Open KSI Format

Ahto Buldas, Andres Kroonmaa, Allan Park

San Francisco - Tallinn
February 2012

Contents

1 Introduction

Keyless signature is an alternative solution to traditional PKI signatures. The word *keyless* does not mean that no cryptographic keys were used during the signature creation. Keys are still necessary for authentication. This means that *signatures can be reliably verified without assuming secrecy of cryptographic keys*.

Kelyess signatures are not vulnerable to key compromise and hence, they provide a solution to the long-term validity of digital signatures. Also the traditional PKI signatures might be protected by timestamps, but as far as the timestamping technology is also PKI-based, the problem of key compromise is still there.

Keyless signatures are created by using *cryptographic hash functions*, which are one-way and collision-resistant, but do not use any keys.

1.1 Signature Creation Process

Keyless signatures are implemented in practice as *multi-signatures*, i.e. many documents are signed at a time. The signing process (Fig. 1) involves the steps of:

- *Hashing*: Creating hash values of the documents;
- *Aggregation*: Creating a per-second global hash tree;
- *Publication*: Creating a hash-tree from the per-second hash values and publishing the result.

The signature creation process also involves signer authentication and if necessary, the trace of authentication is added to the signature.

1.2 Design Goals of the Format

1. *Modularity*: Each server-entity creates certain (independent of others) components of the signature.
2. *Storage-Frendliness*: The signature should be flexibly decomposable and the components should be easily identifiable so that rebuilding the signature remains easy.
3. Order Independence: The order in which the components are stored in signatures is not important, i.e. the real order must be deducible from the components.

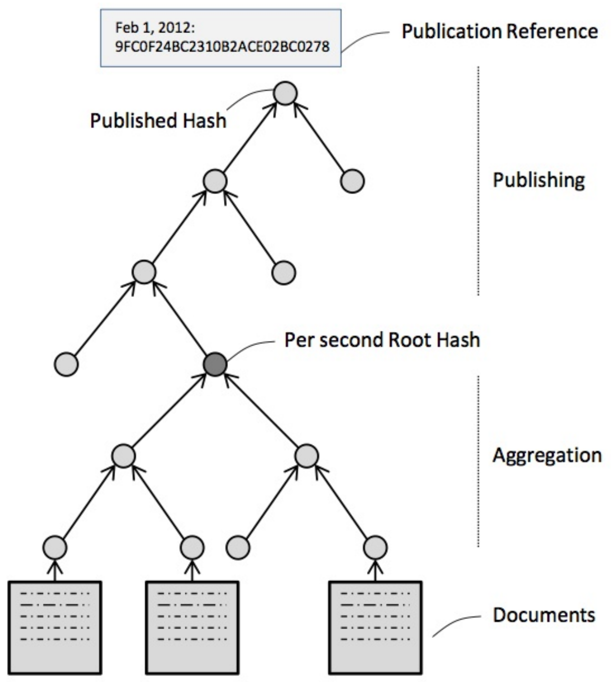

Figure 1: Creation of a hash-tree signature: Documents are hashed, aggregated to per-second root hashes and then published.

4. *Extendability*: It must be possible to define new component types in the future, so there must be a versioning mechanism.

5. *Size Economy*: The size of the signature should be as small as possible. Storage overhead should be avoided.

2 Components

A keyless signature consists of the components of the following types:

- `Aggregation Hash Chain` – Hash chain that represents the computation of a root hash value from an input hash value.

- `Publication Hash Chain` – Hash chain that represents the computation of the published hash value from the per-second root hash value.

- `Authentication Record` – Trace of authenticating a party. Can be an RSA-signature, one-time hash signature, etc.
- `Publication Reference` – Bibliographic reference to the publication.

There may be several components of the same type in a signature.

3 Index

Each component has an `index` field, which is a structured bit-string that encodes the place of the part in per-second global hash tree. Index is used for determining the ordering of the signature components before verification. Index consists of the following types of data fields:

- `time_t` – The UTC time encoded as integer. Identifies the UTC-second that the components belongs to. There may be one or two `time_t`-type fields in every index structure.
- `location pointer` – a sequence of bit-strings that determines the location of the component in the global per-second hash tree.

We will use the following notation for an index:

$$t', t \colon a_1 \ . \ a_2 \ . \ \ldots \ .a_n \ ,$$

where t and t are of type `time_t` (always $t' > t$) and $a_1, \ldots, a_n$ are bit-strings.

3.1 Ordering Components by Location Pointer

Location pointers are orderd lexicographically. We say that a location pointer $i = i_1.i_2. \ldots .i_m$ precedes a location pointer $j = j_1.j_2. \ldots .j_n$, if i is an initial part of j, i.e. if:

- $m \leq n$, and $i_1 = j_1$, $i_2 = j_2$, ... , $i_{m-1} = j_{m-1}$; and
- i_m is an initial part of j_m.

Examples:

$$\begin{array}{lcl}
\texttt{11101.11001} & < & \texttt{11101.11001.10111} \\
\texttt{11101.11001} & < & \texttt{11101.11001011.0110110} \\
\texttt{11101.11001} & \not< & \texttt{11101.11011} \\
\texttt{11101.11001} & \not< & \texttt{11101.110.01}
\end{array}$$

3.2 Components and their Indices

The following rules must be followed when indexing the components (Fig. 2) of a signature:

- *Publication Record* is indexed by a single UTC time value—the publication time.
- *Pulication Hash Chain* is indexed by a pair of UTC time values—the publication time and the creation time.
- *Aggregation Hash Chain* is indexed by the creation time and a location pointer.
- *Authentication Record* is indexed similar to Aggregation Hash Chain.[1]

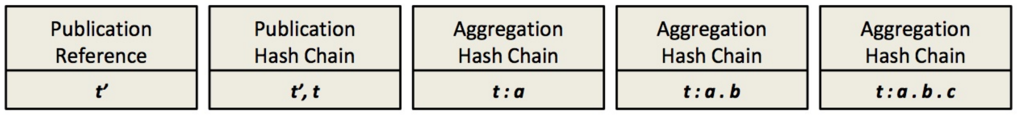

Figure 2: Indexing rules and ordering of the components.

The ordering process of the components involves the following steps:

1. *Sort the components by type*. The aggregation hash chains are in the first group, then the publication hash chain, and finally the publication reference.
2. *Sort the aggregation hash chains* so that their location pointers are in the *decreasing lexicographic order*.

[1]The details are described in the special section about the Authenication Record.

After sorting, the components should be as shown in Fig. 2, where the ordering is from right to left.

4 General Structure of a Keyless Signature

There may be multiple publicatins included in a keyless signature (Fig. 3). There may also be many aggregation branches that correspond to many different documents. This means that the signature format may include a multitude of signed documents, assuming that all of them have the same creation time t (Fig. 3). Keyless signatures may also contain one or more *Authentication Records*. They

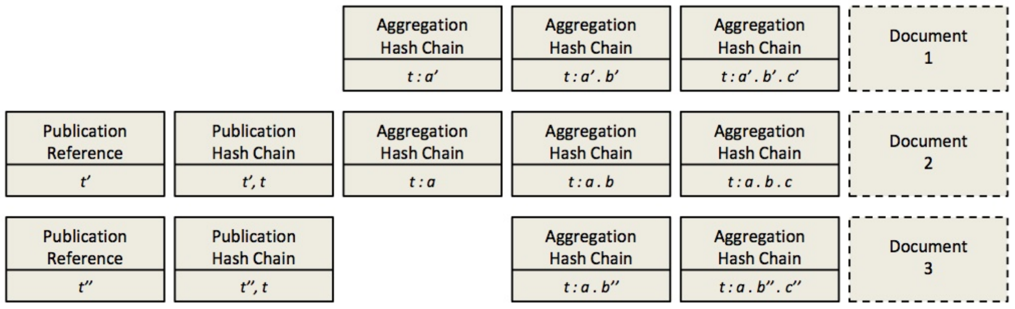
Aggregation Hash Chain t:a'
Aggregation Hash Chain t:a'.b'
Aggregation Hash Chain t:a'.b'.c'
Document 1
Publication Reference t'
Publication Hash Chain t', t
Aggregation Hash Chain t:a
Aggregation Hash Chain t:a.b
Aggregation Hash Chain t:a.b.c
Document 2
Publication Reference t''
Publication Hash Chain t'', t
Aggregation Hash Chain t:a.b''
Aggregation Hash Chain t:a.b''.c''
Document 3

Figure 3: General structure of a keyless signature.

have the same index structure as the other components and belong (are "glued") to the component that has the same index.

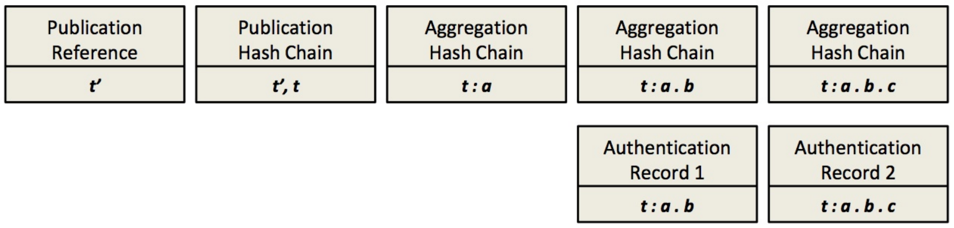
Publication Reference t'
Publication Hash Chain t', t
Aggregation Hash Chain t:a
Aggregation Hash Chain t:a.b
Aggregation Hash Chain t:a.b.c
Authentication Record 1 t:a.b
Authentication Record 2 t:a.b.c

Figure 4: Adding an Authentication Record to a Signature.

5 Structure of the Components

5.1 Imprints

An `imprint` consists of a one byte hash function identifier concatenated with a hash value. Hash functions have one-byte codes that are listed in Tab. 1. The

Table 1: One-byte codes for hash function algorithms.

Code	Algorithm	Output length in bits	Output length in octets
`00`	SHA1	160	20
`01`	SHA2-256	256	32
`02`	RIPEMD-160	160	20
`03`	SHA2-224	224	28
`04`	SHA2-384	384	48
`05`	SHA2-512	512	64
`06`	RIPEMD-256	256	32

imprint structure is formally defined as follows:

```
struct imprint {
   uint8        hash_algorithm
   octet string hash_value
}
```

5.2 Aggragation/Publication Hash Chain

Every hash chain consists of a `input_hash` filed that is followed by a sequence of `left_link` and `right_link` structures.

5.2.1 Link Structure

Each link (either `left_link` or `right_link`) consists of:

- `hash_algorithm` (optional) – hash function that is used to compute the output hash. If missing, the hash function is the same as in the previous hash step.

- `length_balancing_byte` (optional) – positive integer (0..127), set to 0 if missing.
- `sibling_imprint` – root hash of the sibling subtree.

5.2.2 Computation

A Hash Chain represents a computational process. The computation is performed starting from the first hash step and ending with the last one. The input of the computation is an arbitrary octet string and the output is an `imprint`.

Each hash step actually takes as input an arbitrary octet string and a length value (ℓ), and computes an output hash value and the new updated length value $\ell' = \ell + L + 1$, where L is the `length_balancing_byte` of the hash step (Fig. 5).

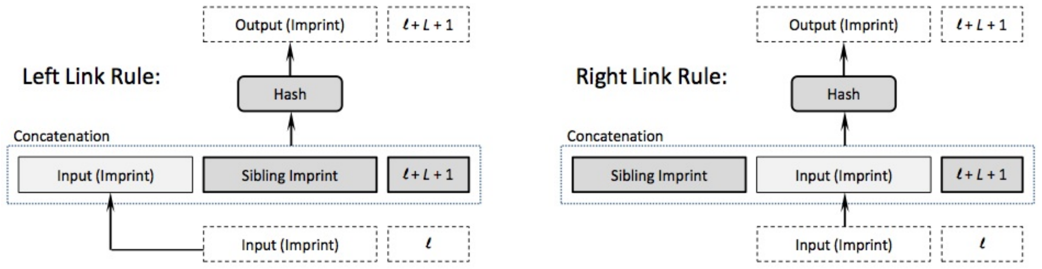

Figure 5: Hashing schemes associated with the `left_link` and the `rigth_link` structures.

The output hash value is computed by applying the hash function (the code of which is represented as `hash_algorithm` of the `left_link` and `right_link` structures) to the concatenation of the input octet string, the `sibling_imprint`, and the new length value $\ell' = \ell + L + 1$.

The initial values of the input hash and the length value are defined as follows:

Initialize: Input: h-input hash, ℓ_0-number of the previously computed links

- The input length value ℓ of the first hash step is set equal to the length ℓ_0 of the previously computed hash chain, or 0 if the hash chain was the first one to compute.

- The input of the first hash step is the `input_hash` field of the `Hash Chain` structure.

Left Link Rule: Input: h-input hash, ℓ-input length, (A, L, S)-hash step data, where A is a hash algorithm identifier, L is the length balancing byte, and S is the sibling imprint.

- Compute $\ell := \ell + L + 1$.
- Compute $h := A \,||\, \mathrm{Hash}_A(h \,||\, S \,||\, \ell)$.
- Output: Output (h, ℓ).

Right Link Rule: Input: h-input hash, ℓ-input length, (A, L, S)-hash step data, where A is a hash algorithm identifier, L is the length balancing byte, and S is the sibling imprint.

- Compute $\ell := \ell + L + 1$.
- Compute $h := A \,||\, \mathrm{Hash}_A(S \,||\, h \,||\, \ell)$.
- Output: Output (h, ℓ).

5.3 Authentication Record

An authentication record represents the result and the trace of authentication. Authentication record always corresponds to either a `left_link` or a `right_link`) structure of a hash chain. The `index` field of the authentication record must be equal to the index of the link.

5.3.1 Structure of an Authentication Record

There are the following fields in an Authentication record:

- `input_hash` (mandatory) – input hash of the authentication record.
- `signer_id` (optional) – the identity of the signer in textual form (utf8-string, etc.).
- `pk_signature` (optional) – a public-key signature of the signer, that contains the follwoing subfields:

- `signature_value` (mandatory) – signature as octet string.
- `certificate_imprint` (optional) – imprint of the signer's public-key certificate.
- `certificate` – (optional) certificate as octet strig.

5.3.2 Public Key Signatures in Authentication Records

Adding a public key signature to the hash tree introduces an additional link structure in the corresponding keyless signatures. The signature is created in the following way (Fig. 6):

1. The output hash of the previous link is signed by using a public-key signature algorithm.
2. The public-key signature is hashed by using a hash function.
3. The hash value (in the form of `imprint`) is defined to be equal to the `sibling_imprint` of the next link structure.

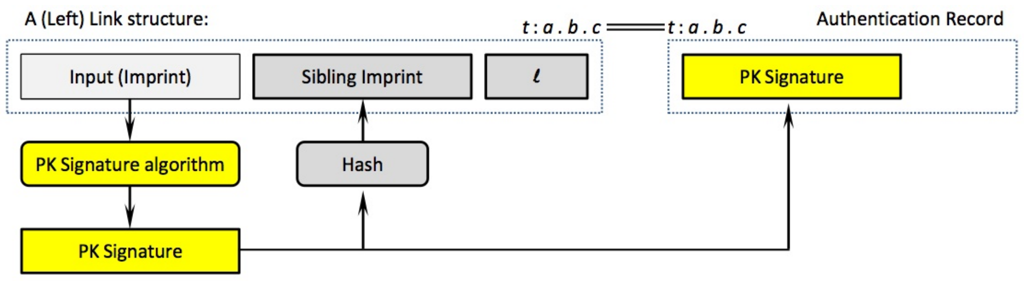

Figure 6: Adding a public-key signature as an `Authentication Record` to a Left Link structure.

A Encoding Rules

A.1 Type-Length-Value (TLV) Encoding

Most objects use Type-Length-Value (TLV) encoding scheme. The Value part of a TLV-encoded object may contain nested TLV objects. Message PDUs are also TLV-encoded objects.

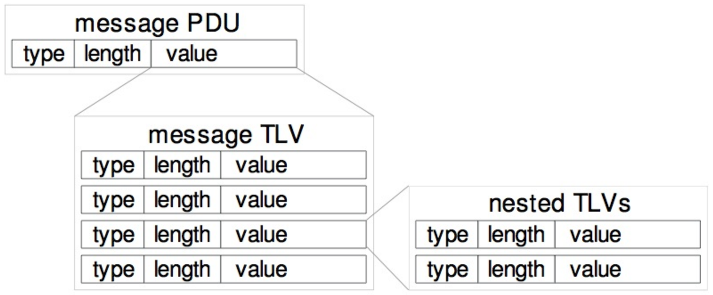

Figure 7: TLV nesting.

Type-Length-Value (TLV) encoding scheme is used to encode data structures and also the messages that is used to transfer them between the entities during the signature generation process. Type encodes how the `Value` field is to be interpreted. Value is an Octet string of Length octets that carries information to be interpreted as specified by the `Type` field.

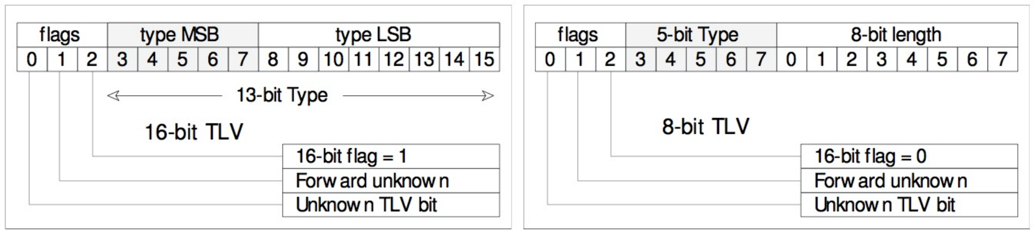

Figure 8: TLV encoding.

For space efficiency two TLV encodings are used. 16-bit TLV (TLV16) encodes 13-bit Type and 16-bit Length. 8-bit TLV encodes 5-bit Type and 8-bit Length. Smaller objects are encoded as 8-bit TLV for lower overhead. 8-bit TLV type has always local significance and identifies the encapsulated structure in the context where it is used. 16-bit TLV type has global significance and identifies the encapsulated structure in the context of the whole signature generation system. Representation of the structures in an implementation software is not specified by this document.

TLV8 and TLV16 are distinguished by a 16-bit flag in first octet of the Type field.

An RFC style MSB 0 bit numbering is used in illustratations. Data words traversing process boundaries must be converted to *network byte order*.

A.2 Bit-String to Integer Conversion

We use the following conversion rule for a bit-string s of length up to $n-1$ to a positive integer of range $1 \dots 2^n - 1$:

1. Concatenate `1` as the most significant bit to s, i.e. compute $s' = 1||s$.
2. Decode s' to an integer in the standard way.

Table 2: Bit-string to integer encoding with $n = 3$.

Bitstring s	$1\|\|s$	Padded	Decimal
	`1`	`001`	1
`0`	`10`	`010`	2
`1`	`11`	`011`	3
`00`	`100`	`100`	4
`01`	`101`	`101`	5
`10`	`110`	`110`	6
`11`	`111`	`111`	7

A.3 Hash Chain

```
TLV [1701] hash_chain {
   TLV [01] index {
      TLV [01] time_t              { uint }
      TLV [02] chain_index         { uint }
      TLV [03] link_index          { uint }
   }
   TLV [02] input_hash               { imprint }

   TLV [04] left_link {
      TLV [01] hash_algorithm        { uint8 }
      TLV [02] length_balancing_byte { uint8 }
      TLV [03] sibling_imprint       { imprint }
   }
   TLV [05] right_link {
      TLV [01] hash_algorithm        { uint8 }
      TLV [02] length_balancing_byte { uint8 }
      TLV [03] sibling_imprint       { imprint }
   }
}
```

A.4 Publication Reference

```
TLV [1702] publication_reference {
   TLV [01] index {
      TLV [01] time_t                { uint }
   }
   TLV [03] utf8_string              { octet string }
}
```

A.5 Authentication Record

```
TLV [1703] authentication_record {
   TLV [01] index {
      TLV [01] time_t                { uint }
      TLV [02] chain_index           { uint }
      TLV [03] link_index            { uint }
   }
   TLV [02] input_hash               { imprint }

   TLV [03] signer_id_utf8           { octet string }

   TLV [04] pk_signature {
      TLV [01] signature_value       { octet string }
      TLV [02] certificate_imprint   { octet string }
      TLV [03] certificate           { octet string }
   }
}
```