

CYBERNETICA  
Institute of Information Security

# Using Batch Hashing for Signing and Time-Stamping

Version 1.2  
Margus Freudenthal

T-4-20 / 2013

Copyright ©2013  
Margus Freudenthal.  
AS Cybernetica, Department of Information Security  
Systems

All rights reserved. The reproduction of all or part of  
this work is permitted for educational or research use  
on condition that this copyright notice is included in  
any copy.

Cybernetica research reports are available online at  
<http://research.cyber.ee/>

Mailing address:  
AS Cybernetica  
Mäealuse 2/1  
12618 Tallinn  
Estonia

# Using Batch Hashing for Signing and Time-Stamping

Margus Freudenthal

Version 1.2

## **Abstract**

This report describes algorithms and data formats for creating a single hash value for a collection of messages. In addition, we create for every message a proof that this message participated in calculation of the hash value. Batch hashes can be used in applications where an expensive cryptographic operation is used in a context with high performance requirements. In particular, this specification describes how batch hashes can be used to implement batch signing and batch time-stamping.

# Contents

<b>1</b>	<b>Introduction</b>	<b>5</b>
1.1	Batch Hashing . . . . .	5
1.2	Hash Lists . . . . .	5
1.3	Merkle Hash Trees . . . . .	6
<b>2</b>	<b>Data Formats</b>	<b>7</b>
2.1	General . . . . .	7
2.2	Hash Chains . . . . .	7
2.3	Hash Chain Results . . . . .	9
2.4	Serializing Hash Steps . . . . .	10
2.5	MIME Types . . . . .	11
<b>3</b>	<b>Processing Hash Chains</b>	<b>11</b>
3.1	Constructing Batch Hashes . . . . .	12
3.1.1	Additional Data Structures . . . . .	13
3.1.2	Main Function . . . . .	13
3.1.3	Building the Merkle Tree . . . . .	14
3.1.4	Building Hash Chains . . . . .	15
3.1.5	Helper Functions . . . . .	18
3.2	Verifying Hash Chains . . . . .	19
<b>4</b>	<b>Using Batch Hashes in ASiC Containers</b>	<b>21</b>
<b>A</b>	<b>XML Schema for Hash Chains</b>	<b>22</b>
<b>B</b>	<b>ASN.1 Module for <code>DigestList</code> Data Type</b>	<b>24</b>
<b>C</b>	<b>Algorithm for Creating Hash Chains</b>	<b>24</b>
<b>D</b>	<b>Algorithm for Verifying Hash Chains</b>	<b>28</b>
<b>E</b>	<b>Examples of the Data Structures</b>	<b>29</b>

# 1 Introduction

## 1.1 Batch Hashing

Cryptographic hashing is used to reduce the size of the messages so that they can be efficiently processed by resource-consuming operations such as signing or time-stamping. This document describes a mechanism for creating batch hash of a collection of messages so that all the messages can be signed or time-stamped in one operation. In particular, this mechanism is useful in the following cases:

- signing several messages using a slow signature creation device, such as smart card;
- signing a document and attachments using only one signature operation;
- time-stamping several data files with one request to reduce the load of the time-stamping service.

More formally, by a batch hash we mean an algorithm  $H$  that, having as input a list  $M_1, \dots, M_n$

of messages, creates a single hash value  $D$  and a set of proofs  $P_1, \dots, P_n$ , so that a there exists a verification algorithm  $V$ , so that  $V(D, P_i, M_i) = 1$  whenever  $D, (P_1, \dots, P_n) \leftarrow H(M_1, \dots, M_n)$ .

## 1.2 Hash Lists

A very simple method for implementing batch hashes is called *hash list*. In order to create a batch hash for messages  $M_1, \dots, M_n$ , the algorithm

1. Hashes all the messages:  $m_i = h(M_i)$  for all  $i = 1 \dots n$ .
2. Creates the *hash list*  $L = (m_1, m_2, \dots, m_n)$  and computes the hash value  $D = h(L)$  of the list.
3. The proof  $P_i$  for any  $M_i$  consists of the hash list  $L$ :  $P_i = L$ .

In order to verify  $D$  as the hash of  $M_i$  based on the proof  $P_i$ , the verifier

1. Computes  $m_i = h(M_i)$ ;
2. Checks if  $m_i \in L$ ;
3. Computes  $m = h(L)$ ; and
4. Verifies that  $m = D$ .

This scheme is very easy to implement and is feasible if the batch size is relatively small. The size of the proof is  $n \cdot |h|$ , where  $|h|$  is the number of output bits of  $h$ .

### 1.3 Merkle Hash Trees

Merkle hash tree [Mer80] is a tree in which every non-leaf node is labeled with the hash of the labels of its children nodes. Merkle trees can be used to implement batch hashes. In this scheme, the hash value  $D$  is the label of the root of the tree. For any message, the proof is made up of *batch residue* that is different for every message and contains hashes that are needed to calculate the tree root, starting from the given message.

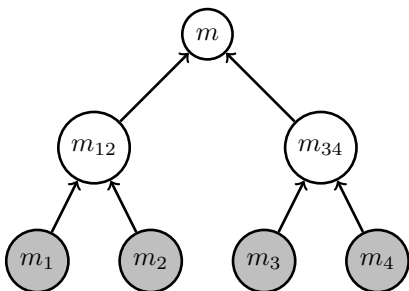


Figure 1: A Merkle hash tree for  $m_1, \dots, m_4$ .

For example, in case of  $n = 4$ , the batch hashes for a batch  $M_1, M_2, M_3, M_4$  is created (by using a hash function  $h$  via the following steps:

1. All the messages are hashed:  $m_i = h(M_i)$  for all  $i = 1 \dots 4$ .
2. The Merkle hash tree (see Figure 1) is computed:  $m_{12} = h(m_1, m_2)$ ,  $m_{34} = h(m_3, m_4)$ ,  $m = h(m_{12}, m_{34})$ .
3. The output of the hash value,  $D = m$ .
4. The proofs are composed as follows:  $P_1 = \{m_2, m_{34}\}$ ,  $P_2 = \{m_1, m_{34}\}$ ,  $P_3 = \{m_4, m_{12}\}$ ,  $P_4 = \{m_3, m_{12}\}$ .

In order to verify, whether a hash value  $D$  is the batch hash of  $M_3$  using proof  $P_3 = \{m_4, m_{12}\}$ , the verifier:

1. Computes  $m_3 = h(M_3)$ ;
2. Computes  $m = h(m_{12}, h(m_3, m_4))$ ; and
3. Compares the value  $m$  with given hash value  $D$  to see if they are the same.

Merkle hash trees are quite efficient, the length of the proof is  $|h| \cdot \log n$ , where  $|h|$  is the number of output bits of the hash function  $h$ .

## 2 Data Formats

### 2.1 General

This chapter describes XML-based data formats for expressing hash values and proofs that are created using Merkle hash trees. Because the data format is more general and allows expressing proofs for data structures that are not derived from strictly trees, the proofs are called *hash chains*.

This specification is based on data formats defined in the XML Signature standard [DSI08]. Additionally, this specification uses the reference processing model described in Section 4.3.3.2 of [DSI08].

The data structures and elements defined in this specification will be located under namespace `http://cyber.ee/hashchain`. The complete XML Schema is shown in Appendix A.

The following listing shows the header of the schema definition.

```
<schema xmlns="http://www.w3.org/2001/XMLSchema"
  targetNamespace="http://cyber.ee/hashchain"
  xmlns:tns="http://cyber.ee/hashchain"
  elementFormDefault="qualified"
  xmlns:ds="http://www.w3.org/2000/09/xmldsig#">
  <import
    schemaLocation="http://www.w3.org/TR/xmldsig-core/xmldsig-core-schema.xsd"
    namespace="http://www.w3.org/2000/09/xmldsig#" />
```

### 2.2 Hash Chains

The main data structure for representing hash computations is hash chain (element `HashChain` of type `HashChainType`), consisting of a series of hash steps (of type `HashStepType`). It is possible to state the default digest method (`DigestMethod`) in the hash chain level so that they do not have to be repeated for every step.

```
<element name="HashChain" type="tns:HashChainType"/>

<complexType name="HashChainType">
  <sequence>
    <element name="DefaultDigestMethod"
      type="ds:DigestMethodType" minOccurs="0"> </element>
    <element name="HashStep" type="tns:HashStepType"
      minOccurs="0" maxOccurs="unbounded"> </element>
  </sequence>
</complexType>
```

Hash step (of type `HashStepType`) represents hashing together a series of values. The values can be either concrete hash values (`HashValue`), references to hash values calculated from other hash steps (`StepRef`), or references to hash values

calculated from other data items (**DataRef**). All the values have the same base type, **AbstractValue** that defines the common elements. Hash steps can have **id** fields that can be used by other hash steps to refer to results of this hash step.

```

<complexType name="HashStepType">
  <sequence>
    <choice maxOccurs="unbounded" minOccurs="0">
      <element name="HashValue" type="tns:HashValueType"/>
      <element name="StepRef" type="tns:StepRefType"/>
      <element name="DataRef" type="tns:DataRefType"/>
    </choice>
  </sequence>
  <attribute name="id" type="ID"/>
</complexType>

```

The **AbstractValueType** is base type for results of hash calculations. All the different kinds of values can have a **DigestMethod** element that indicates the digest algorithm that was used to generate this value. For **StepRef** and **DataRef** elements the **DigestMethod** is used to hash the referenced data. For **HashValue** fields, the **DigestMethod** element is not directly used because the result of the digest operation is included in the element. However, the information about how the hash value was obtained, is used in the hash step calculation (see Section 2.4).

The **DigestMethod** element may be absent in the value element. In that case, the value of the **DefaultDigestMethod** element from the hash chain level is used. In order to keep data sizes small, it is recommended to use chain-level elements if possible and only use digest-level elements if the algorithms for this item differ from the defaults.

```

<complexType name="AbstractValueType">
  <sequence>
    <element minOccurs="0" ref="ds:DigestMethod"/>
  </sequence>
</complexType>

```

The **StepRef** element (of type **StepRefType**) represents a reference to result of another hash step calculation. The hash step is referred to by the attribute **URI**. When computing digest of a **StepRef** element, the input to the digest calculation is the result of processing the referred hash step with the method described in Section 2.4.

```

<complexType name="StepRefType">
  <complexContent>
    <extension base="tns:AbstractValueType">
      <attribute name="URI" type="anyURI" use="required"/>
    </extension>
  </complexContent>
</complexType>

```



The **DataRef** element (of type **DataRefType**) contains an URI that references a data object and a digest value of that data object. It is application-dependent whether the entity computing the value of the hash chain dereferences the URI in a **DataRef** element or simply uses the **DigestValue** field as the value of the reference. If the application decides to dereference the URI, it must check whether the digest calculated from the referenced data matches the digest in the **DigestValue** field.

When computing digest of the referenced data, the data object resulting from dereferencing the **URI** attribute is used as an input to transforms that are indicated by the **Transforms** element. The transformed data object is then digested using **DigestMethod**.

```

<complexType name="DataRefType">
  <complexContent>
    <extension base="tns:AbstractValueType">
      <sequence>
        <element minOccurs="0" ref="ds:Transforms"/>
        <element ref="ds:DigestValue"/>
      </sequence>
      <attribute name="URI" type="anyURI" use="required"/>
    </extension>
  </complexContent>
</complexType>

```

The **HashValue** element (of type **HashValueType**) is used to represent concrete hash values that are calculated from Merkle tree branches that are not included in the hash chain. The **Transforms** and **DigestMethod** elements are not directly used, but they are protected by including them in the serialized hash step.

```

<complexType name="HashValueType">
  <complexContent>
    <extension base="tns:AbstractValueType">
      <sequence>
        <element minOccurs="0" ref="ds:Transforms"/>
        <element ref="ds:DigestValue"/>
      </sequence>
    </extension>
  </complexContent>
</complexType>

```

## 2.3 Hash Chain Results

The **HashChainResult** element (of type **HashChainResultType**) represents the root of a Merkle tree. It is intended that this element is cryptographically protected either by digital signature or a time stamp. The hash chain result contains a digest value (element **DigestValue**) that contains the root hash value of the Merkle tree. Additionally, it contains a reference to a hash step

(attribute **URI**) and identifier of a digest method (element **DigestMethod**) that was used to hash the serialized version of the hash step (see Section 2.4) to construct the digest value in the hash chain result.

```
<element name="HashChainResult" type="tns:HashChainResultType"/>

<complexType name="HashChainResultType">
  <complexContent>
    <extension base="tns:StepRefType">
      <sequence>
        <element ref="ds:DigestValue"/>
      </sequence>
    </extension>
  </complexContent>
</complexType>
```

## 2.4 Serializing Hash Steps

This Section describes a data structure that is used for serializing result of a hash step computation. It uses Abstract Syntax Notation One (ASN.1 [ASN08]) and the Distinguished Encoding Rules (DER [DER08]) to achieve a unique binary representation of a data object.

When a hash chain result or a **StepRef** element references a hash step, then the result of this reference is calculated in the following manner.

1. All the references (**StepRef** and **DataRef** elements) objects are resolved and the resolved data items are digested. This may involve serializing of other hash steps.
2. A **DigestList** ASN.1 data structure is constructed based on all the values in the hash step (see below for details).
3. The **DigestList** data structure is serialized using DER.

The **DigestList** data structure consists of a sequence of **SingleDigest** objects. Each **SingleDigest** object corresponds to a single value in the hash step.

```
DigestList ::= SEQUENCE OF SingleDigest
```

The **SingleDigest** object contains the following fields.

- **digestValue** – for **DataValue** elements, this contains the contents of the **DigestValue** field. For **DataRef** and **StepRef** elements, this contains the digest of the referred data item.
- **digestMethodURI** – contains the value of the **Algorithm** attribute of the **DigestMethod** element in the original value. If the value does not contain the **DigestMethod** element, then the value of the **DefaultDigestMethod** element from the containing hash chain is used instead.

- `transformsURI` – for `DataValue` and `DataRef` nodes, contains the values of the `Algorithm` attributes of the `Transforms` element.

```
SingleDigest ::= SEQUENCE {
  digestValue OCTET STRING,
  digestMethodURI UTF8String,
  transformsURI SEQUENCE OF UTF8String
}
```

The full ASN.1 module describing the `DigestList` data structure is presented in Appendix B.

## 2.5 MIME Types

This section describes MIME types that are used when hash chains and hash chain results are transported over a MIME-based protocol or stored in containers, such as ASiC (see Section 4).

MIME information for hash chains (element `HashChain`).

- File name extension is `.xml`.
- Content type is `application/hash-chain`.

MIME information for hash chain results (element `HashChainResult`).

- File name extension is `.xml`.
- Content type is `application/hash-chain-result`.

## 3 Processing Hash Chains

This chapter describes two algorithms for processing hash chains. First, Section 3.1 presents an algorithm for constructing hash chains for a group of data items. Second, section 3.2 describes an algorithm for verifying a hash chain.

The algorithms are expressed in a pseudocode that uses Ruby syntax (see <http://www.ruby-lang.org/> for more information about the language). In order to avoid dealing with various technical details, we make the following assumptions.

- XML content is expressed as objects. The class names match the type names in the XML schema. Therefore, a hash chain is represented by object of class `HashChainType`.
- XML elements are accessible as properties. In order to conform to Ruby naming convention, first letter of the element name is converted to lower-case. Thus, element `DigestMethod` becomes property `digestMethod`.

- Like XML, the ASN.1 data structures are expressed as objects with fields represented by properties.
- If an element is missing, then the corresponding property is assumed to be `nil`.
- There exists a library providing some helper functions. In particular, the algorithms depend on the following helper functions.
  - `fetch_data(uri)` – takes as input an URI, dereferences it, and returns the referenced data object. The data object can be either an octet stream or an XML node.
  - `perform_transforms(data, transforms)` – takes as input data object and list of transforms (represented by `TransformsType`) and returns the transformed data object. The transforms are processed according to the XML Signature specification [DSI08]).
  - `calculate_digest(data, digest_method)` – takes as input a data object and a digest method identifier (URI) and returns the digested data.
  - `der_encode(data)` – takes as input a `DigestList` object and returns DER encoding of the object.
  - `get_hash_chain(value)` – takes as input a hash step value (of type `HashValueType`, `DataRefType` or `StepRefType`) and returns the `HashChain` element containing this hash step value.

In order keep the control flow of the algorithms more simple, the code does not use object-oriented approach and instead consists of functions that take objects as arguments.

### 3.1 Constructing Batch Hashes

This section describes the algorithm for creating a batch hash for a collection of data items. This is accomplished by constructing a Merkle hash tree and then creating a hash chain for each of the individual data items.

The algorithm takes as input three parameters:

- array of data items that need to be hashed (can be either octet streams or XML nodes);
- list of transforms (of type `TransformsType`) that are applied to the data items before hashing; and
- digest method used.

The algorithm outputs hash chain result (the **HashChainResult** element) and an array of hash chains (**HashChain** element), one for each input data item. Each hash chain proves that the corresponding data item participated in computing the hash chain result.

The algorithm described here is just one possible implementation of batch hashing. For example, the shape of the hash tree can vary between different implementations. The hash steps can reference many input data items instead of one. Additionally, different transforms and digest methods can be used for different data items.

### 3.1.1 Additional Data Structures

The algorithm makes use of several intermediate data structures. First, the **TreeNode** class is used to store the Merkle tree nodes when the tree is built. It has three attributes: **digest** that contains the hash of the child nodes, **children** that stores the array of child nodes, and **transforms** that contains list of transforms that were used to produce the digest in this node (**transforms** is only used for nodes that reference input data items). For leaf nodes that represent digests of the input data items, the field **children** is set to **nil**.

```
class TreeNode
  attr_accessor :digest
  attr_accessor :children
  attr_accessor :transforms
end
```

The **BatchHash** class is used to represent the output of the algorithm. It contains two attributes. Attribute **hash\_chain\_result** contains the result of the hash chain calculation (element **HashChainResult** of type **HashChainResultType**). Attribute **hash\_chains** is an array that contains one hash chain (of type **HashChainType**) for every input data item, in the same order as in the input array.

```
class BatchHash
  attr_accessor :hash_chain_result
  attr_accessor :hash_chains
end
```

### 3.1.2 Main Function

The entry point to the construction algorithm is the **batch\_hash** function. The the function takes the three parameters previously described in this section and returns object of type **BatchHash** that contains the results of the batch hashing. First, the function builds the hash tree for the inputs (line 2). Next, it constructs the **HashChainResult** element based on root of the tree (lines 6-11). The example code assumes that the hash chains will be located in the same XML document as the hash chain result and therefore uses relative URIs (e.g., **#step\_0** to refer to the first step of the hash chain). If this is not the case, the

URI must be changed. Finally, the call to `build_hash_chain` function builds the hash chains for the input data items (line 14).

```
def batch_hash(data_items, item_transforms, digest_method) 1
  tree = build_tree(data_items, item_transforms, digest_method) 2
  3
  result = BatchHash.new 4
  5
  result.hash_chain_result = HashChainResultType.new 6
  # Hash chain result refers to first step of the hash chain. 7
  # The step can also be outside the current document. 8
  result.hash_chain_result.uri = "#step_0" 9
  result.hash_chain_result.digestMethod = digest_method 10
  result.hash_chain_result.digestValue = tree.digest 11
  12
  # Build the hash chains 13
  result.hash_chains = build_hash_chains(tree, digest_method) 14
  15
  return result 16
end 17
```

### 3.1.3 Building the Merkle Tree

The `build_tree` function takes the same arguments as the `batch_hash` function and returns a Merkle hash tree (of type `TreeNode`) built from the input data items. The function handles two main cases.

- If the `data_items` array contains only single item, then a leaf node is constructed, containing digest of the given data item (lines 4-10).
- If the `data_items` array contains multiple items, it is split into two parts and a subtree is built for each part (lines 14-21). Next, a tree node is built based on digests of the subtrees (lines 24-25).

```
def build_tree(data_items, item_transforms, digest_method) 1
  if data_items.length == 1 then 2
    # We have only one item, build a leaf node 3
    result = TreeNode.new 4
    result.digest = transform_and_digest(data_items[0], 5
      item_transforms, digest_method) 6
    result.transforms = item_transforms 7
    # Mark it as leaf node 8
    result.children = nil 9
    return result 10
  else 11
    # Build a non-leaf node 12
    # Decide where to split the input array 13
    split_index = (data_items.length / 2.0).ceil 14
    15
    # Build the left and right child nodes 16
    left_node = build_tree(data_items[0, split_index], 17
      item_transforms, digest_method) 18
    right_node = build_tree( 19
```

```

        data_items[split_index, data_items.length],
        item_transforms, digest_method)
    20
    21
    22
    # Create the corresponding TreeNode object
    23
    return build_tree_node([left_node, right_node],
    24
        digest_method)
    25
    end
    26
end
    27

```

The `build_tree_node` function takes as input array of subtrees and digest method. It returns a tree node that has the input nodes as children.

```

def build_tree_node(children, digest_method)
    1
    result = TreeNode.new
    2
    result.digest = digest_hash_step(children, digest_method)
    3
    result.transforms = nil
    4
    result.children = children
    5
    return result
    6
end
    7

```

The `digest_hash_step` function takes as input an array of tree nodes and a digest method identifier. It constructs a `DigestList` ASN.1 data structure that contains a `SingleDigest` object for every tree node in the input (lines 3-17). The `DigestList` data structure is described in Section 2.4 of this specification. Finally, the data structure is DER-encoded and digested (lines 20-21).

```

def digest_hash_step(nodes, digest_method)
    1
    # Create the data structure that will be digested
    2
    digest_list = DigestList.new
    3
    4
    for node in nodes
    5
        digest_item = SingleDigest.new
    6
        # Fill in the transforms field
    7
        if node.transforms != nil then
    8
            for transform in node.transforms
    9
                digest_item.transformsURI << transform.algorithm
    10
            end
    11
        end
    12
        digest_item.digestMethodURI = digest_method
    13
        digest_item.digestValue = node.digest
    14
    15
        digest_list << digest_item
    16
    end
    17
    18
    # Transform the data structure and calculate digest
    19
    return calculate_digest(der_encode(digest_list),
    20
        digest_method)
    21
end
    22

```

### 3.1.4 Building Hash Chains

The `build_hash_chains` function takes as input a hash tree and a digest method identifier. It returns an array of hash chains where each hash chain corresponds to a leaf node in the input tree.

When constructing hash chains, the algorithm uses partial hash chains that are referred to as *templates*. When walking the tree, each step away from the root adds a new hash step to the template. When the tree walk reaches a leaf node, the template becomes fully constructed hash chain that is added to the result.

The `build_hash_chains` function starts by creating an empty hash chain template (lines 4-6) and an empty result array (line 9). Next, it calls a recursive function `build_for_node` (line 12) that will walk the Merkle tree, build hash chains, and add them to the result array.

```

def build_hash_chains(tree, digest_method) 1
  # Template will contain the global fields of hash chain 2
  # and the previous hash steps. 3
  template = HashChainType.new 4
  # Set the default value so that hash steps will be smaller 5
  template.defaultDigestMethod = digest_method 6
  7
  # This will receive the hash chains for all the data items 8
  result = [] 9
  10
  # Walk the tree and construct the result 11
  build_for_node(tree, template, result) 12
  13
  return result 14
end 15

```

The `build_for_node` function takes as input a tree node, a hash chain template, and a result array. It iterates over the children of the input node. For every child node, the function constructs a new hash step that contains a `StepRef` or a `DataRef` node for the current child and `HashValue` nodes for all its siblings (line 8). In Merkle tree terms, this replaces the corresponding tree branches with concrete hash values. The new step is added to the template (line 10). If the current node is a leaf node, the hash chain is complete and is added to the results (line 14), otherwise the function is recursively called for the current child node (line 17).

```

def build_for_node(node, template, result) 1
  for i in 0..(node.children.length - 1) 2
    current_child = node.children[i] 3
    4
    # Build a new template containing this hash step 5
    new_template = copy_template(template) 6
    # Create hash step for this node 7
    new_step = build_new_step(node.children, i, template) 8
    # Add new step to the end. 9
    new_template << new_step 10
    11
    if is_leaf?(current_child) then 12
      # We have a full hash chain in our hands 13
      result << new_template 14
    else 15
      # Build hash steps for children 16
      build_for_node(current_child, new_template, result) 17
    end 18
  end

```



```
end
end
```

19  
20

The `build_new_step` function takes as input an array of tree nodes, an index and a hash chain template. It returns a hash step (of type `HashStepType`) containing `HashValue` elements for every node in the input array, except the one indicated by `ref_node_idx` parameter.

First, the function constructs a new hash step (lines 2-3). It then iterates over the nodes in the input array. For the node pointed by `ref_node_idx`, the function adds either a `DataRef` or a `StepRef` element to the hash step, depending on whether the node is a leaf node or not. For leaf nodes, the `DataRef` will contain reference to external data, the transforms for processing the referenced data, and the digest of the transformed data (lines 15-19). For non-leaf nodes, the `StepRef` will contain URI of the next hash step (lines 23-25).

If the child node is not the node pointed by `ref_node_idx`, then the function constructs a `HashValue` that contains hash of the tree node (30-34).

```
def build_new_step(nodes, ref_node_idx, template) 1
  new_step = HashStepType.new 2
  new_step.id = step_id(template.length) 3
  4
  # Go over all the nodes 5
  for i in 0..(nodes.length - 1) 6
    if i == ref_node_idx then 7
      # This is the value that should reference the 8
      # next hash step 9
      10
      if is_leaf?(nodes[i]) then 11
        # This was a leaf node. This means that the 12
        # reference is not to another hash step but 13
        # to data item instead. 14
        data_ref = DataRefType.new 15
        data_ref.uri = "/data" 16
        data_ref.transforms = nodes[i].transforms 17
        data_ref.digestValue = nodes[i].digest 18
        new_step << data_ref 19
      else 20
        # We refer to hash step generated for next 21
        # level of the tree 22
        step_ref = StepRefType.new 23
        step_ref.uri = "#" + step_id(template.length + 1) 24
        new_step << step_ref 25
      end 26
    else 27
      # This is hash value corresponding to the tree nodes 28
      # not present in the hash chain 29
      digest_value = HashValueType.new 30
      digest_value.digestValue = nodes[i].digest 31
      digest_value.transforms = nodes[i].transforms 32
      33
      new_step << digest_value 34
    end 35
  end 36
end 37
```

<code>return new_step</code>	38
<code>end</code>	39

### 3.1.5 Helper Functions

The `transform_and_digest` helper function takes as input a data item, list of transformations, and a digest method identifier. It applies the transforms on the input (lines 2-6) and digests the result (line 9).

<code>def transform_and_digest(data, transforms, digest_method)</code>	1
<code>if transforms != nil then</code>	2
<code>before_digest = perform_transforms(data, transforms)</code>	3
<code>else</code>	4
<code>before_digest = data</code>	5
<code>end</code>	6
 <code># Calculate digest</code>	7
<code>return calculate_digest(before_digest, digest_method)</code>	8
<code>end</code>	9
	10

The `copy_template` helper function takes as input a hash chain template (of type `HashChainType`) and returns copy of it.

<code>def copy_template(template)</code>	1
<code>result = HashChainType.new</code>	2
	3
 <code># Copy the default value</code>	4
<code>result.defaultDigestMethod = template.defaultDigestMethod</code>	5
	6
 <code># Copy the hash steps created so far</code>	7
<code>for hash_step in template</code>	8
<code>result &lt;&lt; hash_step</code>	9
<code>end</code>	10
	11
<code>return result</code>	12
<code>end</code>	13

The `step_id` helper function takes as input a hash step's sequence number and returns a string that can be used as the value of the `id` field in the `HashStep` element.

<code>def step_id(id)</code>	1
<code>return "step_#{id}"</code>	2
<code>end</code>	3

The `is_leaf?` helper function takes as input a Merkle tree node and returns `true`, if this node is a leaf node.

<code># Returns true, if the node is a leaf node</code>	1
<code>def is_leaf?(node)</code>	2
<code>return node.children == nil</code>	3
<code>end</code>	4

## 3.2 Verifying Hash Chains

This section describes the algorithm for verifying a hash chain with respect to a `HashChainResult` element that is signed, time-stamped or protected by some other means. The verification algorithm takes as input one parameter: `HashChainResult` element containing a digest (root of the hash tree) that is protected by other means and is the start of the verification process. During verification, the algorithm resolves all the references encountered in the hash chain and may also reference external resources.

The entry point to the verification algorithm is the `verify_hash_chain` function. It takes as a parameter the hash chain result and verifies it. If the verification fails, the function raises an error, otherwise it simply returns to caller. The verification takes place in three steps.

1. The hash step, referenced by the URI in the hash chain, is fetched and serialized according to algorithm described in Section 2.4 (line 3).
2. The serialized hash step is used as an input to digest calculation (lines 6-8).
3. The computed digest is compared with the digest in the hash chain result (lines 12-14). If they match, the verification succeeds, otherwise the verification fails.

```
def verify_hash_chain(hash_chain_result) 1
  # Dereference the URI 2
  hash_step_data = resolve_hash_step(hash_chain_result.uri) 3
  4
  # Calculate digest 5
  digested_data = calculate_digest( 6
    hash_step_data, 7
    hash_chain_result.digestMethod) 8
  9
  # Verify that the calculated digest matches 10
  # the digest in hash chain result 11
  if digested_data != hash_chain_result.digestValue then 12
    raise "Digests_do_not_match" 13
  end 14
  15
  # Otherwise, everything is fine. 16
end 17
```

The function `resolve_hash_step` takes as a parameter an URI pointing to a hash step and returns the serialized form of this hash step. First, it uses the `fetch_data` library function to dereference the URI (line 2). Next, it calls the helper function `serialize_hash_step` to compute the binary representation of the hash step (line 3).

```
def resolve_hash_step(uri) 1
  hash_step = fetch_data(uri) 2
  return serialize_hash_step(hash_step) 3
end 4
```

The `serialize_hash_step` function encapsulates the serialization algorithm described in Section 2.4. It takes as a parameter a `HashStep` element and returns serialized version of it. The function creates an empty `DigestList` object (line 2) and populates it with one `SingleDigest` object for every value in the hash step. The construction of the `SingleDigest` object varies according to type of the value.

- For `HashValue` elements, the `SingleDigest` object is populated with properties of the hash step value (lines 16-19).
- For `DataRef` elements, the URI is dereferenced using the `fetch_data` library function (line 21). If applicable, the indicated transforms are applied to the fetched data objects (lines 22-26). The data is digested and copied to the `SingleDigest` object (lines 27-28). Finally, the digest is compared to the `DigestValue` field of the `DataRef` element (lines 29-31). If the referenced data is not available, the application can forgo dereferencing the URI and use value of the `DigestValue` field as the digest of the referenced data.
- For `StepRef` elements, the hash step is resolved using the `resolve_hash_step` helper function and digested using the digest method indicated either in the `StepRef` element or in the hash chain (lines 35-37).

```

def serialize_hash_step(hash_step) 1
  result = DigestList.new          2
  # Iterate over all the values in the hash step. 3
  # Since there is no intermediate element, the hash step 4
  # is treated as iterable. 5
  for step_value in hash_step      6
    # Prepare the normalized value 7
    digest = SingleDigest.new      8
    9
    digest.digestMethodURI = get_digest_method(step_value) 10
    11
    case step_value                12
    when HashValueType             13
      # Pass digest as is          14
      digest.digestValue = digest.digestValue 15
      for transforms in step_value.transforms 16
        digest.transforms << transforms.algorithm 17
      end 18
    when DataRefType               19
      data = fetch_data(step_value.uri) 20
      if step_value.transforms != null then 21
        to_digest = perform_transforms(data, transforms) 22
      else 23
        to_digest = data 24
      end 25
      digest.digestValue = calculate_digest( 26
        to_digest, get_digest_method(step_value)) 27
      if digest.digestValue != step_value.digestValue then 28
        raise "Digests_do_not_match_in_DataRef" 29
      end 30
    end
  end
end

```

```

        end
        digest.transforms = step_value.transforms
    when StepRefType
        # Derereference the URI
        step_data = resolve_hash_step(step_value.uri)
        digest.digestValue = calculate_digest(
            step_data, get_digest_method(step_value))
    end

    # Add the normalized value to the output
    result << new_value
end

return result
end

```

The `get_digest_method` helper function takes as a parameter a value from a hash step and returns digest method applicable for this value. If the value contains digest method, then this is returned. Otherwise, the `DefaultDigestMethod` element from the containing hash chain is returned instead.

```

def get_digest_method(value)
    if value.digestMethod != nil then
        return value.digestMethod
    else
        return get_hash_chain(old_value).digestMethod
    end
end

```

## 4 Using Batch Hashes in ASiC Containers

This section describes the use of batch hashes for implementing batch signatures and batch time-stamps. Both of these mechanisms are implemented in terms of Associated Signature Containers (ASiC) syntax [ASI13]. In general, this means that the ASiC container contains the following data files:

- a `HashChainResult` element that represents the top of the Merkle tree calculation;
- a `HashChain` element containing a hash chain calculation from the data files to the hash chain result;
- one or more data files that are referenced by the hash chain.

From these data files, the file containing hash chain result must be protected by a cryptographic mechanism, such as a signature or a time-stamp. Thus, the verification process will consist of two separate steps.

1. The verifier checks that the signature or the time stamp is correct and that it references the file containing the hash chain result. This verification is done using the mechanisms described in the ASiC specification.

2. The verifier verifies the signed or time-stamped hash chain using the algorithm described in Section 3.2. This verifies the connection between the protected hash chain result and the input data files.

Typically, the all the data files reside in the same container but this is not mandatory. It is also possible to use e.g., HTTP URIs to reference data that resides outside the ASiC container. It is application-dependent how the references to resources outside the ASiC container are processed. The application can choose either to fetch and verify the resources, to leave the external resources unverified (possibly saving the digest values so that they can be verified later), or failing the verification.

## References

- [ASi13] Electronic Signatures and Infrastructures (ESI); Associated Signature Containers (ASiC), Version 1.3.1. Technical Specification TS 102 918, ETSI ESI, June 2013.
- [ASN08] Information technology – Abstract Syntax Notation One (ASN.1): Specification of basic notation. Standard X.680, ITU Telecommunication Standardization Sector, November 2008.
- [DER08] Information technology – ASN.1 encoding rules: Specification of Basic Encoding Rules (BER), Canonical Encoding Rules (CER) and Distinguished Encoding Rules (DER). Standard X.690, ITU Telecommunication Standardization Sector, November 2008.
- [DSi08] XML Signature Syntax and Processing (Second Edition). W3c recommendation, 10 June 2008.
- [Mer80] Ralph C. Merkle. Protocols for public key cryptosystems. In *Proc. of the 1980 IEEE Symposium on Security and Privacy*, pages 122–134, 1980.

## A XML Schema for Hash Chains

```
<?xml version="1.0" encoding="UTF-8"?>
<schema xmlns="http://www.w3.org/2001/XMLSchema"
  targetNamespace="http://cyber.ee/hashchain"
  xmlns:tns="http://cyber.ee/hashchain"
  elementFormDefault="qualified"
  xmlns:ds="http://www.w3.org/2000/09/xmldsig#">
  <import
    schemaLocation="http://www.w3.org/TR/xmldsig-core/xmldsig-core-schema.xsd"
    namespace="http://www.w3.org/2000/09/xmldsig#" />

  <complexType name="HashChainType">
```

```

    <sequence>
      <element name="DefaultDigestMethod"
        type="ds:DigestMethodType" minOccurs="0"> </element>
      <element name="HashStep" type="tns:HashStepType"
        minOccurs="0" maxOccurs="unbounded"> </element>
    </sequence>
  </complexType>

  <complexType name="HashStepType">
    <sequence>
      <choice maxOccurs="unbounded" minOccurs="0">
        <element name="HashValue" type="tns:HashValueType"/>
        <element name="StepRef" type="tns:StepRefType"/>
        <element name="DataRef" type="tns:DataRefType"/>
      </choice>
    </sequence>
    <attribute name="id" type="ID"/>
  </complexType>
  <complexType name="AbstractValueType">
    <sequence>
      <element minOccurs="0" ref="ds:DigestMethod"/>
    </sequence>
  </complexType>
  <complexType name="StepRefType">
    <complexContent>
      <extension base="tns:AbstractValueType">
        <attribute name="URI" type="anyURI" use="required"/>
      </extension>
    </complexContent>
  </complexType>
  <complexType name="DataRefType">
    <complexContent>
      <extension base="tns:AbstractValueType">
        <sequence>
          <element minOccurs="0" ref="ds:Transforms"/>
        </sequence>
        <attribute name="URI" type="anyURI" use="required"/>
      </extension>
    </complexContent>
  </complexType>
  <complexType name="HashValueType">
    <complexContent>
      <extension base="tns:AbstractValueType">
        <sequence>
          <element minOccurs="0" ref="ds:Transforms"/>
          <element ref="ds:DigestValue"/>
        </sequence>
      </extension>
    </complexContent>
  </complexType>
  <complexType name="HashChainResultType">
    <complexContent>
      <extension base="tns:StepRefType">
        <sequence>
          <element ref="ds:DigestValue"/>
        </sequence>
      </extension>
    </complexContent>
  </complexType>

```

```

        </extension>
      </complexContent>
    </complexType>

    <element name="HashChain" type="tns:HashChainType"/>
    <element name="HashChainResult" type="tns:HashChainResultType"/>
  </schema>

```

## B ASN.1 Module for **DigestList** Data Type

```

BatchHashDigestList
{ iso(1) identified-organization(3) dod(6) internet(1) private(4)
  enterprise(1) cybernetica(3516) id-mod(4) id-mod-batchhashes(8) }

```

```

DEFINITIONS EXPLICIT TAGS ::=
BEGIN

```

```

DigestList ::= SEQUENCE OF SingleDigest

```

```

SingleDigest ::= SEQUENCE {
  digestValue OCTET STRING,
  digestMethodURI UTF8String,
  transformsURI SEQUENCE OF UTF8String
}

```

```

END

```

## C Algorithm for Creating Hash Chains

```

# Represents node in a Merkle hash tree
class TreeNode
  # Digest value of this node
  attr_accessor :digest
  # Array of children (if non-leaf)
  attr_accessor :children
  # Transforms used to build this node
  attr_accessor :transforms
end

# Represents result of a batch hashing operation
class BatchHash
  # The result of the hash chain calculation
  # that is signed or time-stamped (ReferenceType)
  attr_accessor :hash_chain_result
  # Array of hash chains, one chain for every input data item
  attr_accessor :hash_chains
end

# Calculates batch hash of given data items.
# Returns object of type BatchHash.
def batch_hash(data_items, item_transforms, digest_method)
  tree = build_tree(data_items, item_transforms, digest_method)

  result = BatchHash.new

```



```

    result.hash_chain_result = HashChainResultType.new
    # Hash chain result refers to first step of the hash chain.
    # The step can also be outside the current document.
    result.hash_chain_result.uri = "#step_0"
    result.hash_chain_result.digestMethod = digest_method
    result.hash_chain_result.digestValue = tree.digest

    # Build the hash chains
    result.hash_chains = build_hash_chains(tree, digest_method)

  end

  return result
end

# Builds the merkle hash tree of given data items.
# Returns the tree.
def build_tree(data_items, item_transforms, digest_method)
  if data_items.length == 1 then
    # We have only one item, build a leaf node
    result = TreeNode.new
    result.digest = transform_and_digest(data_items[0],
                                         item_transforms, digest_method)
    result.transforms = item_transforms
    # Mark it as leaf node
    result.children = nil
    return result
  else
    # Build a non-leaf node
    # Decide where to split the input array
    split_index = (data_items.length / 2.0).ceil

    # Build the left and right child nodes
    left_node = build_tree(data_items[0, split_index],
                           item_transforms, digest_method)
    right_node = build_tree(
      data_items[split_index, data_items.length],
      item_transforms, digest_method)

    # Create the corresponding TreeNode object
    return build_tree_node([left_node, right_node],
                           digest_method)
  end
end

# Create TreeNode object from the input data
def build_tree_node(children, digest_method)
  result = TreeNode.new
  result.digest = digest_hash_step(children, digest_method)
  result.transforms = nil
  result.children = children
  return result
end

# Serializes a hash step composing of given nodes.
def digest_hash_step(nodes, digest_method)
  # Create the data structure that will be digested
  digest_list = DigestList.new

```

```

    for node in nodes
      digest_item = SingleDigest.new
      # Fill in the transforms field
      if node.transforms != nil then
        for transform in node.transforms
          digest_item.transformsURI << transform.algorithm
        end
      end
      digest_item.digestMethodURI = digest_method
      digest_item.digestValue = node.digest

      digest_list << digest_item
    end

    # Transform the data structure and calculate digest
    return calculate_digest(der_encode(digest_list),
      digest_method)
end

# Helper function that combines two related tasks
def transform_and_digest(data, transforms, digest_method)
  if transforms != nil then
    before_digest = perform_transforms(data, transforms)
  else
    before_digest = data
  end

  # Calculate digest
  return calculate_digest(before_digest, digest_method)
end

# Build hash chains from a given tree.
# Returns array of hash chains.
def build_hash_chains(tree, digest_method)
  # Template will contain the global fields of hash chain
  # and the previous hash steps.
  template = HashChainType.new
  # Set the default value so that hash steps will be smaller
  template.defaultDigestMethod = digest_method

  # This will receive the hash chains for all the data items
  result = []

  # Walk the tree and construct the result
  build_for_node(tree, template, result)

  return result
end

# Builds hash steps for a given tree node and adds them to
# the result.
# The parameter template contains hash steps corresponding
# to higher levels of the tree.
def build_for_node(node, template, result)
  for i in 0..(node.children.length - 1)
    current_child = node.children[i]

```

```

    # Build a new template containing this hash step
    new_template = copy_template(template)
    # Create hash step for this node
    new_step = build_new_step(node.children, i, template)
    # Add new step to the end.
    new_template << new_step

    if is_leaf?(current_child) then
      # We have a full hash chain in our hands
      result << new_template
    else
      # Build hash steps for children
      build_for_node(current_child, new_template, result)
    end
  end
end

# Builds a new hash step and returns it.
# Parameter ref_node_idx is index of the value that
# should be RefValue instead of HashValue.
def build_new_step(nodes, ref_node_idx, template)
  new_step = HashStepType.new
  new_step.id = step_id(template.length)

  # Go over all the nodes
  for i in 0..(nodes.length - 1)
    if i == ref_node_idx then
      # This is the value that should reference the
      # next hash step

      if is_leaf?(nodes[i]) then
        # This was a leaf node. This means that the
        # reference is not to another hash step but
        # to data item instead.
        data_ref = DataRefType.new
        data_ref.uri = "/data"
        data_ref.transforms = nodes[i].transforms
        data_ref.digestValue = nodes[i].digest
        new_step << data_ref
      else
        # We refer to hash step generated for next
        # level of the tree
        step_ref = StepRefType.new
        step_ref.uri = "#" + step_id(template.length + 1)
        new_step << step_ref
      end
    else
      # This is hash value corresponding to the tree nodes
      # not present in the hash chain
      digest_value = HashValueType.new
      digest_value.digestValue = nodes[i].digest
      digest_value.transforms = nodes[i].transforms

      new_step << digest_value
    end
  end
end

```

```

    return new_step
end

# Makes copy of a hash chain template
def copy_template(template)
  result = HashChainType.new

  # Copy the default value
  result.defaultDigestMethod = template.defaultDigestMethod

  # Copy the hash steps created so far
  for hash_step in template
    result << hash_step
  end

  return result
end

# Create unique identifier for a given step
def step_id(id)
  return "step_#{id}"
end

# Returns true, if the node is a leaf node
def is_leaf?(node)
  return node.children == nil
end

```

## D Algorithm for Verifying Hash Chains

```

def verify_hash_chain(hash_chain_result)
  # Dereference the URI
  hash_step_data = resolve_hash_step(hash_chain_result.uri)

  # Calculate digest
  digested_data = calculate_digest(
    hash_step_data,
    hash_chain_result.digestMethod)

  # Verify that the calculated digest matches
  # the digest in hash chain result
  if digested_data != hash_chain_result.digestValue then
    raise "Digests_do_not_match"
  end

  # Otherwise, everything is fine.
end

def resolve_hash_step(uri)
  hash_step = fetch_data(uri)
  return serialize_hash_step(hash_step)
end

def serialize_hash_step(hash_step)
  result = DigestList.new

```

```

# Iterate over all the values in the hash step.
# Since there is no intermediate element, the hash step
# is treated as iterable.
for step_value in hash_step
  # Prepare the normalized value
  digest = SingleDigest.new

  digest.digestMethodURI = get_digest_method(step_value)

  case step_value
  when HashValueType
    # Pass digest as is
    digest.digestValue = digest.digestValue
    for transforms in step_value.transforms
      digest.transforms << transforms.algorithm
    end
  when DataRefType
    data = fetch_data(step_value.uri)
    if step_value.transforms != null then
      to_digest = perform_transforms(data, transforms)
    else
      to_digest = data
    end
    digest.digestValue = calculate_digest(
      to_digest, get_digest_method(step_value))
    if digest.digestValue != step_value.digestValue then
      raise "Digests_do_not_match_in_DataRef"
    end
    digest.transforms = step_value.transforms
  when StepRefType
    # Dererence the URI
    step_data = resolve_hash_step(step_value.uri)
    digest.digestValue = calculate_digest(
      step_data, get_digest_method(step_value))
  end

  # Add the normalized value to the output
  result << new_value
end

return result
end

def get_digest_method(value)
  if value.digestMethod != nil then
    return value.digestMethod
  else
    return get_hash_chain(old_value).digestMethod
  end
end
end

```

## E Examples of the Data Structures

This appendix contains examples of the data structures. The examples are cryptographically correct and can be used to test implementations for compliance with this specification.

We construct the example tree from Section 1.3 and the corresponding hash chains for authenticating each message.

First, the messages  $M_i$  are the strings “one”, “two”, “three”, “four”. Their corresponding hashes  $m_i$  have the following values. Note: from here on, binary data is presented in Base64 encoded form.

- $m_1 = \text{"dpLDrTVAu4A8Ags67mbNiIcSMjTqDG5xQ8Ct1z/0Me0="}$
- $m_2 = \text{"P8TM/nRYcOLA2Z9x8w/wZWyN7dQcwfT03aw2+aF4vM="}$
- $m_3 = \text{"i1udsME9skJWyCmqNkqpDG0uuJGLkjKkq5MTuVTTVV8="}$
- $m_4 = \text{"B0+vCA9aPnThwp0cpqSFaTgsu80yTo1Z0rg+8hwDnwA="}$

Next, we present serialized versions of the hash steps  $M_{12}$ ,  $M_{34}$ ,  $M$  and their corresponding hashes  $m_{12} = h(M_{12})$ ,  $m_{34} = h(M_{34})$ , and  $m = h(M)$ . We use SHA-256 digest function (with identifier <http://www.w3.org/2001/04/xmlenc#sha256>) throughout this example.

- $M_{12} =$

```

MIGeME0EIHASw601QLuAPAIL0u5mzYiHEjI06gxucUPArdc/9DHtDCdodHRw0i8v
d3d3LnczLm9yZy8yMDAxLzA0L3htbGVuYyNzaGEyNTYwADBNCBA/xMz+dFhw4sDZ
n3HzD/B1bI3t1BzB19PTdrDb5oXi8wwnaHR0cDovL3d3dy53My5vcmcvMjAwMS8w
NC94bWxlbmMjc2hhMjU2MAA=

```

- $m_{12} = \text{"MARSPzN9ZeFA+a9/yBxPkyIv/z36AP8GqQzEbFW/l84="}$

- $M_{34} =$

```

MIGeME0EIItnbnDBPbJCVsgpqjZKqQxtLroxI5IypKuTE7LU01VfDCdodHRw0i8v
d3d3LnczLm9yZy8yMDAxLzA0L3htbGVuYyNzaGEyNTYwADBNCBAE768ID1o+d0HC
nRympIVp0Cy7zTJ0jVnSuD7yHA0fAAwnaHR0cDovL3d3dy53My5vcmcvMjAwMS8w
NC94bWxlbmMjc2hhMjU2MAA=

```

- $m_{34} = \text{"CJdduPWzdgD0PdwsNkHdAnS6wQgw20kr5p++u4Wt+kk="}$

- $M =$

```

MIGeME0EIDAUEj8zfwXhQPmVf8gcT5MiL/89+gD/BqkMxGxVv5f0DCdodHRw0i8v
d3d3LnczLm9yZy8yMDAxLzA0L3htbGVuYyNzaGEyNTYwADBNCBAI11249bN2APQ9
3Cw2Qd0CdLrBCDDY6Svmn767ha36SQwnaHR0cDovL3d3dy53My5vcmcvMjAwMS8w
NC94bWxlbmMjc2hhMjU2MAA=

```

- $m = \text{"D7oIIhfhp4ToT729xyx991PvstI5XvpW+d7oeWvXw8E="}$

The previously described Merkle tree is represented by the following hash chain result (assuming that the hash chain is saved to file `/hashchain.xml`).

```

<HashChainResult xmlns:ds="http://www.w3.org/2000/09/xmldsig#"
  xmlns="http://cyber.ee/hashchain"
  URI="/hashchain.xml#step_0">
  <ds:DigestMethod
    Algorithm="http://www.w3.org/2001/04/xmenc#sha256"/>
  <ds:DigestValue>D7oIIfhfp4ToT729xyx991PvstI5XvpW+d7oeWvXw8E=</ds:DigestValue>
</HashChainResult>

```

Next, we construct a hash chain for each of the input message. The data files are assumed to reside in external files `/data1` to `/data4`.

The following XML element contains the hash chain for message  $M_1$ .

```

<HashChain xmlns:ds="http://www.w3.org/2000/09/xmldsig#"
  xmlns="http://cyber.ee/hashchain">
  <DefaultDigestMethod
    Algorithm="http://www.w3.org/2001/04/xmenc#sha256"/>
  <HashStep id="step_0">
    <StepRef URI="#step_1"/> <!-- M12 -->
    <HashValue> <!-- m34 -->
      <ds:DigestValue>CJdduPWzdgD0PdwsNkHdAnS6wQgw20k
        r5p++u4Wt+kk=</ds:DigestValue>
    </HashValue>
  </HashStep>
  <HashStep id="step_1">
    <DataRef URI="/data1"> <!-- M1 -->
      <ds:DigestValue>dpLDrTVAu4A8Ags67mbNiIcSMjTqDG5
        xQ8Ct1z/0Me0=</ds:DigestValue>
    </DataRef>
    <HashValue> <!-- m2 -->
      <ds:DigestValue>P8TM/nRYc0LA2Z9x8w/wZWyN7dQcwf
        T03aw2+aF4vM=</ds:DigestValue>
    </HashValue>
  </HashStep>
</HashChain>

```

The following XML element contains the hash chain for message  $M_2$ .

```

<HashChain xmlns:ds="http://www.w3.org/2000/09/xmldsig#"
  xmlns="http://cyber.ee/hashchain">
  <DefaultDigestMethod
    Algorithm="http://www.w3.org/2001/04/xmenc#sha256"/>
  <HashStep id="step_0">
    <StepRef URI="#step_1"/> <!-- M12 -->
    <HashValue> <!-- m34 -->
      <ds:DigestValue>CJdduPWzdgD0PdwsNkHdAnS6wQgw20k
        r5p++u4Wt+kk=</ds:DigestValue>
    </HashValue>
  </HashStep>
  <HashStep id="step_1">
    <HashValue> <!-- m1 -->
      <ds:DigestValue>dpLDrTVAu4A8Ags67mbNiIcSMjTqDG5
        xQ8Ct1z/0Me0=</ds:DigestValue>
    </HashValue>
    <DataRef URI="/data2"> <!-- M2 -->
      <ds:DigestValue>P8TM/nRYc0LA2Z9x8w/wZWyN7dQcwf
        T03aw2+aF4vM=</ds:DigestValue>
    </DataRef>
  </HashStep>
</HashChain>

```

```
</DataRef>
</HashStep>
</HashChain>
```

The following XML element contains the hash chain for message  $M_3$ .

```
<HashChain xmlns:ds="http://www.w3.org/2000/09/xmldsig#"
  xmlns="http://cyber.ee/hashchain">
  <DefaultDigestMethod
    Algorithm="http://www.w3.org/2001/04/xmlenc#sha256"/>
  <HashStep id="step_0">
    <HashValue <!-- m12 -->
      <ds:DigestValue>MARSPzN9ZeFA+a9/yBxPkyIv/z36AP8
        GqQzEbfW/l84=</ds:DigestValue>
    </HashValue>
    <StepRef URI="#step_1"/> <!-- M34 -->
  </HashStep>
  <HashStep id="step_1">
    <DataRef URI="/data3"> <!-- M3 -->
      <ds:DigestValue>i1udsME9skJWycmqNkqpDG0uujGLkjK
        kq5MTuVTTVV8=</ds:DigestValue>
    </DataRef>
    <HashValue <!-- m4 -->
      <ds:DigestValue>B0+vCA9aPnThwp0cpqSFaTgsu80yTo1
        Z0rg+8hwDnwA=</ds:DigestValue>
    </HashValue>
  </HashStep>
</HashChain>
```

The following XML element contains the hash chain for message  $M_4$ .

```
<HashChain xmlns:ds="http://www.w3.org/2000/09/xmldsig#"
  xmlns="http://cyber.ee/hashchain">
  <DefaultDigestMethod
    Algorithm="http://www.w3.org/2001/04/xmlenc#sha256"/>
  <HashStep id="step_0">
    <HashValue <!-- m12 -->
      <ds:DigestValue>MARSPzN9ZeFA+a9/yBxPkyIv/z36AP8
        GqQzEbfW/l84=</ds:DigestValue>
    </HashValue>
    <StepRef URI="#step_1"/> <!-- M34 -->
  </HashStep>
  <HashStep id="step_1">
    <HashValue <!-- m3 -->
      <ds:DigestValue>i1udsME9skJWycmqNkqpDG0uujGLkjK
        kq5MTuVTTVV8=</ds:DigestValue>
    </HashValue>
    <DataRef URI="/data4"> <!-- M4 -->
      <ds:DigestValue>B0+vCA9aPnThwp0cpqSFaTgsu80yTo1
        Z0rg+8hwDnwA=</ds:DigestValue>
    </DataRef>
  </HashStep>
</HashChain>
```