Hash Chain: a Scalable Content Provenance and Integrity Verifying Protocol for NDN

ETRI
박세형 (labry@etri.re.kr)
신용윤 (uni2u@etri.re.kr)

Oct 8, 2020

~~Problem Definition~~

- The modern **server** usually has two **CPU** sockets on its motherboard. These **CPUs** usually start at 4 **cores** and go, as of 2015 for Intel Xeon, up to 18 **cores** per **CPU**. While 4-socket and larger **servers** exist, they are less common today. A two-socket **server** with 36 **cores** is an insane overkill for most applications as of 2015. – from Quora

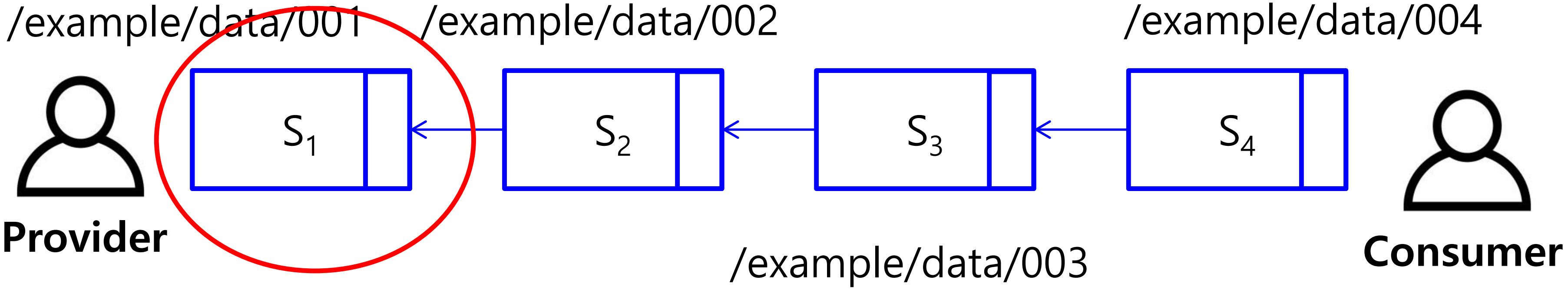
- Hash Chain can only utilize one core which is could lead to poor performance even compared to RSA-with-SHA256.

Hash Chain light-weight per-packet authentication

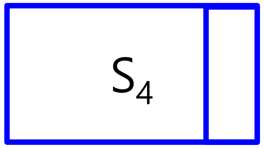
- Hash Chain (HC) has two mechanisms.
  - Batch Hash Chain: This is the default mode that guarantee the provenance upon receiving the packet. However, the provider needs to have the entire sequences before generating hash chain signatures.

  - Real-time Hash Chain: This is apt for real-time usage. However, this cannot guarantee the provenance until it receives the last packet. For example, traffic signal of vehicular networks, other real-time tactile IoT signals

Forward Chain Signature Generation Example



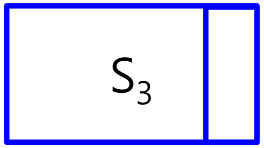
/example/data/004



$K_4 = H(name_4 \| data_4 \| 0x0000)$

$Null\_hash = H(0x00000000000000000000000)$

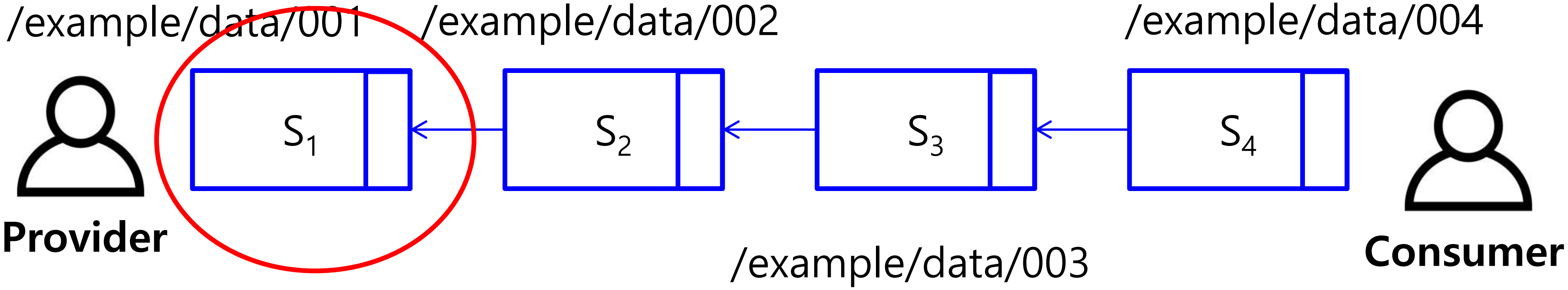
/example/data/003

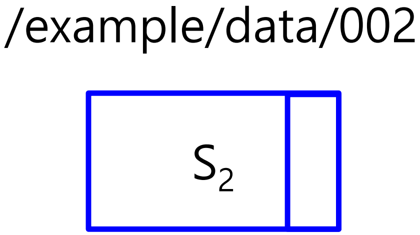


$K_3 = H(name_3 \| data_3 \| k_4)$

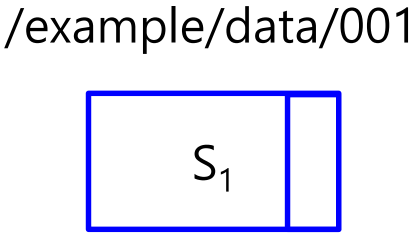
$k_4 = 0x66970e0d57360fdd4835c80$

Forward Chain Signature Generation Example





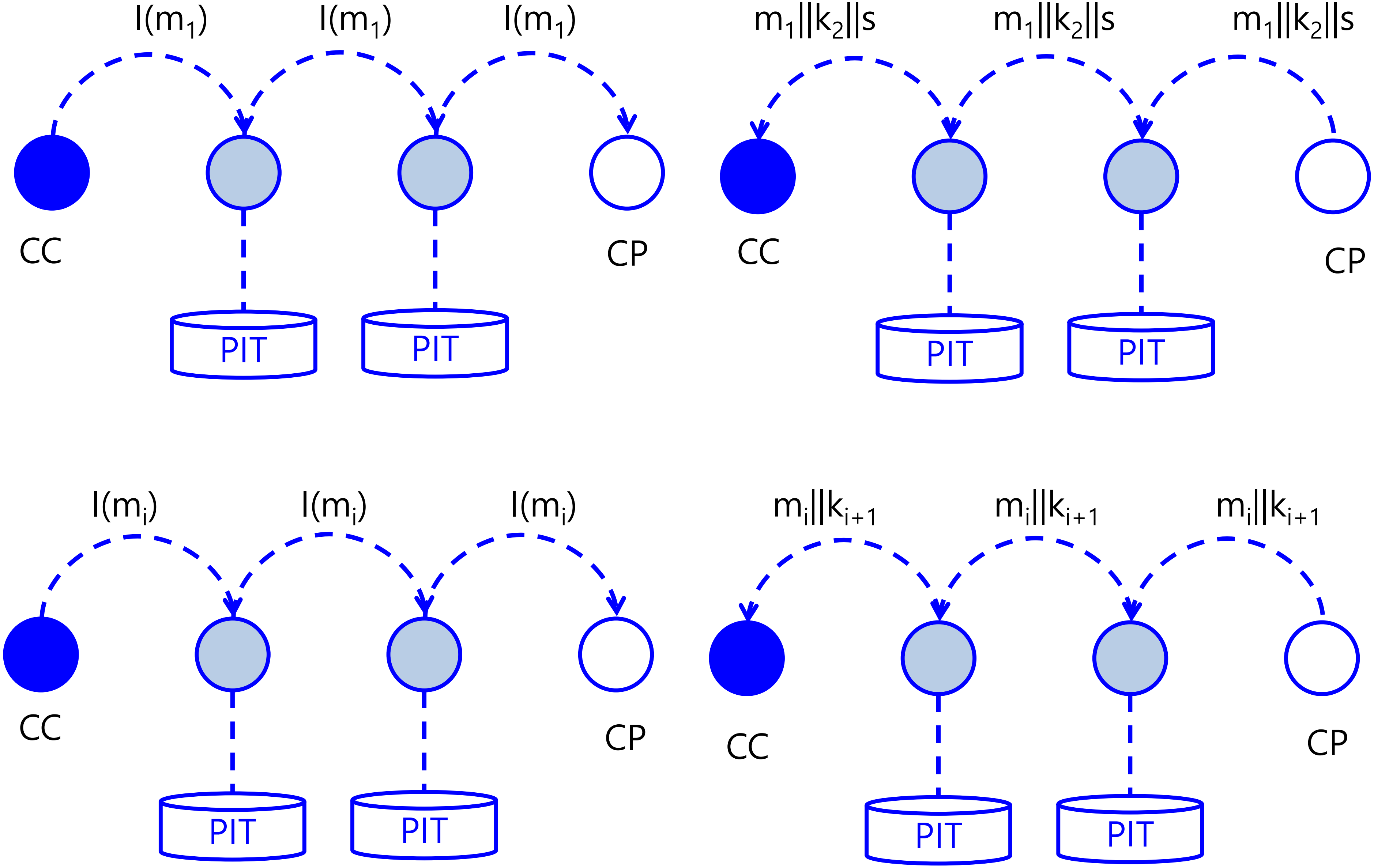
$$K_2 = H(name_2 || data_2 || k_3)$$



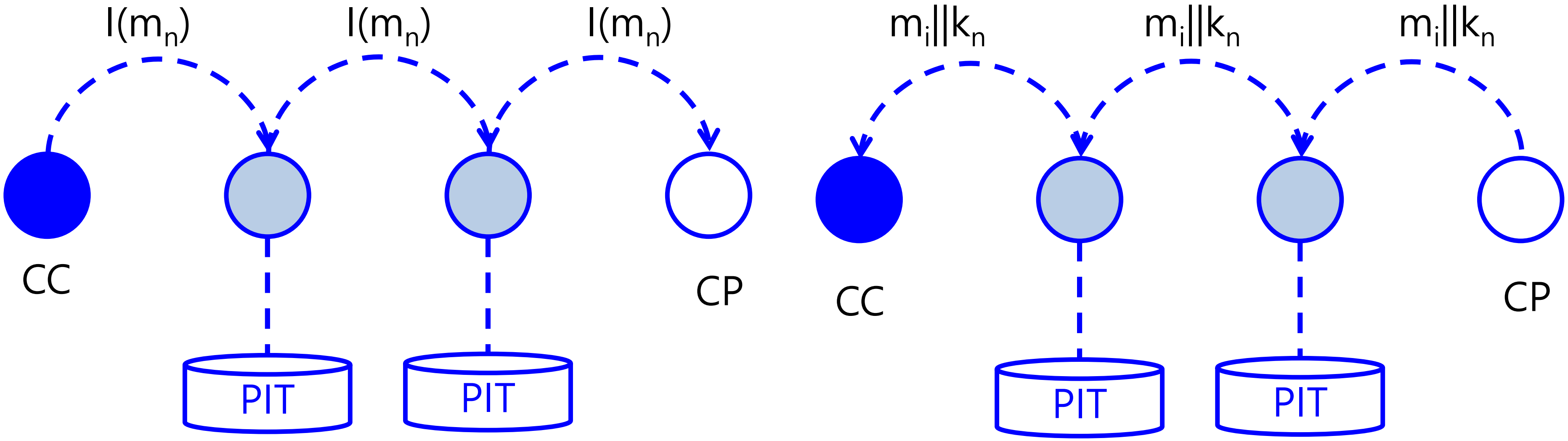
$$K_1 = H(name_1 || data_1 || k_2)$$

Forward Chain Signature Generation Example

- 0-step m_1 = $name_1$ + $data_1$
- (BC-1) $k_n = H(m_n \,||\, null\_hash)$
- (BC-2) $k_i = H(m_i \,||\, k_{i+1})$, all n > i > 1
- (BC-3) $k_1 = H(m_1 \,||\, k_2)$
- (BC-4) RSA_SIGN($PR_{CP,}$ K_1)
- (BC-5) S = E(PR_{CP} , $K_{1,}$ ~~K_n~~) This works without K_n

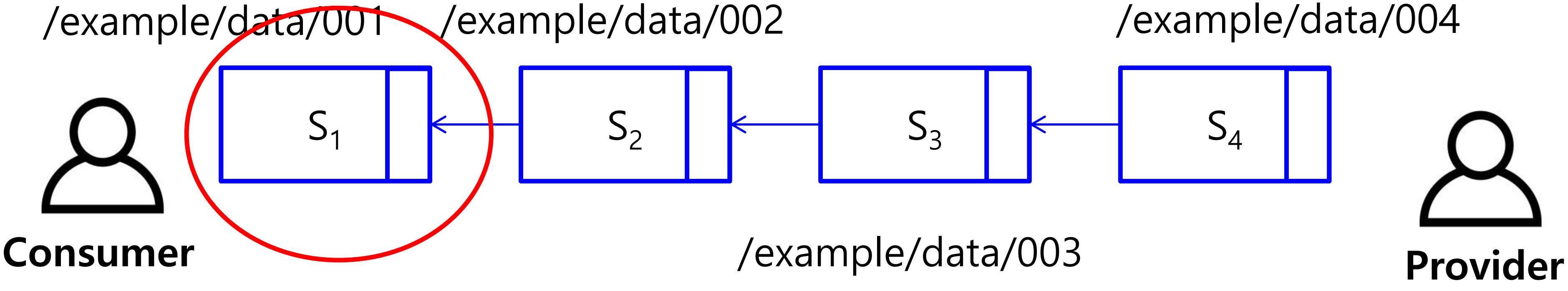


Propagation of Interest and Data packets for the Forward Hash Chain

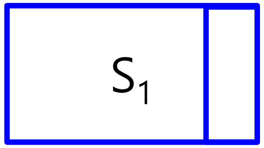


Propagation of Interest and Data packets for the Forward Hash Chain

Forward Chain Signature Verification Example



/example/data/001

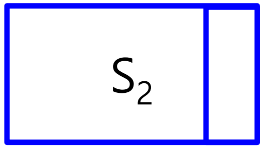


~~K_4,~~K_1 = Decrypt($\mathrm{PUB}_{CP,}$ S)

K_1 = H(name_1 || data_1 || k_2)

If K_1 == K_1' : pass

/example/data/002

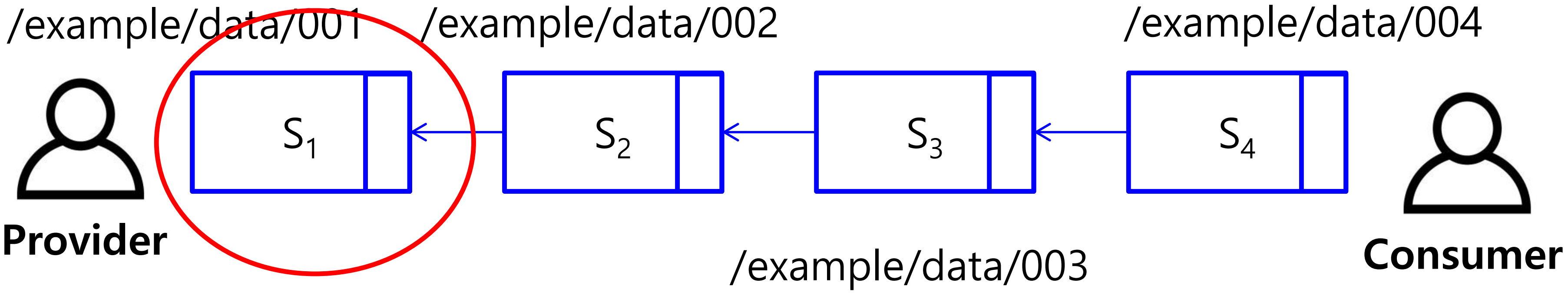


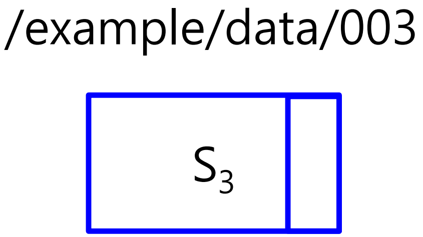
K_2 = H(name_2 || data_2 || k_3)

If K_2 == K_2': pass

※ K_1' prime means calculated

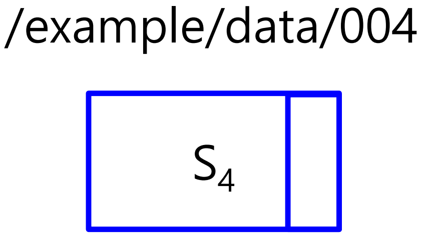
Forward Chain Signature Verification Example





K_3 = H($name_3$ || $data_3$ || k_4)

If K_3 == K_3' : pass



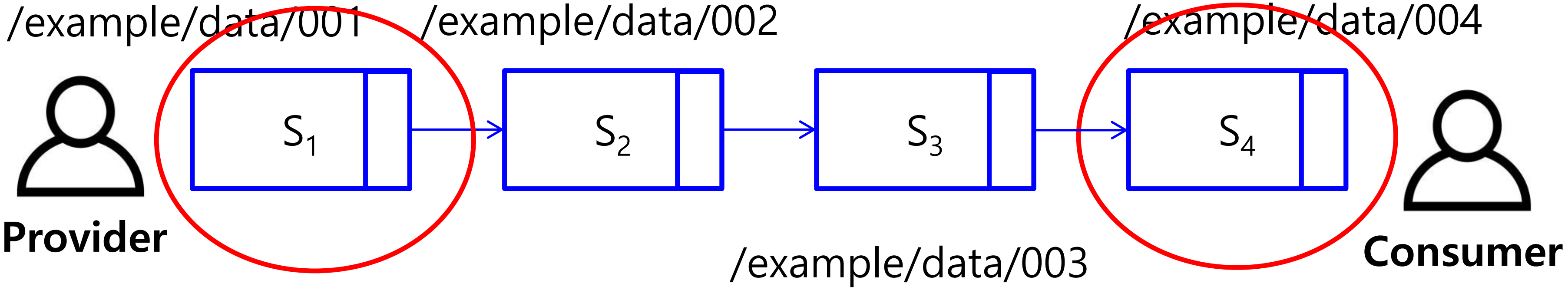
K_4 = H($name_4$ || $data_4$ || null_hash)

If K_4 == K_4': pass

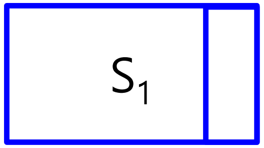
특허 내용

- Real-time Backward Hash Chain
- Backward HC is for real-time
- However, it is not enough.

Backward Chain Signature Generation Example



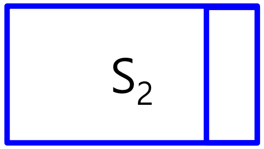
/example/data/001



$K_1 = H(name_1 \,||\, data_1 \,||\, 0x0000)$

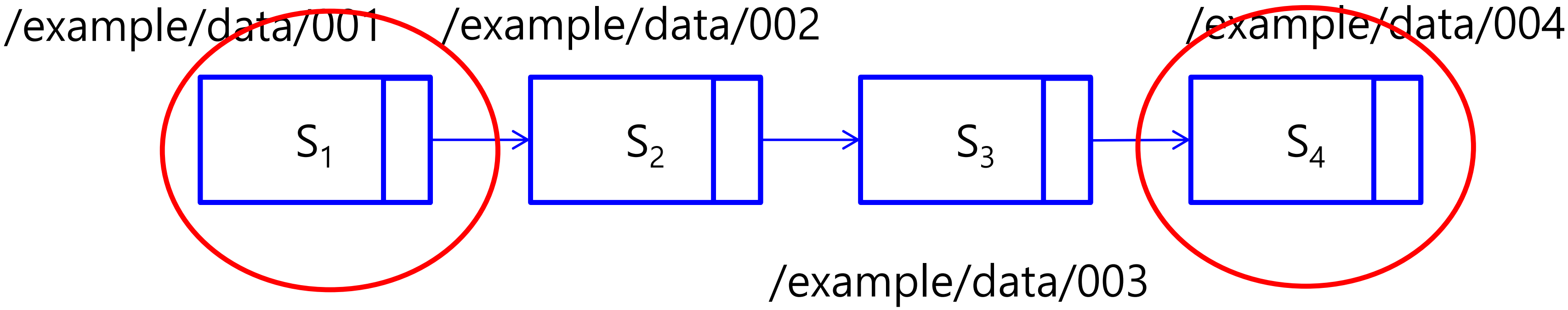
$Null\_hash = H(0x00000000000000000000000)$

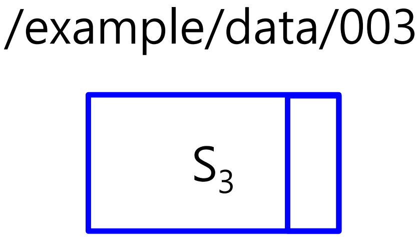
/example/data/002



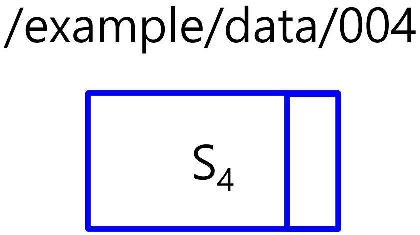
$K_2 = H(name_2 \,||\, data_2 \,||\, k_1)$

Backward Chain Signature Generation Example





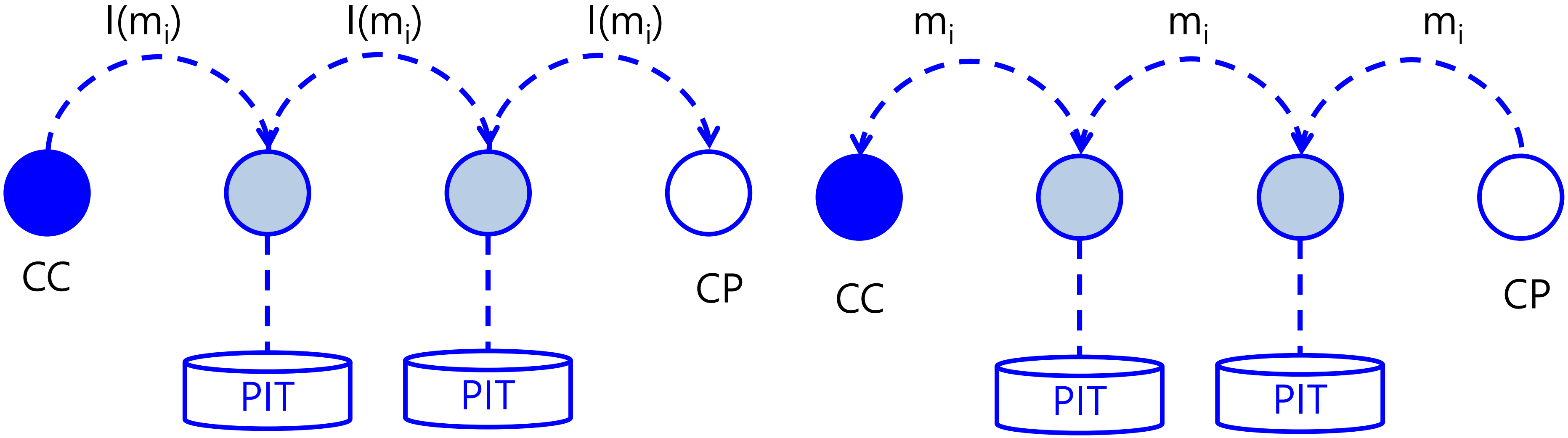
$K_3 = H(name_3 \| data_3 \| k_2)$



$K_4 = H(name_4 \| data_4 \| k_3)$

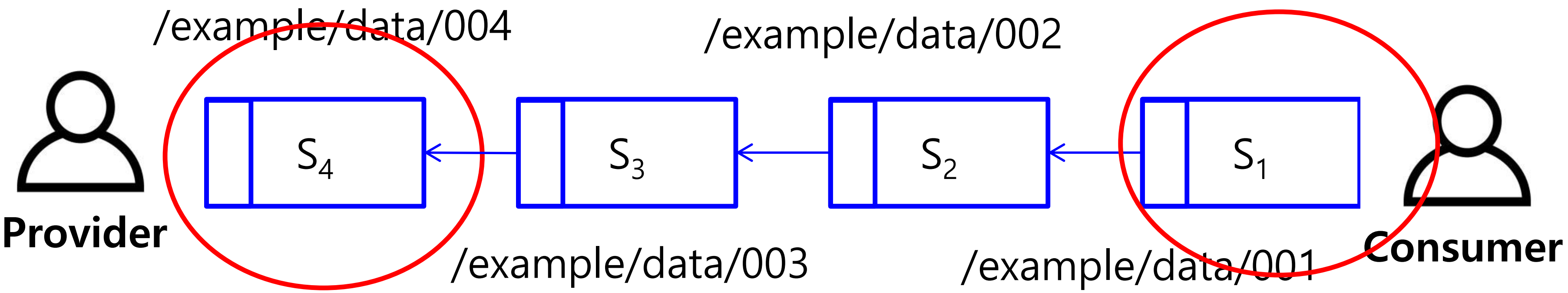
Real-time Backward Chain Signature Generation

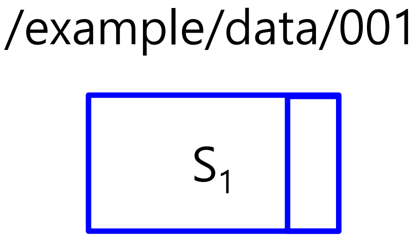
- 0-step m_1 = $name_1$ + $data_1$
- (BC-1) $k_1 = H(m_1 \| null\_hash)$
- (BC-2) RSA_SIGN($PR_{CP,}$ K_1)
- (BC-3) $k_i = H(m_i \| k_{i-1})$, all $n > i > 1$
- (BC-4) $k_n = H(m_n \| k_{n-1})$
- (BC-5) RSA_SIGN($PR_{CP,}$ K_n)



Propagation of Interest and Data packets for the Real-time Batch Hash Chain

Real-time Backward Chain Signature Verification Example

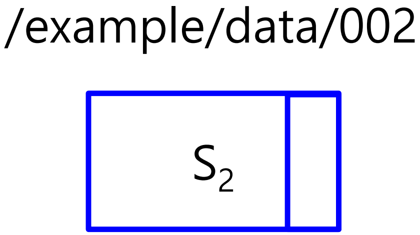




K_1 = Decrypt($PUB_{CP,}$ K_1)

K_1 = H($name_1$ || $data_1$ || 0x0000)

If K_1 == K_1′ : pass

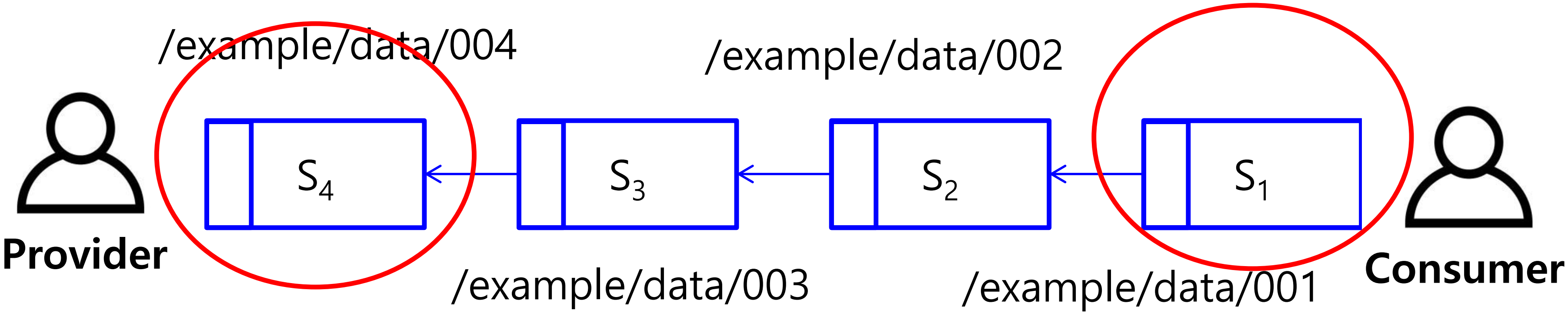


K_2 = H($name_2$ || $data_2$ || k_1)

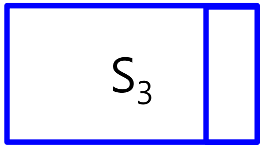
If K_2 == K_2′: delayed

※ K_1′ prime means calculated

Real-time Backward Chain Signature Verification Example



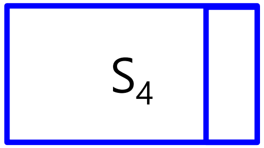
/example/data/003



$K_1 = H(name_3 || data_3 || k_2)$

If $K_3 == K_3'$: delayed

/example/data/004



$K_4 = Decrypt(PUB_{CP,} K_4)$

$K_4 = H(name_4 || data_4 || K_3)$

If $K_4 == K_4'$: pass

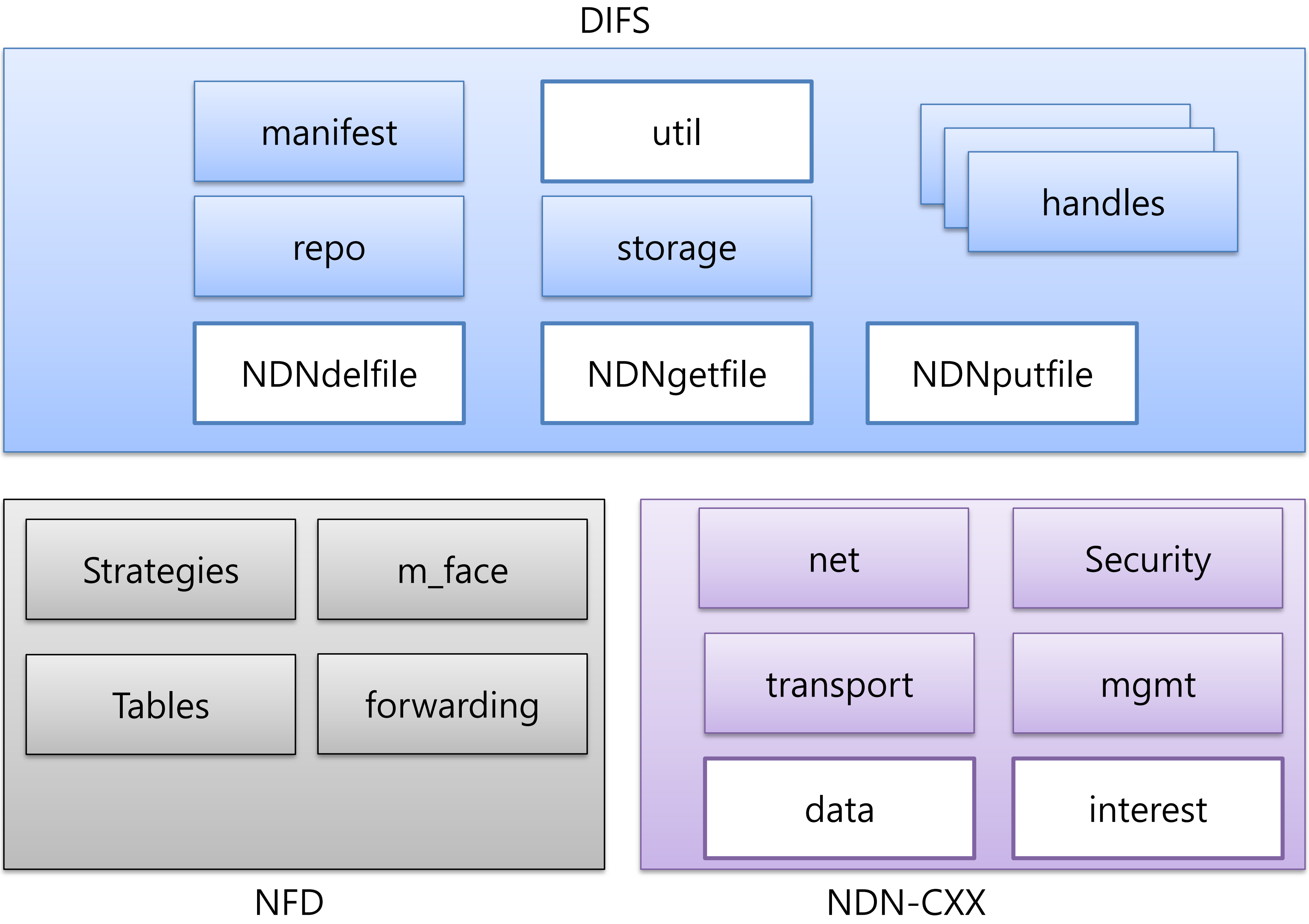
Value	Reference	Description
0	DigestSha256	Integrity protection using SHA-256 digest
1	SignatureSha256WithRsa	Integrity and provenance protection using RSA signature over a SHA-256 digest
3	SignatureSha256WithEcdsa	Integrity and provenance protection using an ECDSA signature over a SHA-256 digest
4	SignatureHmacWithSha256	Integrity and provenance protection using SHA256 hash-based message authentication codes
5	SignatureSha256WithHashChain	Integrity and provenance protection using HashChain signature over a Sha256 digest
6	DigestBlake3	Integrity protection using Blake-3 digest
7	SignatureBlake3WithHashChain	Integrity and provenance protection using HashChain signature over a BLAKE3 digest
2,5-200		reserved for future assignments
>200		unassigned

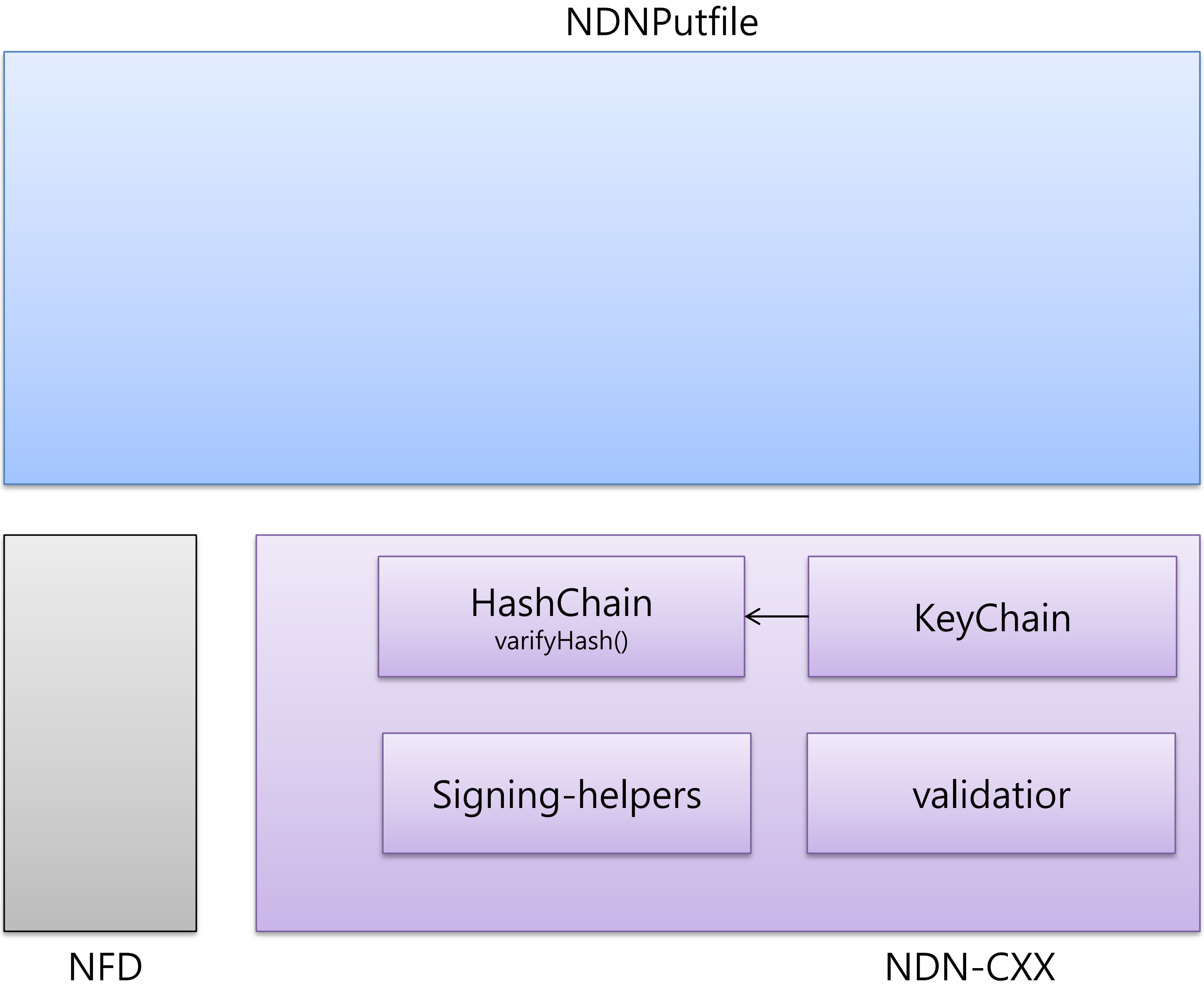
BLAKE-3

- BLAKE-3 is compatible with SHA-256
  - https://github.com/BLAKE3-team/BLAKE3
  - https://github.com/BLAKE3-team/BLAKE3/tree/master/c
  - BLAKE3 is based on an optimized instance of the established hash function BLAKE2 and on the original Bao tree mode. The specifications and design rationale are available in the BLAKE3 paper. The default output size is 256 bits. The current version of Bao implements verified streaming with BLAKE3.

Implementation

- https://named-data.net/doc/NDN-packet-spec/current/signature.html
- Ndn-cxx HashChain
- ndn-cxx/ndn-cxx/security/
  - Along with signature-sha256-with-ecds
  - Digest-sha256
  - Digest-blake3
  - Signature-hash-chain-with-blake3
- Validation-policy and key-chain:
  - Make changes to validate hash chain
  - Make changes to validate Blake3


DIFS
manifest
util
handles
repo
storage
NDNdelfile
NDNgetfile
NDNputfile
Strategies
m_face
Tables
forwarding
NFD
net
Security
transport
mgmt
data
interest
NDN-CXX



NDNPutfile

HashChain
varifyHash()

KeyChain

Signing-helpers

validatior

NFD

NDN-CXX

Implementation – A - Generation

- SignatureSha256WithHashChain:
  - Ndnputfile

```
Entire_segment = Segment file (entire_file)
Rev(entire_segment)
Prev_hash=null
For (segment: entire_segment) {
K = Keychain.sign(segment, prev_hash, ndn::signingWithHashChainSha256)
Prev_hash = k
List.add(k)
}
Data = PrepareSegmentToTransmit(segment,K)
Secret = encode(private_cp, k1)
For(data: entire_data) {
M_face(data)
}
```

  - NDN-CXX
  - https://github.com/uni2u/difs-cxx/blob/blake/ndn-cxx/security/key-chain.cpp

```
void
KeyChain::sign(Data& data, const prev_hash& hash, const SigningInfo& params)
{
  Name keyName;
  SignatureInfo sigInfo;
  std::tie(keyName, sigInfo) = prepareSignatureInfo(params);

  data.setSignatureInfo(sigInfo);

  EncodingBuffer encoder;
  data.wireEncode(encoder, true);

  Block sigValue(tlv::SignatureValue,
                 sign({{encoder.buf(), encoder.size()}}, keyName, prev_hash, params.getDigestAlgorithm()));

  data.wireEncode(encoder, sigValue);
}
```

Implementation – A - Verification

- SignatureSha256WithHashChain:
  - Ndngetfile

```
void
Consumer::onUnversionedData(const Interest& interest, const Data& data)
{
  data = fetchNextData();
  segmentNo = extactSegmentNo(data)
  if (!verifyData(data, segmentNo)) {
    BOOST_THROW_EXCEPTION(Error("Error verifying hash chain"));
  }
  readData(data);
}

bool
Consumer::verifyData(const Data& data, const uint segmentNo)
{
  bool ret;
  auto content = data.getContent();

  if(segmentNo == 0) {
              m_validator.validate(data,
              std::bind(&ManifestHandle::onDataValidated, this, interest, _1, processId),
              [](const Data& data, const ValidationError& error){NDN_LOG_ERROR("Error: " << error);});
              prevHash = exractKey(data);
  }

  ret = HashChain::verifyHash(content.value(), content.value_size(), prevHash);

  for (int i = 0; i < util::HASH_SIZE; i += 1) {
    prevHash[i] = content.value()[i];
  }

  return ret;
}
```

Implementation – B - Generation

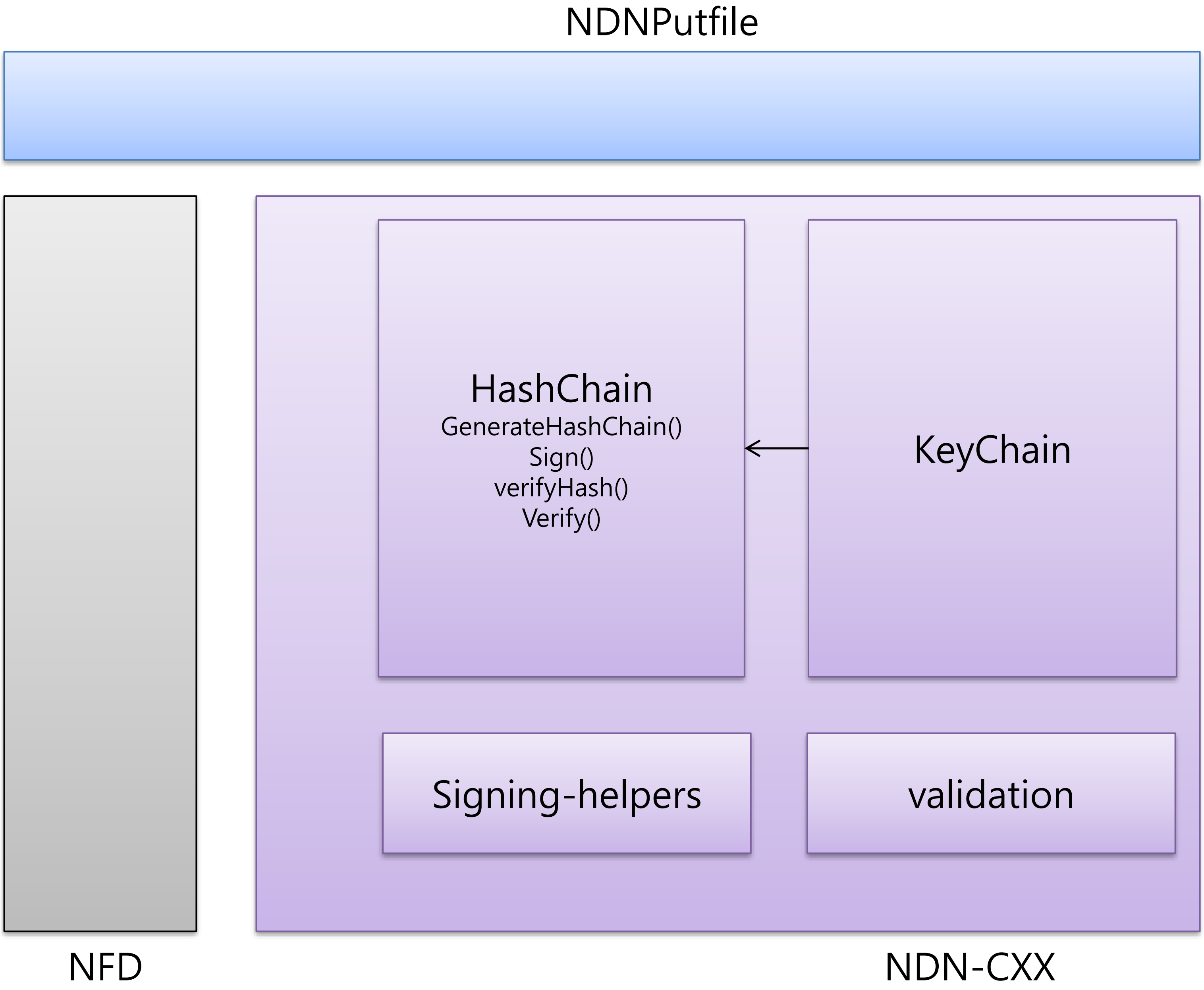
- SignatureSha256WithHashChain:
  - Ndnputfile

```
Entire_segment = Segment file (entire_file)
Rev(entire_segment)
HashChain hashChain(validator, Sha256); // hashChain(validator, Blake3)
//we hand_over validator to validate the first segment.

list_of_signatures = hashChain.generateHashChain(entire_segment);

Entire_data = PrepareSegmentToTransmit(list_of_signatures, entire_segment,)
Secret = encode(private_cp, list_of_signatures[0])
For(data: entire_data) {
M_face(data)
}
```

Implementation – B - Generation

- SignatureSha256WithHashChain:
  - NDN-CXX
  - https://github.com/uni2u/difs-cxx/blob/blake/ndn-cxx/security/hash-chain.cpp

```
List<signiture> HashChain::sign(DataSequence& data_entire) {
      List<signiture> list = List<signiture>();
      For (segment: entire_segment) {
      K = HashChain::sign(segment, prev_hash, ndn::signingWithHashChainSha256)
      Prev_hash = k
      List.add(k);
      }
    return list;
}

}
void
HashChain::sign(Data& data, const prev_hash& hash, const SigningInfo& params)
{
  Name keyName;
  SignatureInfo sigInfo;
  std::tie(keyName, sigInfo) = prepareSignatureInfo(params);

  data.setSignatureInfo(sigInfo);

  EncodingBuffer encoder;
  data.wireEncode(encoder, true);

  Block sigValue(tlv::SignatureValue,
                 sign({{encoder.buf(), encoder.size()}}, keyName, prev_hash, params.getDigestAlgorithm()));

  data.wireEncode(encoder, sigValue);
}
```



NDNPutfile

HashChain
GenerateHashChain()
Sign()
verifyHash()
Verify()

KeyChain

Signing-helpers

validation

NFD

NDN-CXX

Implementation – B - Verification

- SignatureHashChainWithSha256:
  - Ndngetfile

```
void
Consumer::onUnversionedData(const Interest& interest, const Data& data)
{
  data = fetchNextData();
  segmentNo = extactSegmentNo(data)
  if (!verifyData(data, segmentNo)) {
    BOOST_THROW_EXCEPTION(Error("Error verifying hash chain"));
  }
  readData(data);
}

bool Consumer::verifyData(const Data& data, const uint segmentNo)
{
  bool ret;
  auto content = data.getContent();

  if(segmentNo == 0) {
              m_validator.validate(data,
              std::bind(&ManifestHandle::onDataValidated, this, interest, _1, processId),
              [](const Data& data, const ValidationError& error){NDN_LOG_ERROR("Error: " << error);});

  }
  ret = HashChain::verifyHash(content.value(), content.value_size(), prevHash);

  for (int i = 0; i < util::HASH_SIZE; i += 1) {
    prevHash[i] = content.value()[i];
  }

  return ret;
}
```

Conclusion

- Let’s discuss!
- How to embed a Signature and keys
  - Signature in the data
  - (Data – size of signature)
  - Keys are placed as same as DigestSha256
  - https://named-data.net/doc/ndn-cxx/current/doxygen/d4/d08/sha256_8cpp_source.html