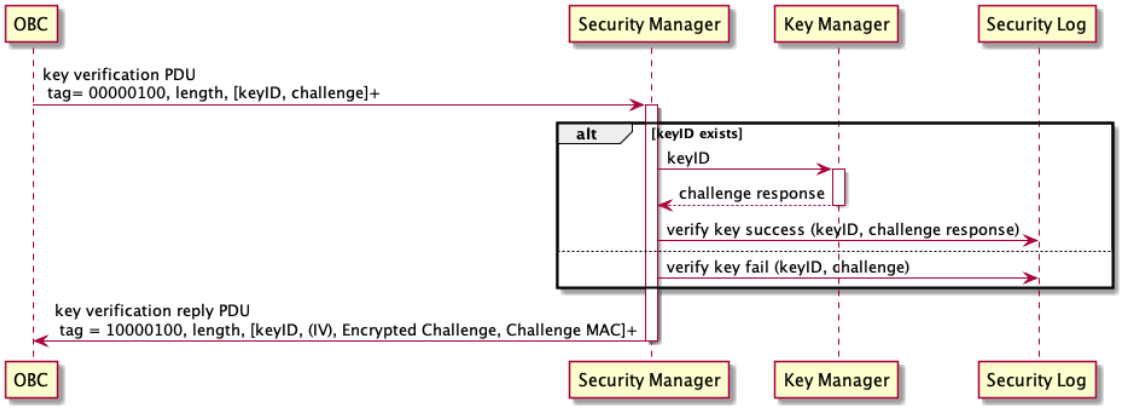
2.6 Key Deactivation



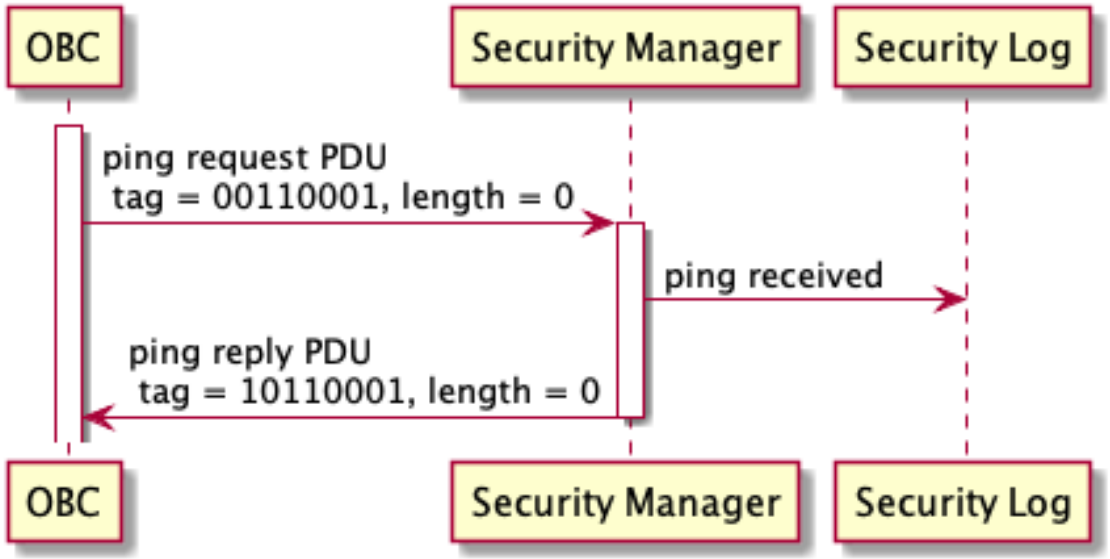
2.7 Key Inventory



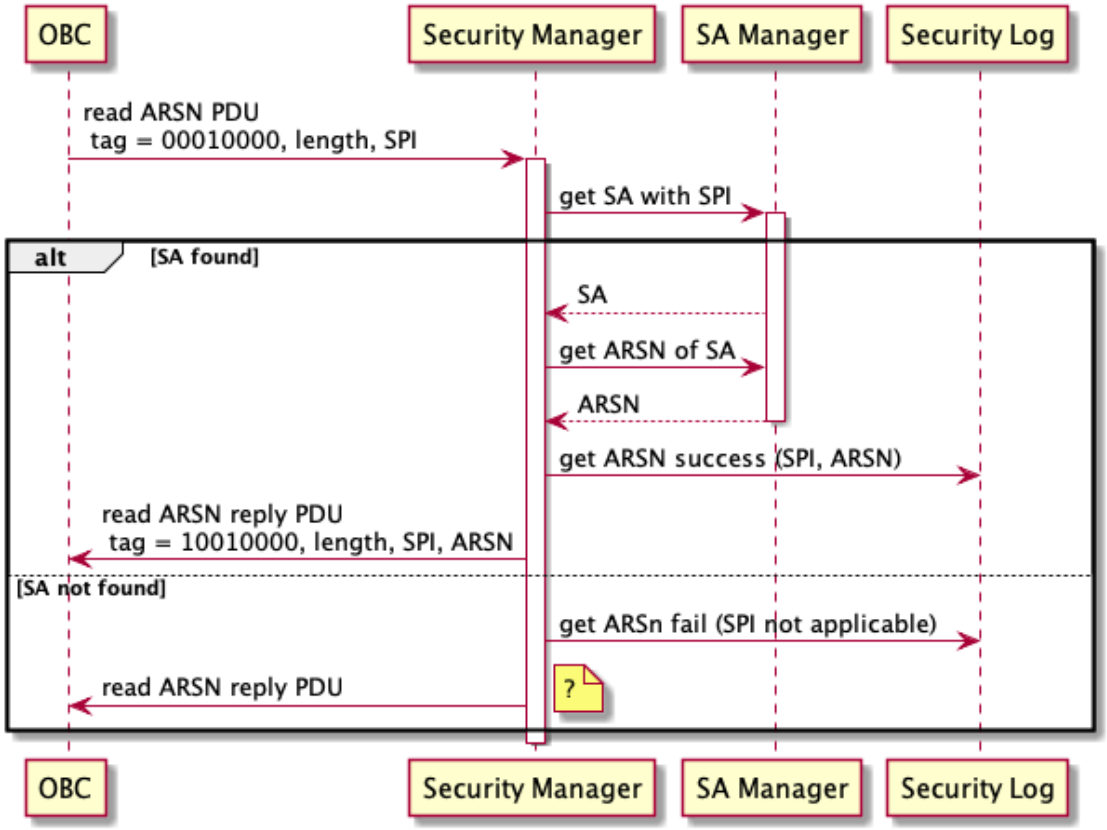
2.8 Key Verification



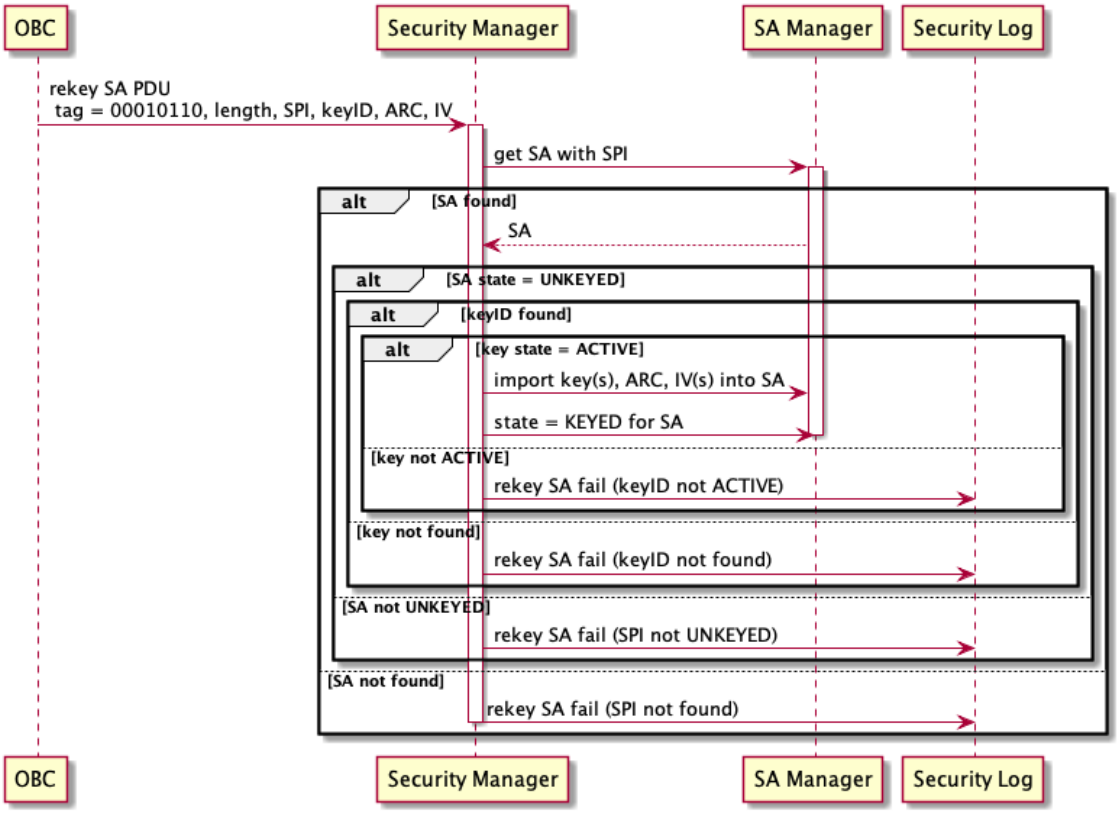
2.9 Ping



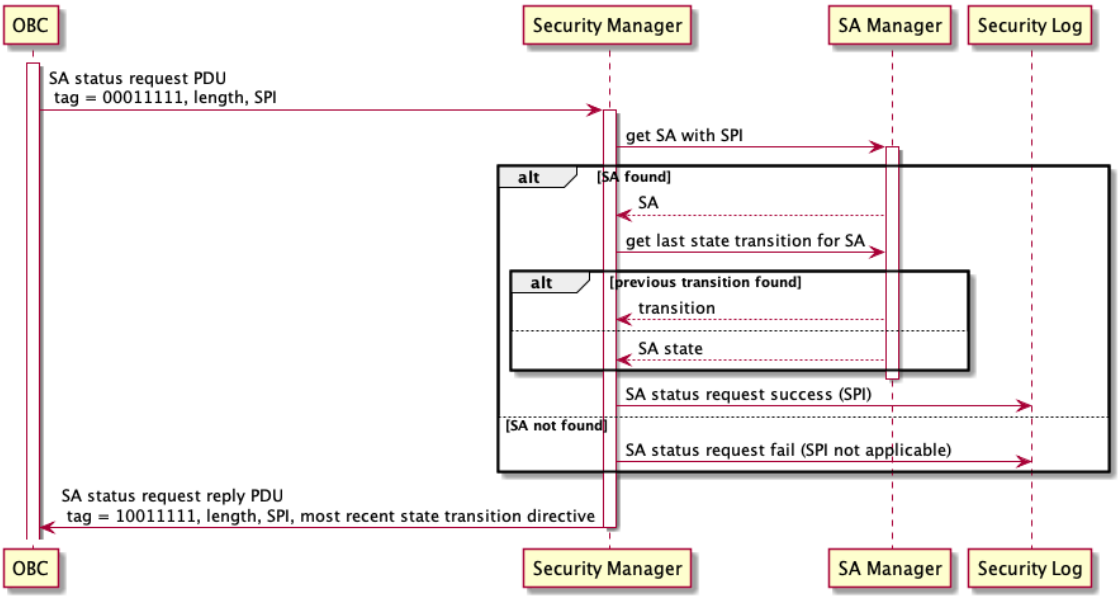
2.10 Read ARSN



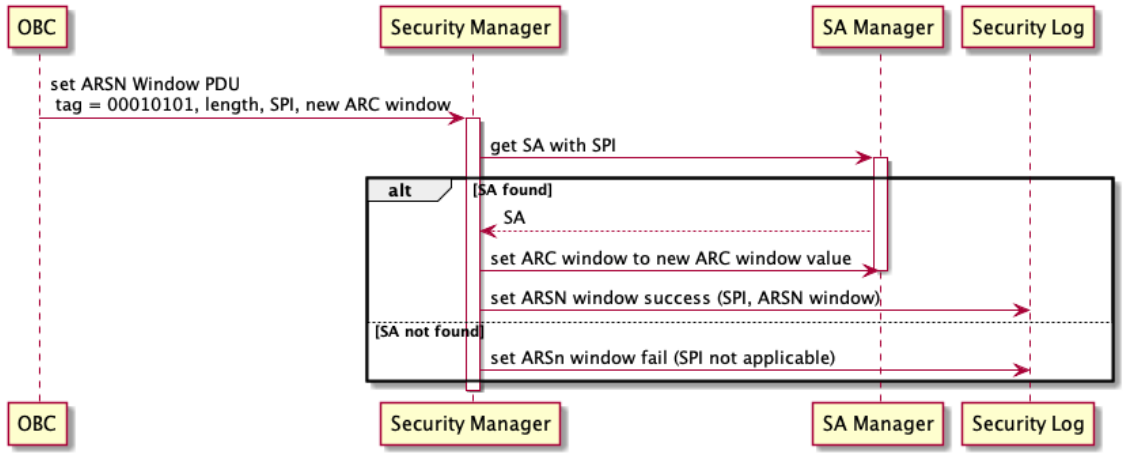
2.11 Rekey SA



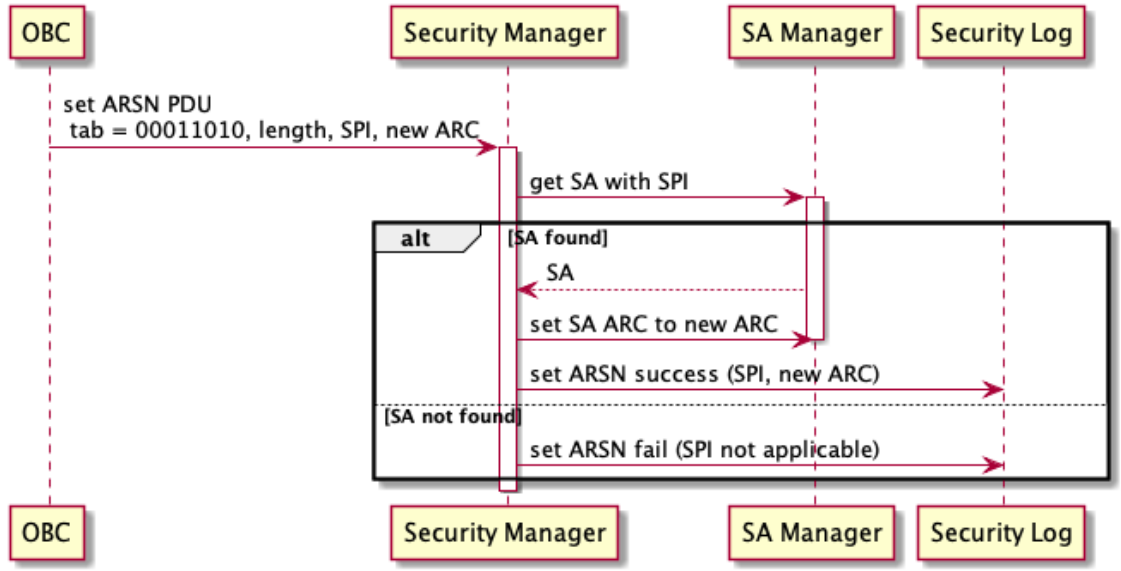
2.12 SA Status Request



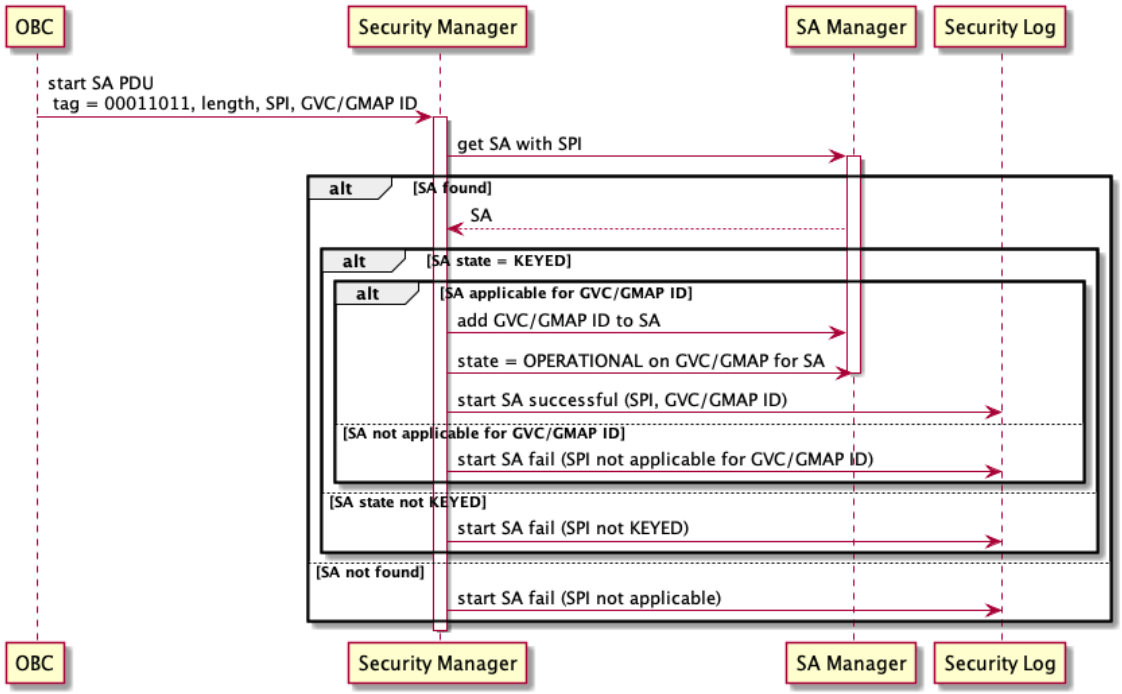
2.13 Set ARSN Window



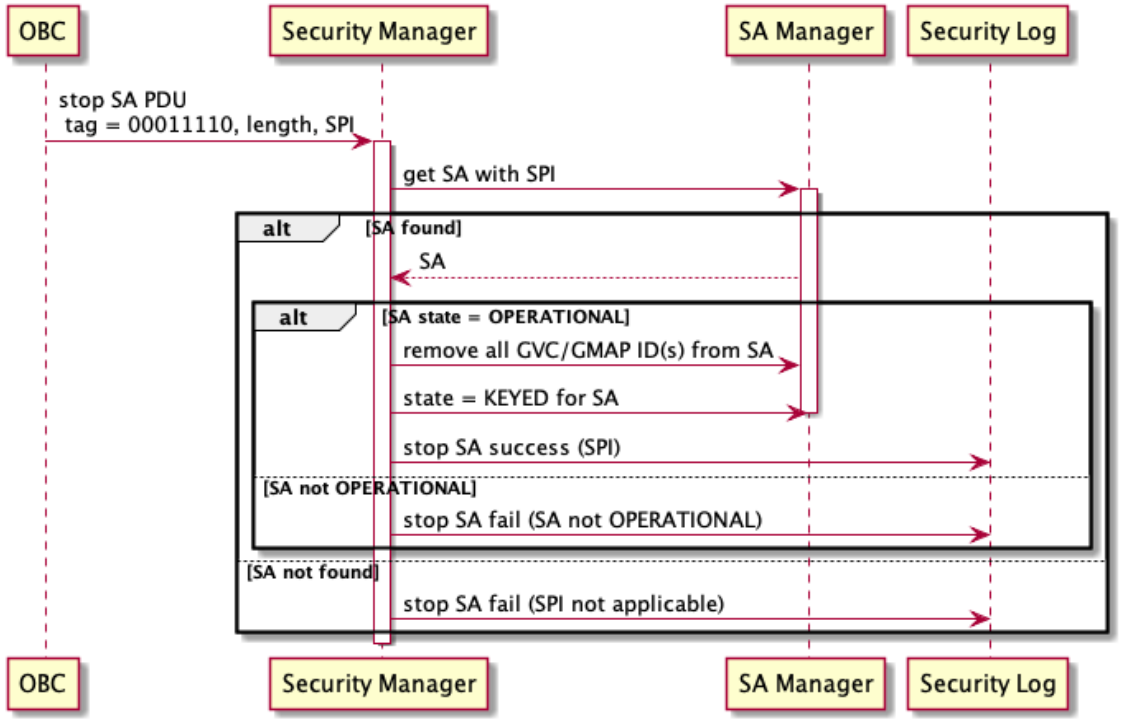
2.14 Set ARSN



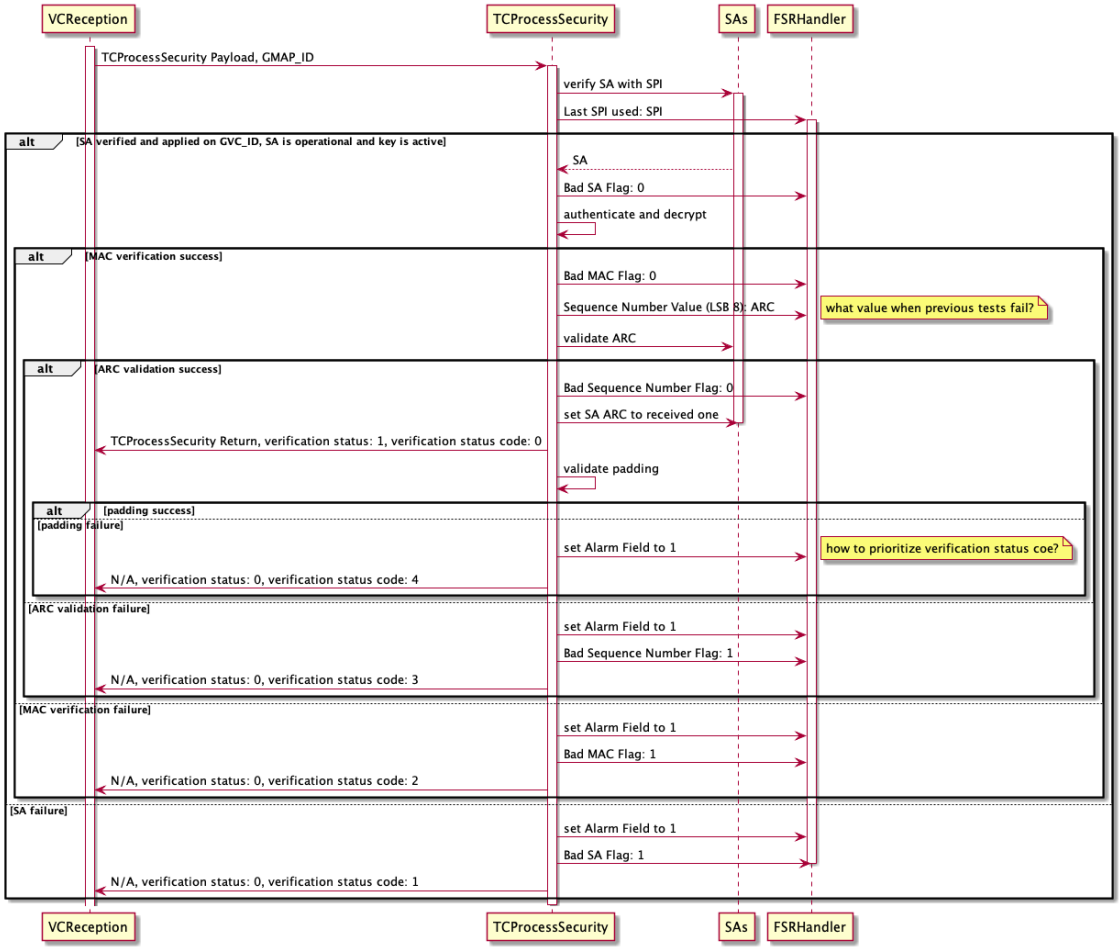
2.15 Start SA



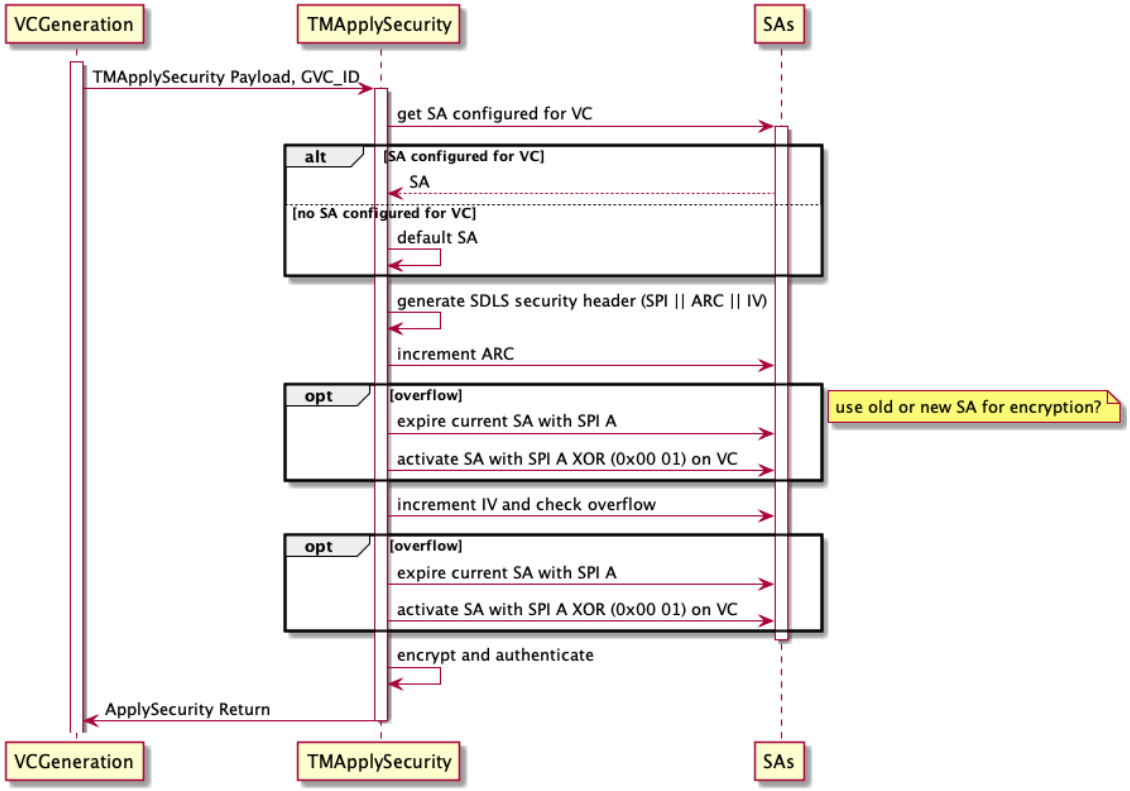
2.16 Stop SA



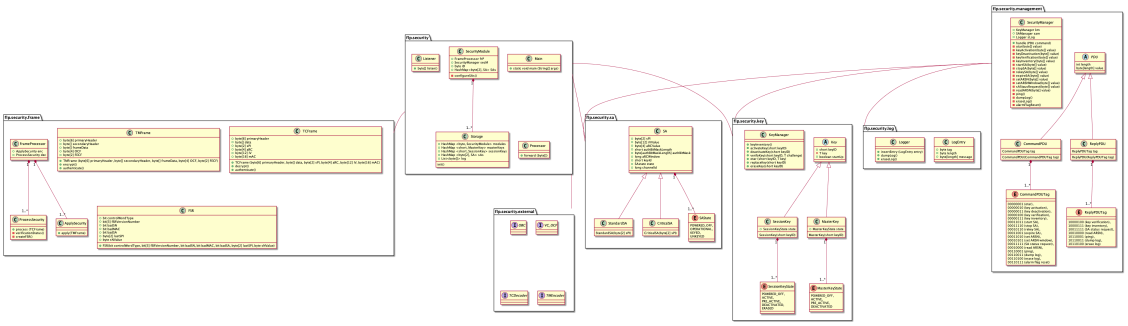
2.17 TC Process Security



2.18 TM Apply Security

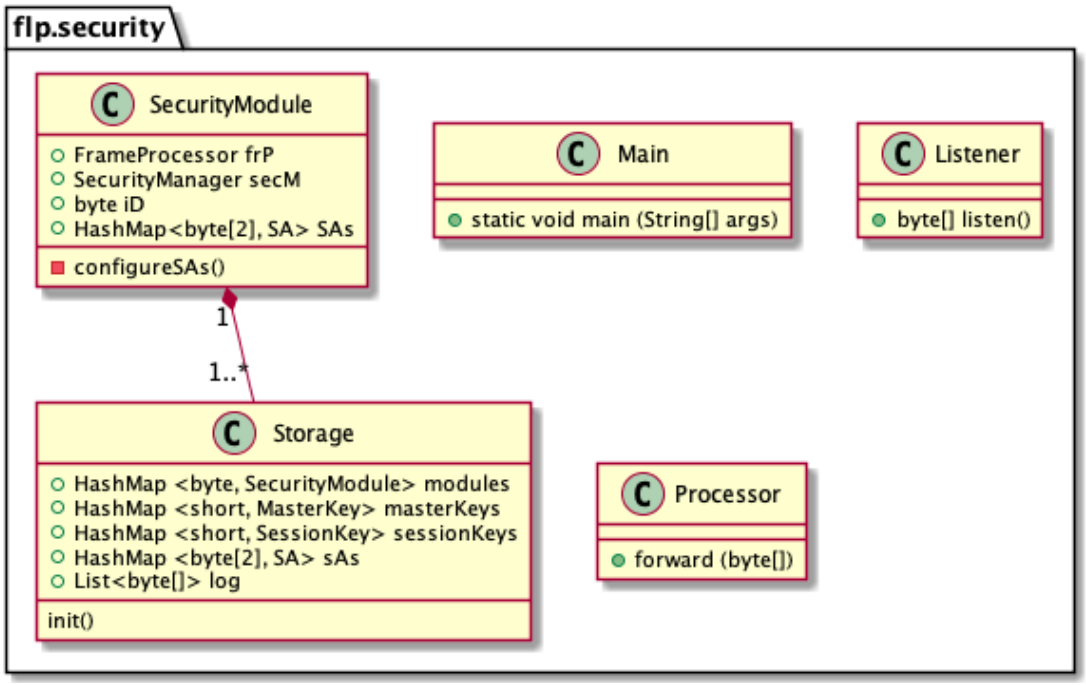


3 Class Diagrams



3.1 Basic Functionality

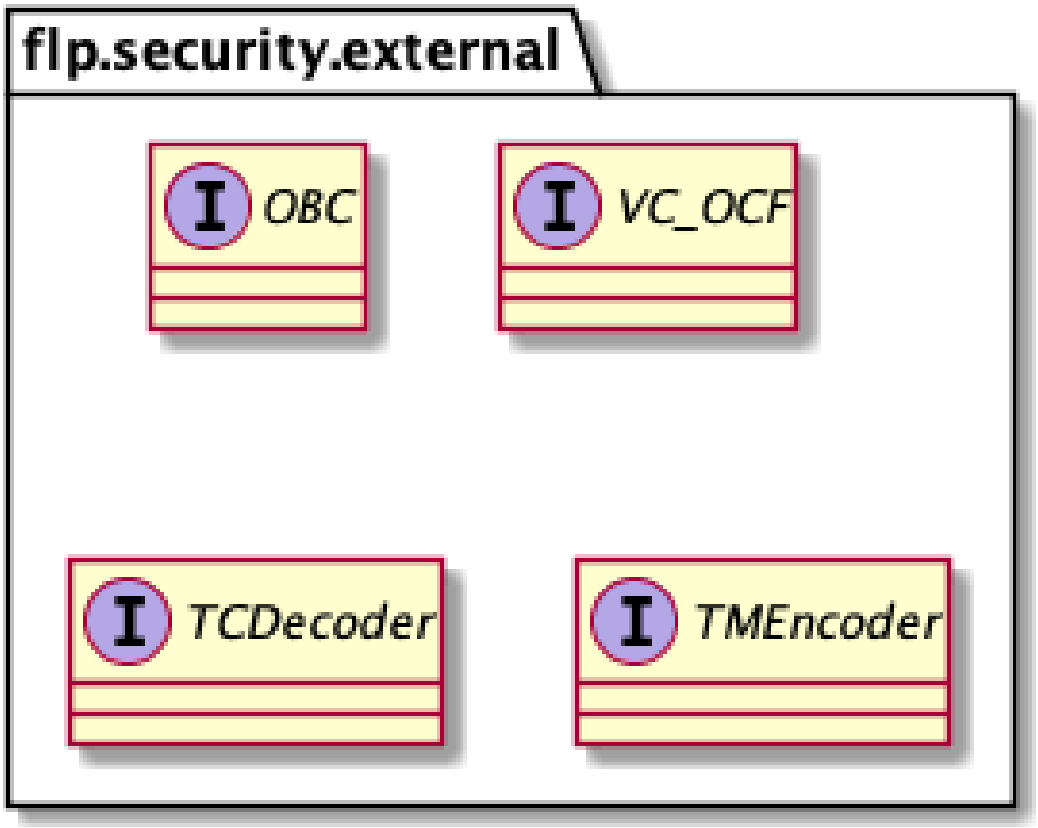
In this package the basic functionality of the satellites security module is handled. Concrete algorithmic structure is to be done as I find it hard to model it on paper. But here is at least a rough



idea to handle it.

3.2 External Interfaces

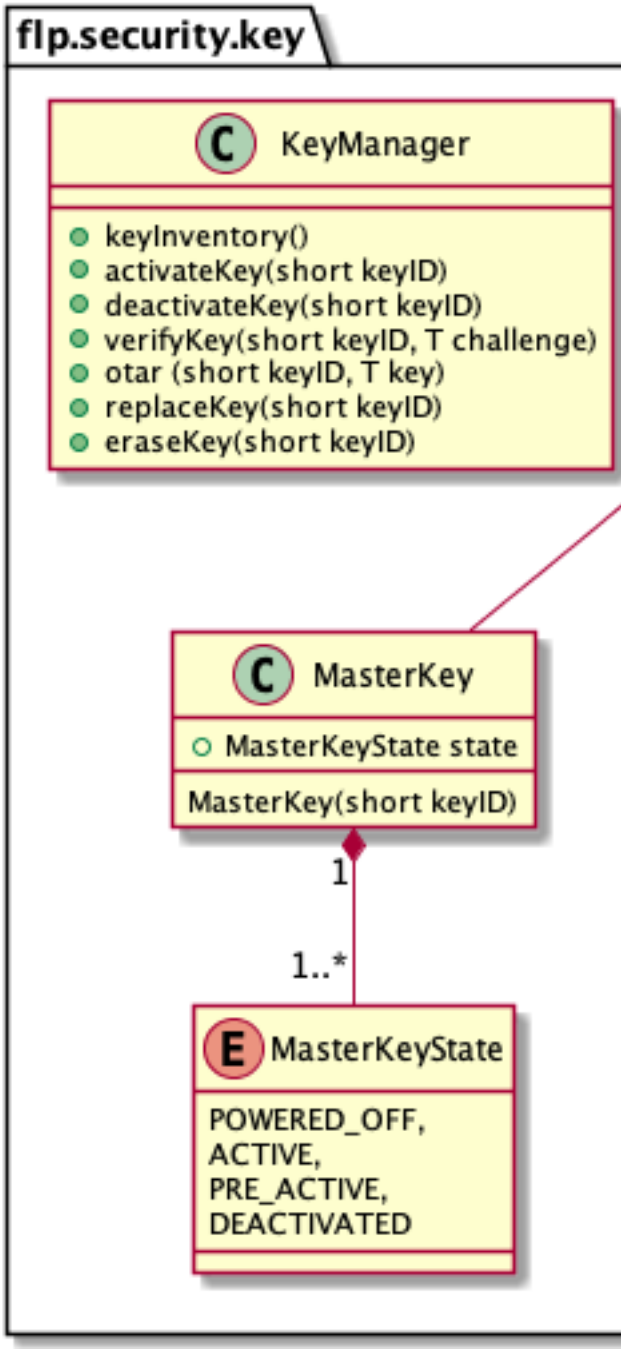
This package only contains some interfaces of the satellite outside the security plugin only. It is meant to group interfaces which need to be communicated with as seen mostly in the sequence dia-



grams.

3.3 Key Management

This package contains the functionality to manage keys and therefore mostly handles PDUs received from the ground in order to manage keys and their states.



Algorithm 1 keyInventory(short firstId, short lastId)

```
result ← ∅
amountOfKeys ← 0
count ← firstId
while count ¡= lastID do
    key ← Storage.getKey(count)
    if key != null then
        amountOfKeys++
        result.append(count, count.State)
    end if
    count++
end while
return amountOfKeys, result
```

Algorithm 2 activateKey(short keyId)

```
key ← Storage.getKey(keyId)
if key == null then
    return keyId not existent
end if
if key.State != PRE_ACTIVE then
    return keyId not PRE_ACTIVE
end if
key.setKeyState (ACTIVE)
keyId ACTIVE
```

Algorithm 3 dactivateKey(short keyId)

```
key ← Storage.getKey(keyId)
if key == null then
    return keyId not existent
end if
if key.State != ACTIVE then
    return keyId not ACTIVE
end if
key.setKeyState (DEACTIVATED)
keyId DEACTIVATED
```

Algorithm 4 verifyKey(short keyId, object challenge

```
key ← Storage.getKey(keyId)
if key == null then
    return keyId not existent
end if
response ← key.computeResponse(challenge)
return keyId, response
```

Algorithm 5 otar(short keyId, object key

```
1: if Storage.getKey(keyId) != null then
2:     return keyId already exists
3: end if
4: Key nKey = new Key(keyID)
5: nKey.set(key)
6: Storage.addKey(nKey)
7: return keyId
```

Algorithm 6 replaceKey(short keyID

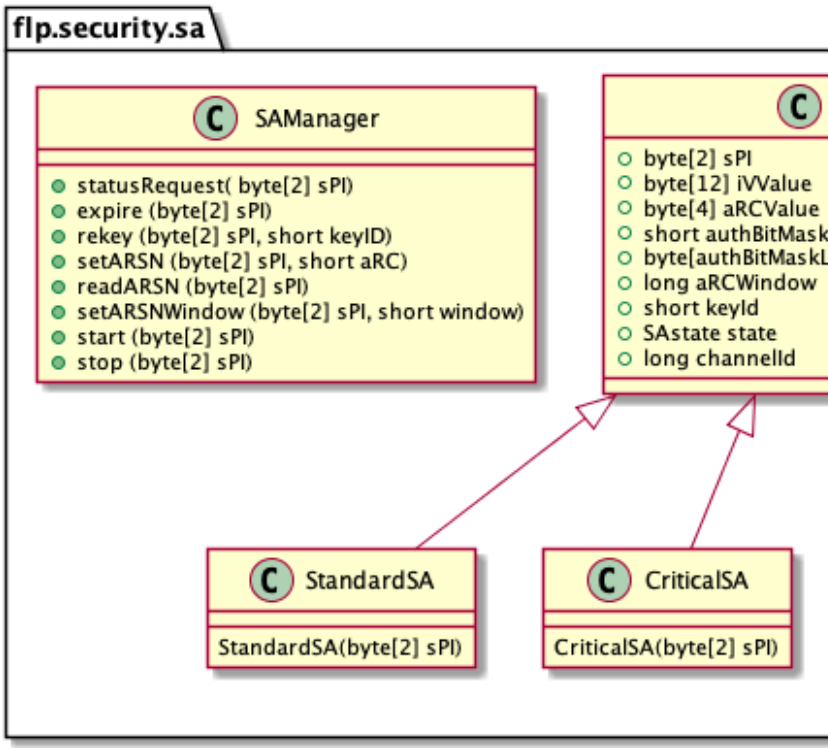
```
1: if Storage.removeKey == null then
2:     return false
3: end if
4: Storage.keys.append (keyId, Key.getKey(keyID))
5: return true
```

Algorithm 7 eraseKey(short keyID)

```
1: if Storage.getKey(keyID) == null then
2:     return false
3: end if
4: Storage.keys.remove(keyID)
5: return true
```

3.4 SA Management

Similar to the previous package this one is responsible for the management of Security Associations and handles PDUs received from the ground which were forwarded by Security Management to



handle managed parameters and states of SAs.

Algorithm 8 statusRequest(byte[2] sPI)

```
1: sa = Storage.getSA(sPI)
2: if sa == null then return sPI does not exist
3: end if
4: SAstate curr = sa.getStete
5: SAstate prev ) sa.getPrevState
6: if prev != null then
7:     return prev, curr
8: end if
9: return curr
```

Algorithm 9 expire(byte[2] sPI)

```
1: sa = Storage.getSA(sPI)
2: if sa == null then
3:     return sPI not existent
4: end if
5: if sa.getState != KEYED then
6:     return sPI not KEYED
7: end if
8: sa.setState(UNKEYED)
9: sPI UNKEYED
```

Algorithm 10 rekey(byte[2] sPI, short keyID)

```
1: sa = Storage.getSA(sPI)
2: if sa == null then
3:     return sPI does not exist
4: end if
5: if sa.getState != UNKEYED then
6:     return sPI not UNKEYED
7: end if
8: key = Storage.getKey(keyID)
9: if key == null then
10:    return keyID does not exist
11: end if
12: if key.getState != ACTIVE then
13:    return keyID not ACTIVE
14: end if
15: sa.addKey(keyID)
16: sa.setState(KEYED)
17: return sPI KEYED with keyID
```

Algorithm 11 setARSN(byte[2] sPI, short aRC

```
1: sa = Storage.getSA(sPI)
2: if sa == null then
3:     return sPI does not exist
4: end if
5: if sa.getARCWindow < aRC then
6:     return aRC invalid
7: end if
8: sa.setARC(aRC)
9: return sPI set ARSN to aRC
```

Algorithm 12 readARSN(byte[2] sPI)

```
1: sa = Storage.getSA(sPI)
2: if sa == null then
3:     return sPI does not exist
4: end if
5: return sa.getARC
```

Algorithm 13 setARSNWindow(bytte[2] sPI, short window)

```
1: sa = Storage.getSA(sPI)
2: if sa == null then
3:     return sPI does not exist
4: end if
5: sa.setARCWindow(window)
6: return true
```

Algorithm 14 start(byte[2] sPI)

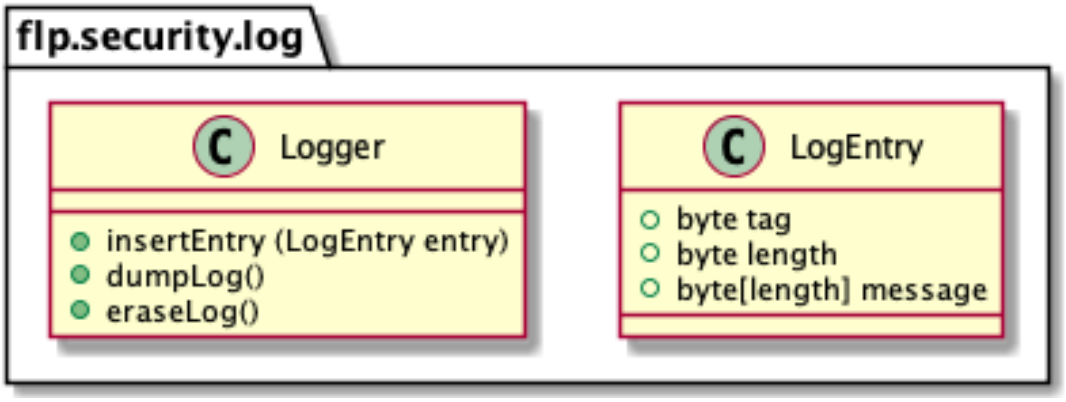
```
1: sa = Storage.getSA(sPI)
2: if sa == null then
3:     return sPI does not exist
4: end if
5: if sa.getState != KEYED then
6:     return sPI not KEYED
7: end if
8: if ! getChannel().contains(sPI) then
9:     return sPI not applicable for channel
10: end if
11: sa.setState(OPERATIONAL, getChannel)
12: return true
```

Algorithm 15 stop(byte[2] sPI)

```
1: sa = Storage.getSA(sPI)
2: if sa == null then
3:     return sPI does not exist
4: end if
5: if sa.getState != OPERATIONAL then
6:     return sPI not OPERATIONAL
7: end if
8: sa.setState(KEYED)
9: return true
```

3.5 Security Log

This package is used to implement a Security Log as a mean of logging actions of the security management of the space segment. As errors are mostly not directly handled by the space segment and also successive actions are not always directly transmitted to the ground, the log can be used to collect the behaviour of the security management and later be read by requesting the log with a Dump Log PDU.



Algorithm 16 insertEntry(LogEntry entry)

```
1: Storage.getLog.append(entry)
```

Algorithm 17 dumpLog()

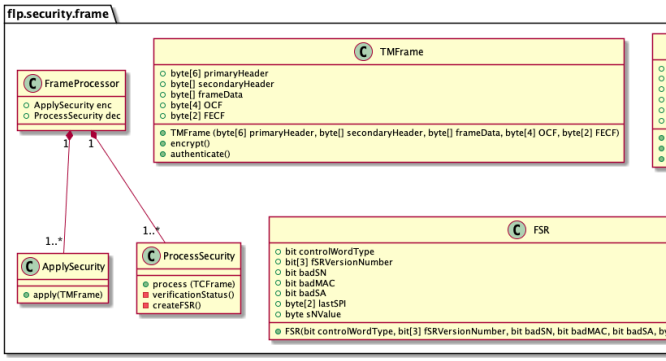
```
1: return Storage.getLog()
```

Algorithm 18 eraseLog()

```
1: for entry in Storage.getLog() do
2:     Storage.Log.remove(entry)
3:     count = 0
4:     for entry in Storage.Log do count ++
5:     end for
6: end for
7: return count, Storage.Log.getSize()
```

3.6 Frame Processing

This package implements the decryption, encryption and authentication of frames received and to be sent. The concrete implementation is only sketched here as it also occurred to me hard to model on paper and might change in the actual code implementation.



Algorithm 19 TMFrame.encrypt()

```
channel = extractID()
sa = channel.getSA()
if sa == null then
    sa = getDefaultSA()
end if
secHeader = sa.sPI —— sa.getARC —— sa.getIV
old = getARC()
sa.setARC(old + 1)
if ARC overflow then
    sPI = sa.sPI
    SAManager.expire(sPI)
    SAManager.activate(sPI XOR (0x00 01), channel)
end if
iv = sa.getIV()
sa.setIV(iv + 1)
if IV overflow then
    sPI = sa.sPI
    SAManager.expire(sPI)
    SAManager.activate(sPI XOR (0x00 01), channel)
end if
authenticate = Headers XOR AuthBitmask
ea = apply AES-GCM 256 on Data and autehnticate
return ea as byte[]
```

Algorithm 20 ApplySecurity.apply()

```
frame.encrypt()
```

Algorithm 21 ProcessSecurity.process(TCFrame frame)

```
frame.decrypt()
verificationStatus()
createFSR()
```

Algorithm 22 TCFrame.decrypt()

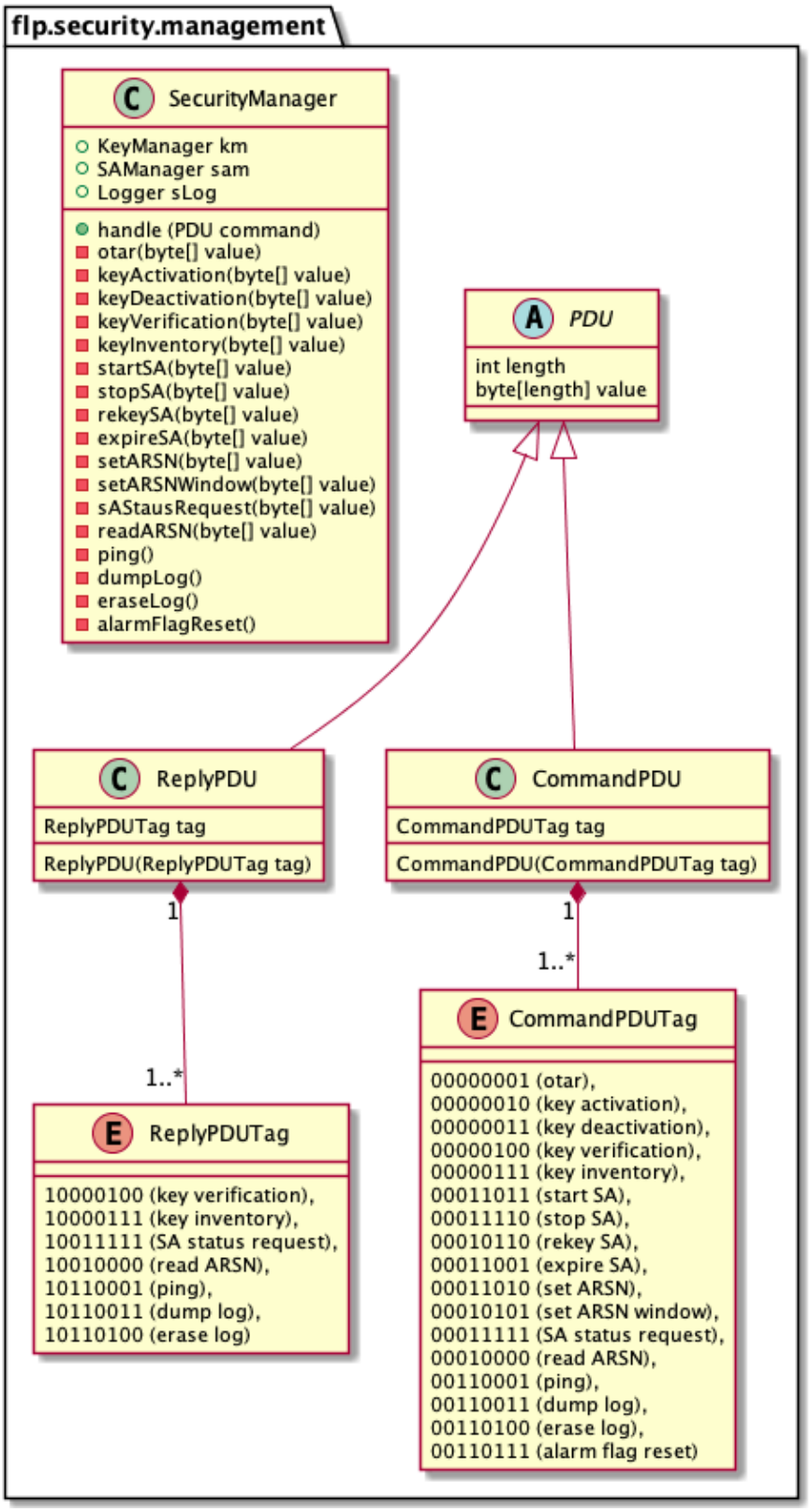
```
sPI = extractSPI()
sa = Storage.getSA(sPI)
badSAFlag = 0
if sa == null OR sa != channel.SA OR sa.getState != OPERATIONAL then
    badSAFlag = 1
    alarmField = 1
end if
badMACFlag = 0
remove AES-GCM 256
if failure then
    badMACFlag = 1
    alarmField = 1
end if
badSNFlag = 0
validateARC
if failure then
    badSNFlag = 1
    alarmField = 1
end if
FSR(badSNFlag, badMACFlag, badSAFlag,lastSPI, sa.ARC)
```

Algorithm 23 verificationStatus()

```
if failure in decryption then
    return 0
end if
return 1, verificationStatusCode()
```

3.7 Security Management

This package is responsible for all security management functions and therefore receives all commands received as PDUs from the ground. It is responsible for forwarding it to appropriate other packages and classes to handle and carry out the services requested. Therefore it can receive any PDU and determines what to do next and forward to the appropriate other packages.


flp.security.management
SecurityManager
KeyManager km
SAManager sam
Logger sLog
handle (PDU command)
otar(byte[] value)
keyActivation(byte[] value)
keyDeactivation(byte[] value)
keyVerification(byte[] value)
keyInventory(byte[] value)
startSA(byte[] value)
stopSA(byte[] value)
rekeySA(byte[] value)
expireSA(byte[] value)
setARSN(byte[] value)
setARSNWindow(byte[] value)
sAStausRequest(byte[] value)
readARSN(byte[] value)
ping()
dumpLog()
eraseLog()
alarmFlagReset()
PDU
int length
byte[length] value
ReplyPDU
ReplyPDUTag tag
ReplyPDU(ReplyPDUTag tag)
CommandPDU
CommandPDUTag tag
CommandPDU(CommandPDUTag tag)
1
1..*
1
1..*
ReplyPDUTag
10000100 (key verification),
10000111 (key inventory),
10011111 (SA status request),
10010000 (read ARSN),
10110001 (ping),
10110011 (dump log),
10110100 (erase log)
CommandPDUTag
00000001 (otar),
00000010 (key activation),
00000011 (key deactivation),
00000100 (key verification),
00000111 (key inventory),
00011011 (start SA),
00011110 (stop SA),
00010110 (rekey SA),
00011001 (expire SA),
00011010 (set ARSN),
00010101 (set ARSN window),
00011111 (SA status request),
00010000 (read ARSN),
00110001 (ping),
00110011 (dump log),
00110100 (erase log),
00110111 (alarm flag reset)

Algorithm 24 handle(PDU command

```
switch(command.getTag())
case 00000001: otar(command.getValue())
case 00000010: keyActivation(command.getValue())
case 00000011: keyDeactivation(command.getValue())
case 00000100: keyVerification(command.getValue())
case 00000111: keyInventory(command.getValue())
case 00011011: startSA(command.getValue())
case 00011110: stopSA(command.getValue())
case 00010110: rekeySA(command.getValue())
case 00011001: expireSA(command.getValue())
case 00011010: setARSN(command.getValue())
case 00010101: setARSNWindow(command.getValue())
case 00011111: statusRequest(command.getValue())
case 00010000: readARSN(command.getValue())
case 00110001: ping()
case 00110011: dumpLog()
case 00110100: eraseLog()
case 00110111: alarmFlagReset()
```

Algorithm 25 otar(byte[] value)

```
content = decrypt(value)
i = 0
while i ¡ content.length do
    keyId = parseKeyID(content, i)
    i = i + 2
    key = parseKey(content, i)
    i = i + 32
    status = km.otar(keyId, key)
    if status = keyID already exists then
        sLog.insertEntry(LogEntry(otar, keyId, fail))
    end if
    if status = keyID then
        sLog.insertEntry(LogEntry(otar, keyID, success))
    end if
end while
```

Algorithm 26 keyActivation(byte[] value

```
i = 0
while i ¡ value.length do
    keyId = parseKeyId(value, i)
    i = i + 2
    status = km.activateKey(keyId)
    if status = keyId not existent then
        sLog.insertEntry(LogEntry(keyActivation, keyId, fail, not found))
    end if
    if status = keyId not PRE_ACTIVE then
        sLog.insertEntry(LogEntry(keyActivation, keyId, fail, not PRE_ACTIVE))
    end if
    if status = keyId ACTIVE then
        sLog.insertEntry(LogEntry(keyActivation, keyId, success))
    end if
end while
```

Algorithm 27 keyDeactivation(byte[] value)

```
i = 0
while i ¡ value.length do
    keyId = parseKeyId(value, i)
    i = i + 2
    status = km.deactivateKey(keyId)
    if status = keyId not existent then
        sLog.insertEntry(LogEntry(keyDeactivation, keyId, fail, not found))
    end if
    if status = keyId not ACTIVE then
        sLog.insertEntry(LogEntry(keyDeactivation, keyId, fail, not ACTIVE))
    end if
    if status = keyId DEACTIVATED then
        sLog.insertEntry(LogEntry(keyDeactivation, keyId, success))
    end if
end while
```

Algorithm 28 keyVerification(byte[] value)

```
i = 0 = ∅
while i ¡ value.length do
    keyID = parseKeyId(value, i)
    i = i + 2
    challenge = parseChallenge(value, i)
    i = i + ?
    status = km.verifyKey(keyId, challenge)
    if status = keyId not existent then
        sLog.insertEntry(LogEntry(keyVerification, keyId, fail, not found))
    end if
    if status = keyId, response then
        sLog.insertEntry(LogEntry(keyVerification, keyId, success))
        reply.append(keyId)
        reply.append(response.encrypted())
    end if
end while
ReplyPDU result(10000100)
result.setLength(reply.length)
result.setValue(reply)
return result
```

Algorithm 29 keyInventory(byte[2] value)

```
firstId = parseKeyId(value, 0)
lastId = parseKeyId(value, 2)
status = km.keyInventory(firstId, lastId)
sLog.insertEntry(LogEntry(keyInventory, firstId, lastId))
ReplyPDU answer(10000111)
answer.setLength(status.length)
answer.setValue(status)
return answer
```

Algorithm 30 startSA(byte[] value

```
i = 0
while i ¡ value.length do
    sPI = parseSPI(value, i)
    i = i + 2
    message = sam.start(sPI)
    if message = sPI does not exist then
        sLog.insertEntry(LogEntry(startSA, sPI, fail, not found))
    end if
    if message = sPI not KEYED then
        sLog.insertEntry(LogEntry(startSA, sPI, fail, not KEYED))
    end if
    if message = sPI not appicable for channel then
        sLog.insertEntry(LogEntry(startSA, SPI, fail, not applicable))
    end if
    if message = true then
        sLog.insertEntry(LogEntry(startSA, sPI, success))
    end if
end while
```

Algorithm 31 stopSA(byte[] value

```
i = 0
while i ¡ value.length do
    sPI = parseSPI(value, i)
    i = i + 2
    message = sam.stop(sPI)
    if message = sPI does not exist then
        sLog.insertEntry(LogEntry(stopSA, sPI, fail, not found))
    end if
    if message = sPI not OPERATIONAL then
        sLog.insertEntry(LogEntry(stopSA, sPI, fail, not OPERATIONAL))
    end if
    if message = true then
        sLog.insertEntry(LogEntry(stopSA, sPI, success))
    end if
```

Algorithm 32 rekeySA(byte[] value)

```
i = 0
while i ¡ value.length do
    sPI = parseSPI(value, i)
    i = i + 2
    keyId = parsekeyId(value, i)
    i = i + 2
    message = sam.rekey(sPI, keyId)
    if message = sPI does not exist then
        sLog.insertEntry(LogEntry(rekeySA, sPI, keyId, fail, sPI not found))
    end if
    if message = sPI not UNKEYED then
        sLog.insertEntry(LogEntry(rekeySA, sPI, keyId, fail, sPI not UNKEYED))
    end if
    if message = keyID does not exist then
        sLog.insertEntry(LogEntry(rekeySA, sPI, keyId, fail, keyId not found))
    end if
    if message = keyId not ACTIVE then
        sLog.insertEntry(LogEntry(rekeySA, sPI, keyId, fail, keyId not ACTIVE))
    end if
    if message = sPI KEYED then
        sLog.insertEntry(LogEntry(rekeySA, sPI, keyId, success))
    end if
end while=0
```

Algorithm 33 expireSA(byte[] value)

```
i = 0
while i ¡ value.length do
    sPI = parseSPI(value, i)
    i = i + 2
    message = sam.expireSA(sPI)
    if message = sPI does not exist then
        sLog.insertEntry(LogEntry(expireSA, sPI, fail, not found))
    end if
    if message = sPI not KEYED then
        sLog.insertEntry(LogEntry(expireSA, sPI, fail, not KEYED))
    end if
    if message = sPI UNKEYED then
        sLog.insertEntry(LogEntry(expireSA, sPI, success))
    end if
end while
```

Algorithm 34 setARSN(byte[] value)

```
i = 0
while i ¡ value.length do
    sPI = parseSPI(value, i)
    i = i + 2
    aRC = parse(value, i)
    i = i + 2
    message = sam.setARSN(sPI, aRC)
    if message = sPI does not exist then
        sLog.insertEntry(LogEntry(setARSN, sPI, aRC, fail, sPI not found))
    end if
    if message = aRC invalid then
        sLog.insertEntry(LogEntry(setARSN, sPI, aRC, fail, aRC invalid))
    end if
    if message = sPI set ARSN to aRC then
        sLog.insertEntry(LogEntry(setARSN, sPI, aRC, success))
    end if
end while
```

Algorithm 35 setARSNWindow(byte[] value)

```
i = 0
while i ¡ value.length do
    sPI = parseSPI(value, i)
    i = i + 2
    aRCWindow = parse(value, i)
    i = i + 2
    message = sam.setARSNWindow(sPI, aRCWindow)
    if message = sPI does not exist then
        sLog.insertEntry(LogEntry(setARSNWindow, sPI, fail, sPI not found))
    end if
    if message = true then
        sLog.insertEntry(LogEntry(setARSNWindow, sPI, success))
    end if
end while
```

Algorithm 36 sAStatusRequest(byte[] value)

```
i = 0
status = ∅
while i ¡ value.length do
    sPI = parseSPI(value, i)
    i = i + 2
    message = sam.statusRequest(sPI)
    if message = sPI does not exist then
        sLog.insertEntry(LogEntry(sAStatusRequest, sPI, fail, not found))
        status.append(null)
    end if
    if message = curr, prev then
        sLog.insertEntry(LogEntry(sAStatusRequest, sPI, success))
        status.append(message)
    end if
end while
ReplyPDU answer(10011111)
answer.setLength(status.length)
answer.setValue(status)
return answer
```

Algorithm 37 readARSN(byte[] value)

```
i = 0
status = ∅
while i ¡ value.length do
    sPI = parseSPI(value, i)
    i = i + 2
    message = sam.readARSN(sPI)
    if message = sPI does not exist then
        sLog.insertEntry(LogEntry(readARSN, sPI, fail, not found))
        status.append(null)
    end if
    if message = aRC then
        sLog.insertEntry(LogEntry(readARSN, sPI, aRC, success))
        status.append(aRC)
    end if
end while
ReplyPDU answer(10010000)
answer.setLength(status.length)
answer.setValue(status)
return answer
```

Algorithm 38 ping()

```
ReplyPDU answer(10110001)
answer.setLength(0)
return answer
```

Algorithm 39 dumpLog()

```
ReplyPDU answer(10110011)
message = sLog.dumpLog()
answer.setLength(message.length)
answer.setValue(message)
return answer
```

Algorithm 40 eraseLog()

```
ReplyPDU answer(10110100)
message = sLog.eraseLog()
answer.setLength(message.length)
answer.setValue(message)
return answer
```