KeeeX Universally Verifiable Files

Understanding KeeeX Metadata Statements v2.0

Some features described here are in the scope of pending or delivered patents owned by KeeeX SAS

This document explains and documents the concepts and features made available by KeeeX Fusion v2.0 and complements or replaces the white paper made public under the name White-Paper-KeeeX-keeexed-xopok-nybum.pdf (that can be retrieved by web searching 'xopok-nybum').

Authors: **the KeeeX Team**

Abstract

KeeeX provides tools and technology required to make files verifiable

- in autonomy
- without perceived alteration of their content and behavior whenever possible
- for long lasting durations
- regardless of their format
- using lightweight metadata.

A keeexed verifiable file has the property that unless otherwise stated any bitwise change or addition to the file can be detected, either because the payload, or hash or signature was edited. A file cannot be re-signed and still remain valid, but powerful mechanisms of indirect and delegated signatures are provided. Entire file verifiability also covers the entire freely embedded technical and user metadata useful for automation, labelling, provenance, bindings that were injected in the file.

Making verifiability independent from the file format has paramount importance in that it helps file verifiers to not track file format changes and thus easily retain backward compatibility. Only changes in the specification matter, which very seldom happens. This form of universality is crucial to large scale industrial deployments.

Making files bitwise unforgeable is important, but modern requirements ask for unforgeable files to be attached dynamic information under control (think about newer/latest versions, owner, obsolescence or completion status, requirement to destroy...). This is possible by either

This keeexed file is signed by 1NkZmqDcTKmAWJaJM957HJo84CFHe5JXZn and must be verified on https://services.keeex.me/verify. Its idx (hash) is:
xulip-hivik-fopag-cacyn-dihep-salaf-mevag-zomyd-mihoc-pacoc-negeb-tulih-hefin-coris-zycab-pufec-cuxex

writing inside file themselves when applicable or by by-directionally binding file properties to arbitrary registries (blockchain included). This last feature is patented.

Keeping the verification metadata lightweight has superior importance in the context of generative AI and mass production / streaming / storage of content.

This document describes the concepts and features via the KeeeX metadata language used within verifiable files, possibly including the one you're reading. KeeeX metadata can be easily found using a simple text editor, even on binary files. Reading this should help an engineer understand what is under the hood, or convince themselves that manual verification is possible. It also explains a number of extra features, including how files are linked to their references and versions, how they can be indexed by their embedded hash (called their IDX), how some files may display or exploit this IDX, how files can maintain unforgeable links to variable properties stored in registries or the blockchain and more.

The KeeeX metadata is present in files at positions that do not interfere with its usual behavior, which is almost always possible. For instance, a comment is used to host them in html files and many other program files.

Table of Contents

1. Introduction and Core concepts

The main idea of KeeeX is to achieve the verifiability of a file's integrity, authenticity, date, provenance, metadata in general regardless of it's format, in such a way that a file carries its own proofs without perceived alteration and that the full range of file bytes are protected unless otherwise stated.

Verifiability can be achieved independent from the format thanks to a simple tag based metadata language where an easily identified marker (the pattern `keeex` ... `xeeek`) allows for locating metadata within files, a key metadata component being the `self` statement that contains the file's own hash, as will be explained below.

Making verification independent from file formats allows for implementing and delivering verifiers that stay valid for long durations, major releases of the metadata language being seldom (version 1 has lasted since 2013). Minor releases always preserve upward compatibility and as of V2.0 all files declare the version of KeeeX Fusion (the 'keeexer') that was used to produce them.

Furthermore, the KeeeX metadata language provides a wealth of added value over standard hashing schemes. Most importantly comes the fact that the very own file's hash (called its IDX) can be used as a Digital Object Identifier (DOI) for the file. This is so because this hash will differ whichever bitwise change occurs to the file).

To explain this, consider a file containing embedded KeeeX metadata with:

1. **a hash (IDX)** in a `keeex` `self` statement
2. **a signing public key and a signature** as part of a `keeex` `signature` statement

where a new hash can be computed over the entire file except for the bitwise hash and signature above and compared as equal to 1.

It can be easily assessed that

- a/ if a change to the hash is made, it becomes incorrect,
- b/ if a change is made to the signature it becomes incorrect,
- c/ if a change to any other byte is made then both the hash and signature are wrong and
- d/ no option exists to change the signing authority since it would change the hash itself and result in a file with a new identifier.

Manually performing such edits and submitting the file to a verifier like https://s.keeex.me/verify will perfectly illustrate this.

Added value stems from the fact that the file's 'self' hash is bubble babble encoded for human readability as a list of five letter words where consonants and vowels alternate for pronounceability.

Also useful comes the fact that since a file's hash is plaintext ascii encoded and can be indexed by search engines, it can be used within another file to denote the first file, a possibility called a 'reference', as almost any keeexed file will contain.

For all these reasons and more, the file's hash according to KeeeX has been given a non ambiguous new name: since the early days in 2013, this 'identifier' has been called an 'IDX'. An

IDX thus:

- is computed over both the content of the original file and the content of the metadata.
- excludes parts of the file where hashes, signatures and otherwise reserved space are stored.
- is encoded to facilitate human readability, content based indexing, human exploitation as a kind of DOI (Digital Object Identifier) and inter file reference and linking.

Although there is only one mandatory KeeeX Statement in a keeexed file (the `self` statement), it is rare to find a file with only this statement, as other statements provides important authentication information and details.

As per version 2.0 was also introduced a new multihash algorithm to warrant outstanding durability of evidence.

2. Explaining the KeeeX Statements Version 2.0

This chapter presents the version 2 of the KeeeX metadata statement language as can be found within any file keeexed at this level (if no version is mentioned in a keeexed file this means that it is in v1). This v2 allows for a number of new features compared to the v1, including parallel additive multi hashes, incremental or multi level keeexing, delegated signatures, registry mapping (that falls within the scope of a new granted and pending patent) and a replacement for 'protected' statements: the 'reserved' statements.

For readability in the sequel, the syntax will not be presented formally but rather by the example. It is interesting that you open a keeexed file on the side in a text editor to match this literary content. Again, maybe the one you're currently reading.

2.1. Rationale

The KeeeX metadata language was crafted to obey a number of key principles. Here are a few.

2.1.1. Frugality

Because the KeeeX technology is used by the industry for file formats that can result in very small files (e.g. markdown, html, pdf, xml, json...), and also because it should be used in network intensive situations like social networks or streaming, the metadata size added to the initial file payload should not be exaggerated. Indeed, KeeeX metadata size typically falls within the 1K-2K range per file. Specific implementations may even be designed to equip files with less than 300 bytes of metadata.

2.1.2. Auditabity, readability

Also it was chosen since inception that KeeeX metadata should be plaintext human readable even in binary formats, for auditability and indexability reasons. No obfuscation or obscure encodings come to play.

2.1.3. Security, resistance to injection attacks

Because a keeexed file's hash resides inside the file itself, it's value cannot be computed over the entire file. It was chosen however to preserve the default property that any change made to a file from byte 0 to end can be detected, unless the user decides otherwise.

A special kind of statement called `reserved` allows for excluding more space from the hash computation under strict control. Since reserved statements can be signed (these signatures where possible in v1 but they are enforced by the syntax in v2) change detection from byte 0 to end remains true. Reserved statements also mention their "purpose", which allows for human verification of adequacy without further automation of semantic verification. Bindings between reserved statements and delegated or indirect signatures also enforce added control.

Under those settings, a keeexed file's embedded hash (IDX) is indeed a digital objet identifier.

2.1.4. Human use, indexability

Everybody knows that hashes appear as binary garbage and cannot be referred to in a conversation or at least part of them memorized. Within plaintext KeeeX metadata, IDX are Bubble Babble encoded so that they are pronounceable, memory and search engine friendly which proves outstandingly useful since a file's hash actually behaves as the unique identifier for a unique file.

Furthermore, KeeeX helps users clone the file's hash in positions exploited by file processors. For instance, if you view this file under the form of a keeexed html, you should read it's very own IDX here :

xidev-vepid-zicud-hopog-kiloz-kasuh-sycim-fofel-renym-fonur-hohig-zeneg-cogol-narys-cafeb-lopan-lixix

otherwise the placeholder text

zuzi-ntinen-cudi-bfudab-poru-hsalup-leti-nsehoh-polu-cbazec-dahy- rgogun-roke-hvusin-fato-ztymabf-exar

will show up, a behavior that will be explained later.

If you read a keeexed pdf version of the same content, the author may have taken the provision to display the file's identifier in a overlay at page bottom for instance. You can test searching your disk for the first words in the matching name.

2.1.5. Sovereignty

A KeeeX verifier remains a simple program that only scans files in search for metadata statements. It is so simple it can easily work within web pages that never leak information outside beyond optional hashes (when time-stamping or blockchain anchoring was used) and perfectly operate offline.

Of course indeed the KeeeX tool suite also lets our users produce files on premises in absolutely sovereign conditions, no matter if backend, mobile or web based apps are used.

2.1.6. Durability

Some files require very long lasting protection against forgery. Think for instance about pay slips, or diplomas. KeeeX addresses this by way of a novel parallel cascading multihash algorithm that helps the user freely combine the effects of chosen hashing algorithms in a way that differs from usual parallel or piping combinations. This is detailed in annex 4.

2.2. Quick overview of the KeeeX metadata statements

KeeeX proposes a limited list of metadata statements :

- `self` : this is where file hashes or IDX reside
- `name` : a place to declare names attached to the file (translations included)
- `ref` : where references to other files or concepts are set
- `signature` : this is where the signatures of hashes or reserved data reside
- `prop` : the statement used to attach key/value properties to a document
- `reserved` : a statement to prepare free space for data that will be known after the hashes are computed`
- `license` : signed information about the license of the user who created the file

Every statement has one or two values plus optional extra parameters for semantic details.

2.3. Examples

The two examples below illustrate how KeeeX metadata show up in usual files. 256 bit hashes, digital signatures and even timestamps are fairly long so they have been redacted for "clarity" by escaping most of their text using `[...]`. Also escaped are confidential details in descriptions or names.

- Predefined property names start with `"kx."`.
- Provisional or beta property names start with `"prekx."` (these properties are not documented here until they are promoted but future verifiers will commit to extract and display their values)

- User defined custom property names are free (as `AIEdited` below : an example of AI Labelling) .

Note : in the examples below, as possibly in other code samples in this document, any occurrence of the string `keeeX` is a replacement for `keeex` used to prevent undue processing of sample metadata by a KeeeX processor, in the intent to make the edit clearly visible

2.3.1. Metadata in a document file keeexed using KeeeX Fusion

```
keeeX self xuket-sucop[...], {kx.main:2,kx.app:@keeex/js-
fusion@7.1.4,kx.decode:raw,kx.format:raw} xeeek
keeeX self 0a0[...]bd5, {kx.algorithm:sha256,kx.encoding:hex,kx.recursive}
xeeek
keeeX license "KeeeX|xonaf-nyzuc[...]|prod", "2025[...]26Z|HEA[...]tg=",
{kx.mode:online} xeeek
keeeX name "[...]", {main} xeeek
keeeX prop "kx.description", "[...]" xeeek
keeeX prop "kx.author", "[...]" xeeek
keeeX prop "prekx.aiTags","[...]" xeeek
keeeX prop "prekx.aiGenerated","false" xeeek
keeeX prop "AIEdited", "5%", {prompt:translate to EN,model:gemma3n:e4b} xeeek
keeeX prop "kx.afterbc", "BC|ETH.main|23240150|288[...]d5a|2025[...]19Z" xeeek
keeeX signature "19r[...]UJ5", "2025[...]78Z|HIGt[...]XI=",
{kx.sigType:bitcoin} xeeek
keeeX signature "162[...]qRM", "---[...]---|cGx[...]k0=", {kx.sigType:bitcoin}
xeeek
keeeX prop "kx.time", "2025-08-28T14:17:36.656Z" xeeek
```

This file also illustrates the presence of still pending signature (the second).

2.3.2. Metadata in a Photo file produced and keeexed using Collect And Prove

```
keeeX self xetez-tabyv-[...], {kx.main:2,kx.app:@keeex/js-
fusion@7.2.1,kx.decode:raw,kx.format:jpeg} xeeek
keeeX self 91b[...]479, {kx.algorithm:sha256,kx.encoding:hex,kx.recursive}
xeeek
keeeX license "License[...]25|xolik-[...]|prod", "2025[...]19Z|HNd1a[...]pqQ=",
{kx.mode:offline} xeeek
keeeX name "Collect And Prove Media [...]", {main} xeeek
keeeX prop "kx.description", "" xeeek
keeeX prop "kx.author", "Collect And Prove App" xeeek
keeeX prop "prekx.appName", "Collect And Prove" xeeek
keeeX prop "prekx.appOs", "android" xeeek
keeeX prop "prekx.appVersion", "10.8.0" xeeek
keeeX prop "prekx.aiTags","car,door,bump", {model:tensorFlow} xeeek
keeeX prop "prekx.aiGenerated","false" xeeek
keeeX prop "aiEdited", "0%" xeeek
keeeX prop "KEEEX_UNIQUE_SCENARIO_KEY", "841[...]00c" xeeek
keeeX ref robez-[...]-nixor, {kx.feature:geolocation} xeeek
keeeX prop "kx.geolocation.status", "ok", {kx.id:0} xeeek
keeeX prop "kx.geolocation.accuracy", "12.86400032043457", {kx.id:0} xeeek
keeeX prop "kx.geolocation.altitude", "122.08001708984375", {kx.id:0} xeeek
keeeX prop "kx.geolocation.heading", "0", {kx.id:0} xeeek
keeeX prop "kx.geolocation.latitude", "[...]", {kx.id:0} xeeek
keeeX prop "kx.geolocation.longitude", "[...]", {kx.id:0} xeeek
keeeX prop "kx.geolocation.timestamp", "2025[...]000Z", {kx.id:0} xeeek
keeeX signature "193T[...]bm4", "2025[...]81Z|IFj[...]MMs=",
{kx.sigType:bitcoin} xeeek
keeeX signature "13qP[...]AgY", "2025[...]85Z|H2X[...]JM=",
{kx.sigType:bitcoin} xeeek
keeeX prop "kx.time", "2025-08-28T13:30:42.146Z" xeeek
```

2.4. Syntax Overview

A KeeeX statement is located between two keywords: `keeex` and `xeeek`. The content of a statement follows this syntax:

`keeex` `<type> "<value1>"[, "<value2>"][, {extra}]` `xeeek`

Values can optionally be enclosed within delimiters (see below). `value2` is optional, depending on the statement type. `extra` is a key-value structure whose format is described in a following section. It is enclosed in brackets.

At verification time the entire statement text will participate in the hash computation except for the `self`, `signature` and `reserved` statements. In all cases however the hash computation will at least span over the statement 'head' that includes the statement type and over the tail that includes the 'extra' :

`keeex` `<type>` and `[, {extra}]` `xeeek`

This prevents any keeexed file from a tentative edit that would add or remove a statement yet keep the same IDX value.

2.5. Text encoding

Statements are ASCII-only text. Non ASCII characters (code point over 127) are encoded using URL-like escape sequences. The only exception is the `%` character (37) encoded as `%25` to make the transformation fully reversible.

In addition to this encoding, all value strings (`value1` and `value2`) are escaped as JSON strings, which means that double quotes are escaped, and the escape symbol (`\`) is escaped too.

Some injection methods cause specific modifications to the injected text. If decoding the written metadata requires more conversion than the default, the kx.decode extra will be present in the main `self` statement. If present, the requested decoding is done before returning effective data to the user.

2.6. Statements location

KeeeX statements may be found at different positions in a file, depending on the file format and its requirements. Although this is often the case they may not participate in a contiguous block.

Statement location or ordering has no impact on verification, since a verifier solely needs to collect all statements by a one time scan of the file to operate.

Typical positions are

- in a location suitable to not alter the original data payload and any file processor's behavior. Think for instance about a comment in a programming file.
- if possible in a place that might be preserved by file editors across modifications, which helps tell a modified file from a non-keeexed file. This cannot always be achieved.
- in explicit free to user convenience metadata space. Think for instance about EXIF or Dublin Core metadata in media or office files.

2.7. Statement values delimiters

The valid delimiters for statement values default to double quote (`"`) or otherwise HTML-style escaped double quote (`&quot;`). When one style is used on a set of statements, it remains consistent across all statements. Double quotes are used whenever possible for the sake of readability of metadata.

2.8. Extra statement property

The `extra` property that can be optionally found in a statement defines a key-value object. It is based on JSON with the following restrictions: values can only be number, boolean or string, but not nested objects (the extra thus appears as a 'flat' object).

Extra properties whose key starts with `kx.` are reserved and never convey user defined semantics.

3. Statement types

This section lists all the supported types of statements, along with their meaning and usage, and their optional extra properties.

The only mandatory statement in a file is a `self` statement containing the file's IDX.

Unless stated otherwise, statements are kept in full when computing a file's IDX.

KeeeX provisions for the possibility to `keeex` files at several different 'heights' which means that unless stated otherwise, the statements with a height lower than the topmost height value for the file will be ignored. This serves for several purposes :

- mainly to `keeex` files that may embed other previously keeexed files, as can be the case for a zip archive, or an editor file containing embedded pictures
- add a signatory or other metadata to an existing file without changing content
- also for the technical usage of keeexing the same file multiple times after edits. This use case may however in many cases be better served by the use of a reference to a previous version.

3.1. Extra property valid with any statement type

The following property can be set in the `extra` of any statement.

3.1.1. `kx.height` extra (numeric)

If missing, it defaults to 1. It indicates at which keeexing "height" the statement is located. When verifying a file, only statements of the highest height are to be used; lower level statements being discarded.

If a composite file contains other subfiles that can be extracted, the composite file can be verified at it's own height, and extracted subfiles as well. This is so with office and zip files.

3.2. Statement `self` (file identifier)

This statement contains file identifiers, computed by using hash algorithms or combinations.

The identifier is set in `value1`, without double quotes. There is no `value2`.

There are two kinds of `self` identifiers:

- The main identifier is the file's IDX. It's extra has the numeric property `kx.main`. There is exactly one main `self` per height.
- Other identifiers, computed using different hash algorithms and encodings that may be present for special purposes.

To be valid, all identifiers for a given file must match the current file content minus the placeholders reserved for `self`, `signature` and `reserved` statements.

Note that the current specification already supports a parallel cascading multihash for the main `self`. The presence of multiple `self` in a file normally does not account for cybersecurity (to strengthen the un-forgeability of the file). Instead to serve this objective the proper algorithm sequence should have been specified for the main IDX (the multihash must involve at least two algorithms, as explained in annex 4).

3.2.1. IDX computation for verification

Verification starts with computing the main self and comparing it to the value present in the file.

When computing a file identifier, all instances of the content of `value1` for `self` statements are fully skipped in the whole file, including in the `main` self statement itself. Also skipped are content present into `signature` and `reserved` statements, which will be described in the dedicated paragraphs.

It is worth noting that KeeeX allows for replicas of the IDX to appear in the file at more locations than in the `keeex` `self` statement. This is used to place a copy of the IDX in a position that can be exploited by a processor, for instance to display a QR Code of an url containing the IDX when rendering an html page, or to display the IDX as part of a pdf overlay. The file you are currently reading possibly illustrates this.

3.2.2. `kx.main` extra for 'self' (numeric)

Denotes the 'main' IDX identifier for the file. Every set of statements for a given height has one and only one `kx.main self` identifier. The value is the statement specification's major version, which is 2 for files keeexed under the current specification. Since `kx.main` was a boolean in v1, the absence of a value denotes a file keeexed in v1.

3.2.3. `kx.subversion` extra for 'self' (numeric)

Indicates that the file uses a subversion of the main statement versions. Only features that change the IDX calculation require new subversions. A verifier checks if it knows the currently used subversion, and displays a suitable error message when it does not. Files have no subversion indicator if not required.

3.2.4. `kx.app` extra for 'self' (string)

The name of the application used to create the file. For `js-fusion` for instance, this should be something like `@keeex/js-fusion@6.0.0`.

3.2.5. `kx.archive` extra for 'self' (numeric)

Indicates that this `keeex` statement level was created using the given archive level. Archive mode for keeexing is an option that has no impact on verification. It is used for fast archival without concern for the file behavior or appearance. A file keeexed in archive mode can easily be reverted to its original.

3.2.6. `kx.id` extra for 'self' (numeric)

Identifies this `self` for reference in signatures or external processes. The value defaults to 0 which applies to the `kx.main self` alone, where it is left unmentioned. Explicit values of `kx.id` start at 1. No two `self` or `reserved` statements may have the same `kx.id` value at a given `kx.height`.

3.2.7. `kx.algorithm` extra for 'self' (string)

Specifies the algorithm required to compute the identifier. Defaults to the value specified in the annex 4 : `sha3-256<sha256` (sha3-256 being the final output)

Valid hash algorithms include "`sha224`","`sha256`", "`sha512`", `sha3-256`, `sha3-512` plus any combination of valid hash algorithms separated with a `<` to use multihash (as described in the computing IDX annex).

No `kx.main self` may use a multihash involving fewer than 2 algorithms from the valid list. This prevents against a malicious actor attempting to attack a given IDX by using a weaker algorithm.

3.2.8. `kx.encoding` extra for 'self' (string)

Specifies the encoding used to render the IDX. It defaults to the value specified in the computing IDX annex 4.

Valid encodings include :

- "b58": `base58`
- "bubble": `bubble-babble` (the default)
- "hex": `hexadecimal`

3.2.9. `kx.decode` extra for 'self' (string)

The description of an injection method to indicate how data must be decoded to retrieve the correct initial value. This property is named `kx.decode` and defaults to `raw` if not specified. The possible values are:

- `raw` : no special modifications
- `xml` : escapes the characters using basic html entities encoding (`>` and `&` at least are encoded with their equivalent `>` and `&`)

3.2.10. `kx.recursive` extra for 'self' (boolean)

If present, this `self` is recursive, meaning it is computed on the self main value instead of on the file contents. This is used to obtain a quickly computed IDX from the main in technical contexts that require so as for instance time-stamping in some cases.

3.3. Statement `name` (file name)

Used to indicate various "readable" symbolic names associated to the file. This statement is optional, but can be used to display names in automated processes.

It also allows for files to stay immune to renaming in contexts where a storage system for instance would generate a seemingly random name for the file.

3.3.1. `kx.main` extra for 'name' (boolean)

Can be used to indicate to a processor a default name to display to users.

3.4. Statement `prop` (file property)

This allows for the storage of arbitrary key-value properties within the file. Properties allow users to record within the file any element pertaining to the context in which the file occurs : a container number for a photo, a prompt or nonce for AI generated content, a label saying whether a file is AI or human generated... More generally any possible Data.

The key is set in `value1` and the value in `value2` . Properties whose name starts with `kx.` are reserved and never used as user custom properties.

For a list of predefined property key names, see the annex 3 on special properties.

3.4.1. `kx.base64` extra for 'prop' (boolean)

If this property is present, the content of `value2` is expected to be encoded in base64 and must be decoded for the user to retrieve the original data.

3.5. Statement `ref` (IDR reference)

Contains a reference to something (file, concept, tag, etc.) using an IDR. The IDR is set as `value1` and reference semantics can be set in the `extra` .

The syntax for an IDR is a bubble babble encoded IDX where the leading and trailing 'x' chars (that are non significant) are replaced by 'r'. This allows for indexing and search engines to discriminate among the file instances that match some IDX and the ones that refer to them.

3.5.1. `kx.previous` extra for 'ref' (boolean)

The special prop named `kx.previous` is used to indicate a reference to a previous version of the file.

3.5.2. `kx.feature` extra for 'ref' (string)

Is used to identify that a specific feature was used in the file. The value of that extra is a hint as to what the feature is.

Some special `kx.feature` values are found in the special refs annex 2, alongside with their interpretation.

3.5.3. `kx.link` extra for 'ref' (string)

Indicates that the ref is a generic link with the role described in the extra.

3.6. Statement `signature` (digital signature)

Represents a single signature, of a `self` or `reserved` statement. Digital signatures are usually made using a private/public key-pair. KeeeX signature statements store three informations:

- an "address", related to the public key
- the "signature" itself
- the type of signature

The notion of address comes from using bitcoin-message signatures, and initially represented what is actually called an "address" in this context. As of this specification this is a broader term to identify the public part of a public signature key, in a way that makes it possible to verify the signature.

The signature value has two fields: a fixed-size ISO-8601 date-time (down to the millisecond), and the signature itself. They are separated with a `|`. The date is *always* written as UTC with the `Z` timezone indicator (this warrants a fixed size).

Example: `2021-08-20T09:18:33.379Z`.

The type of signature is stored as a string value in the extra with the key `kx.sigType`.

3.6.1. IDX computation

When a signature statement is cleaned for computing identifiers, the signature (the `value2`) is removed, including the delimiters (double quote or other).

3.6.2. Signature subject

By default, a signature is computed on the `main` IDX of the file. Optionally, an `extra` property named `kx.topicId` can be used. In that case, the signature is computed on the statement with a matching `kx.id`.

The whole signature subject is the content described in the previous paragraph prefixed with the timestamp string described above.

3.6.3. Signature placeholder

It is possible to `keeex` a file without computing all signatures at the same time; in this case, a placeholder of the appropriate size is set in the `value2` field. This is useful when a file must be circulated among signatories.

It is thus not always an error to encounter a file where some signatures are missing, notably in the signing process itself. Within a signature placeholder, the date-time is set as `----------T----------` which makes obvious the fact that the signature is missing.

The signature placeholder itself is a string that ends with the expected signing address and is padded to the left by repetitions of the string "placeholder".

3.6.4. `bitcoin` signature (parameter value of `kx.sigType` extra for 'signature')

These signatures are based on bitcoin-message signatures. This is the format typically used to sign classic bitcoin transactions, and includes the mechanisms to convert a public key to an "address" (which is used as the `address` field).

All implementations are compatible with this kind of signatures.

3.6.5. `x509` signature (parameter value of `kx.sigType` extra for 'signature')

These signatures use standard protocols to create digital signatures. The public key is backed by a full certificate. The check of the certificate's emitter is left to the verification process.

For these signatures, the `address` is the full X509 certificate in base64 with the BEGIN/END delimiters (PEM). The `signature` itself is also stored in base64.

3.6.6. Complex addresses

Some signatures can have a special address string which requires further interpretation before verification. These cases are described in the sections below, according to the following:

- If a `signature` statement `extra` contains the `kx.indirect` boolean property, then the `address` field is used differently : it represents a trusted peer which must have proposed the actual address to use for computing the signature. This requires an auxiliary reserved statement.
- If a `signature` statement `extra` contains the `kx.delegate` boolean property, then the `address` field is checked differently.
- If both `kx.indirect` and `kx.delegate` are present, `kx.indirect` is resolved first.

3.6.6.1. `kx.indirect` extra for 'signature' (boolean)

Instead of storing the actual signature address in the statement, an indirect reference is used. This reference relies on a known third-party to "link" this reference to an actual address.

The reference is a JSON string representing the following object:

```
{
  "ref": "<text reference>",
  "trustedPeer": "<trusted third-party address>"
}
```

In the presence of this kind of address, the actual address used is resolved by looking for a `reserved` statement with the `kx.indirectSignature` extra and whose `value1` equals `<text reference>`. Once found, its `value2` is read and interpreted as the concatenation of the actual address to use and a signature by the trusted third party separated by `|`. The full content of a complete `value2` entry for this `reserved` statement looks like:

```
<actual address>|signature(<IDX>|<text reference>|<actual address>)>
```

The `value2` of the `signature` statement must be a valid signature of the `IDX` by `actual address`.

This scheme has the following properties:

- The text reference and the trusted peer are known at the time of writing the metadata, and are part of the IDX
- The effective address used is confirmed by the trusted peer on a file per file basis and can't be reused spuriously (the trusted peer signs a string that contains the IDX)
- Effective signatures can use an address that is not known at the time of keeexing

3.6.6.2. `kx.delegate` extra for 'signature' (boolean)

The `value2` of a `signature` statement having the `kx.delegate` extra may possibly have been computed by a different address than the address `@A1` in `value1`. (this feature is only available for bitcoin type signatures)

If a delegation was used the effective address is resolved by:

- searching for a `reserved` statement with the extra prop `kx.delegation` and `@A1` in `value1`
- processing `value2` as a comma separated sequence of delegation information in the format specified below. Each object in the sequence (if valid) replaces the current address by another address, ultimately providing the address to check the `value2` of the `signature` statement against.

Each delegation information object is a string that represents the concatenation of the following elements separated by a `|` character:

- delegator address in its text representation as usually used in a `signature` statement (first object uses `@A1`)
- delegate address
- signature role, if applicable (if not, an empty string is used)
- a date indicating the "not before" validity date of the delegation, encoded in ISO8601 (only the date portion) (inclusive)
- a date indicating the "not after" validity date of the delegation, same as the previous field, also inclusive
- the signature of all the previous fields by the delegator address, without any concatenation characters

These elements allow for multiple levels of delegation. No operational semantics are attached to roles, in particular in relation with the `kx.role` extra. This is left to final users. The role here is an information about the signature itself, not the signing entity. For instance this can be a string as "Validation".

Unlike in the indirect signature case, delegation objects can be reused across multiple files, thereby allowing a delegator to transmit signing authority.

3.6.7. `kx.topicId` extra for 'signature' (numeric)

When a signature is not computed on the main IDX, this property indicates on which statement the signature is computed. It can reference either a `self` or a `reserved` statement. For `reserved` statements, the signature is applied to `value2`.

Note that signing reserved statements restores the bitwise verifiability of files, since the content of such reserved payload (that was not know when the file was initially processed) can still be verified in the end.

3.6.8. `kx.role` extra for 'signature' (string)

A signature statement can provide a purely informative role indication (as e.g. "Validation", "Endorsement", "Proof of reading"...), which is user-provided and of no significance to the usual signature verification process and has no relationship with the possible status of the signing entity.

Such a role is indicated in the `kx.role` extra as an ASCII-string whose length is limited to 50 bytes (usual string encoding applies).

KeeeX assigns no hierarchy semantics to signature roles, even in the context of delegation.

3.6.9. `kx.name` extra for 'signature' (string)

The name to display as the digital signature's identity. This value is purely informative, as it is not actually digitally signed or endorsed by any kind of third party beyond the file creator himself.

3.6.10. `kx.ts` extra for 'signature' (boolean)

A signature can have the `kx.ts` extra property set; This tells the verifier that in addition to the system timestamp present in the signature placeholder, a timestamp was created on KeeeX's server and should be available to KeeeX tools by an api call. Verification thereby requires calling KeeeX's API to check the timestamp of the hash of the following elements concatenated as a string:

- signed identifier (usually the main IDX)
- signature timestamp present into the signature placeholder
- signature address
- the signature itself

3.6.11. `kx.optional` extra for 'signature' (boolean)

Indicates that the related signature is not mandatory to identify the file as fully valid. A file will always needs at least one valid signature of the main self to be considered valid.

3.7. Statement `license` (user license)

The `license` statement is made of two values that summarize the user's license. There are two modes of operations at creation time: online and offline.

3.7.1. Online mode

When online mode was used at keeexing (i.e. by not providing an offline license file), the license informations were retrieved from the KeeeX API and embedded in the keeexed file.

- `value1` is the license text and the active user profile IDX, concatenated with a `|`: `${licenseText}|${userIDX}|${target}`
- `value2` is the signature (similar to signature statements) of `value1`: `${signatureDate}|${signature}`

When the signature is computed by the KeeeX's server, the signable is prepended with the signature's timestamp, so the real signable value is `${timestamp}|${licenseText}|${userIDX}|${target}`. For the signature, the `timestamp` is formatted in ISO8601.

This statement has the extra property `kx.mode:online` set. Fusion's implementation should ensure that the signature timestamp is no further than an hour away from the current time (it should renew it regularly, ideally at most every 30 minutes).

3.7.2. Offline mode

In offline mode, KeeeX Fusion depends on a license file to be provided. This file is keeexed and signed by KeeeX's license key, and also contains a secondary signature similar to the one used in online mode. There are two differences with the online mode:

- the signature timestamp is frozen at the license creation time
- the extra property is `kx.mode:offline`

Offline license avoids checking the relative distance between the license's signature's timestamp and the file's creation time.

3.8. Statement `reserved` (reserved space with restrictions)

`Reserved` statements allow for holding content unknown at file creation time within their `value2` parameter. It is frequent that such statements contain padding text since some file formats do not resist changing length.

Reserved fields with pre defined specific meaning are described in annex 1.

3.8.1. IDX computation

When computing the file identifiers, the content of `value2` is removed, including the delimiters (they are handled in a way similar to `signature` statements).

3.8.2. `kx.id` extra for 'reserved' (numeric)

A property named `kx.id` in the extra of each `reserved` helps to identify them formally. This identifier is used so that a signature can target a the data provided in `value2`. In the case of `reserved` statements, the signature applies to the following:

```
${mainIDX}|${value2}
```

The full length of `value2` is used in the signature (including padding if present).

3.8.3. `kx.htmlComment` extra for 'reserved' (boolean)

When present, the content of `value2` is surrounded by `-->` and `<!--` and these are removed before retrieving the actual data. This feature is useful to "insert" dynamic data into HTML files.

3.8.4. `kx.transaction` extra for 'reserved' (boolean)

When present, `value1` identifies a blockchain (or blockchain-like structure) known to KeeeX verifiers, and `value2` indicates a reference on said structure.

3.8.5. `kx.base64` extra for 'reserved' (boolean)

If this property is present, the content of `value2` is expected to be encoded in base64 and must be decoded for the user to retrieve the original data.

3.8.6. `kx.flexible` extra for 'reserved' (boolean)

If a `reserved` statement has this extra property, it is allowed to change size depending on the effective content. The absence of this property indicates that the statement can't change size, and if the actual data is shorter than the reserved space padding is used to keep it at the same length.

3.8.7. `kx.indirectSignature` extra for 'reserved' (boolean)

Indicates that this statement contains informations pertaining to an indirect signature.

3.8.8. `kx.delegation` extra for 'reserved' (boolean)

Indicates that this statement contains informations pertaining to a delegated signature.

4. Computing identifiers (IDX, self)

When computing the identifier of a file, we process all data except for specific parts using hashing algorithms.

Regarding the main self statement:

- files keeexed with the version 1 of the KeeeX statements used by default `sha256` encoded in bubble-babble. The hash also accounted for a salt appended at the end of the

file.

- Files keeexed with version 2 default to using `sha3-256+sha256` encoded in bubble-babble for the main idx, and do not use a salt.

The algorithms are still listed in the file, but they are enforced by the version of the statements. Further revisions of the statements may allow other families of algorithms to be used with V2 than the ones listed in annex 4.

The input for the hash algorithms is the 'cleaned' file where :

- parts of `self`, `signature` and `reserved` statements are skipped as described above
- replicas of the file identifiers (either the placeholders, or the effective identifiers) are skipped

Computing a "multihash" result is fully described in the annex 4, but boils down to this:

- for each hash algorithm, process the (cleaned) file as usual
- in the order specified append the output of the hash as input data for the next hash and so on
- the output of the last hash is the result

5. Conclusion

This document wishes to introduce and back the principles of KeeeX and to provide enough information for a curious user to understand the contents of the KeeeX metadata that may be found inside a file (as for instance by opening this file if keeexed in a raw text editor).

We hope that the details given also make the reader confident that in an ultimate situation, it would be possible to verify a keeexed file "by hand". This is so because the way hashes and signatures are computed is detailed and the algorithms used are named in cleartext in the file.

6. ANNEX 1 Special reserved fields

`reserved` statements have two properties, `value1` and `value2`. While `value2` can change at any time, `value1` cannot.

Reserved statements are used to store data not known when the IDX is computed.

This annex describes special `reserved` statements and how they should be read. Special reserved statements all have a special `extra` property that starts with `kx.` and are described below.

6.1. `kx.indirectSignature` extra for `reserved` statement

Used for signature address indirection.

`value1` must match the address text reference, while `value2` must contain the actual address to use and its signature by the trusted peer.

6.2. `kx.delegation` extra for `reserved` statement

Used for signature address delegation. See the `signature` statement for more informations.

`value1` is the originally expected address. `value2` is the delegation informations, signed by the original address. Note that there can be multiple delegation informations, the maximum depth being limited at keeexing time.

6.3. RFC3161 timestamps

If `value1` is "RFC3161", then `value2` must hold the full RFC3161 certificate in base64

Statement also has the `kx.base64` extra.

6.4. Blockchain anchoring

`value1` can be `bitcoinTree`, in which case `value2` may contain the Merkle branch proof extracted from a Merkle tree built at the time of anchoring. The Merkle branch connects the file's IDX to a hash stored as OP_Return Data in a transaction on the Bitcoin network for perpetually verifiable proof of existence.

7. ANNEX 2 Special refs

The references below have a special meaning when found in a file.

7.1. Rollup files (certificates)

`relec-toluz-podys-hofev-posav-sypim-nymol-fokob-racor-deded-zibus-cyrad-typov-nafed-gadif-bidov-kixir` This is a legacy ref, but still used in rollup certificates. It indicates that some extra info are found in the certificate itself, like its btcId. In V2 the `reference` statement may optionally have the `kx.feature:rollup` extra.

7.2. Non production files

`rofet-peryg-ditib-motem-getaf-hubyr-vydim-zezes-fizut-nytib-lokyl-bobif-deled-nysov-zukes-cocih-vexer` (together with the `kx.feature:nonproduction` and `kx.target:demo|test|msg` extra)

Used in files whose license is identified as a demonstration/test license. The `kx.target` extra is expected to be `test` or `demo`, but can be any string. This may also be used to *manually* mark a file as non-production if needed, hence the separation from the license statement.

7.3. Geolocated files

We have two versions of the geolocation informations. The legacy version (in files produced by the PhotoProof mobile app) and the new version.

As of this V2 files with generic geolocation info should have the following feature ref: `robez-sutih-fufyz-mifev-fatyk-musuk-nugas-vivab-suzoh-pyvec-soges-tofyn-pidug-gucac-lanep-diveb-nixor` (with the `kx.feature:geolocation` extra).

Files with this ref will have the following props:

- `kx.geolocation.timestamp` : UNIX timestamp of the GPS data (optional)
- `kx.geolocation.status` : status of the geoloc:
  - "disabled" if the device have geolocation feature disabled
  - "mock" if a mockup location is detected
  - "ok" if everything is nominal
- `kx.geolocation.latitude` : latitude, positive means north (optional)
- `kx.geolocation.longitude` : longitude, positive means east (optional)
- `kx.geolocation.accuracy` : accuracy of the position, in meters (optional)
- `kx.geolocation.altitude` : altitude in meters (optional)
- `kx.geolocation.heading` : compass orientation (optional)
- `kx.geolocation.label` : label to associate to the geolocated point (optional)

7.4. Mail+Attachments zip files

These files contain the reference :

```
rilot-kulum-duzid-relip-figun-taden-rodun-nibop-cafob-gibad-tumur-bylyh-mygah-mutag-pahan-bovim-maxur
```

Must be present with the extra `kx.feature:mailattachment`.

When present, the file is expected to be a keeexed ZIP file containing a mail. The expected content:

- `metadata.json` : see below
- `body.html` : the body of the mail in HTML form (either this one or `body.txt` must be present)
- `body.txt` : the body of the mail in text form (either this one or `body.html` must be present)
- `attachments/*` : attachment files

The `metadata.json` file has the following form:

```
{
  "subject": "mail subject",
  "from": "mail from address",
  "to": ["mail to address"],
  "cc": ["mail CC address"],
  "attachments": [
    "name of first attachment file",
    "name of second attachment file",
    "…"
  ]
}
```

8. ANNEX 3 Special properties

Some properties can be used to attach predefined semantics or behavior to keeexed files.

These statements usually involve using a `prop` statement with keys prefixed with `kx.`.

8.1. Prop `kx.author`

Indicates the author of a document. This historically used a IDR as a reference to the author, based on KeeeX Collaborative tools.

It is set by setting the `author` property on the `mdata` property when keeexing.

8.2. Prop `kx.description`

An informative description of the file.

It is set by setting the `description` property on the `mdata` property when keeexing.

8.3. Prop `kx.time`

The system date when the file is keeexed. This is automatically set unless the `noMetadataTime` property was set at keeexing time.

8.4. Prop `kx.revert`

Indicate how to "rebuild" the original file from the keeexed file. This props contain a JSON string as described in the `rebuild.md` file.

8.5. Prop `kx.identityURL`

Declares an identity URL used to resolve this file's signature's addresses. The value is expected to be a string denoting a web service url.

A sharding hierarchy is used for disk access efficiency on the server in the case very large numbers of signing addresses are used (for instance as a result of using deterministic address generation from a master key in the spirit of BIP32). For example, the certificate for the address `1234ABCD5678EFGH` would be found at the following path on the provided service:

```
/.well-known/keeex/identity/certificate/1234/ABCD/5678EFGH.json
```

By using this feature, the keeexer of a file declares a trusted third party able to assert if a given signing address is legitimate. This prevents using a centralized source of trust and allows for seamless automation of verifiers. This also allows the service to deliver personalized identity details.

8.6. Prop `kx.afterbc`

Holds a string indicating a recent public blockchain block. The string format is dependent upon the kind of blockchain, but clearly present the name of the blockchain as well as block and date informations. It is built using the `@keeex/afterbc` library.

8.7. Prop `kx.blockchain`

Indicates a property that references a blockchain. The purpose of such a reference is to find extra, dynamic information stored in a trusted registry or on a blockchain. This allows for unforgeable documents to track changes that are globally available.

The value is expected to have at least the following fields, in a JSON-encoded string :

```
{
  blockchain: "keeexEthereum",
  modality: "ownership",
  reference: {
    type: "smartcontract",
    address: enefteAddress,
    interface: "ERC721",
  },
}
```

Modalities can denote ownership, status, latest version etc...

8.8. Family of Properties `kx.geolocation...`

A property with geolocation information. The family features:

- `kx.geolocation.timestamp` : UNIX timestamp of the GPS data (optional)
- `kx.geolocation.status` : status of the geoloc:
- - "disabled" if the device have geolocation feature disabled
  - "mock" if a mockup location is detected
  - "ok" if everything is nominal
- `kx.geolocation.latitude` : latitude, positive means north (optional)
- `kx.geolocation.longitude` : longitude, positive means east (optional)
- `kx.geolocation.accuracy` : accuracy of the position, in meters (optional)
- `kx.geolocation.altitude` : altitude in meters (optional)
- `kx.geolocation.heading` : compass orientation (optional)
- `kx.geolocation.label` : label to associate to the geolocated point (optional).

The value of the props is combined by the verifier as a JSON object containing the following properties:

- `timestamp` : ISO8601 string of the geolocation information
- `status` : a string indicating the geolocation capture status. The only expected value, if present, is "OK", but implementation might provide custom strings. This is informative only.
- `latitude` : value (in degrees) for the latitude; positive value goes north, negative goes south
- `longitude` : value (in degrees) for the longitude; positive goes east, negative goes west
- `accuracy` : in meters

Multiple geolocation informations can be provided for a file, each with different identifiers set in the `kx.name` extra. This is useful for instance when a pdf file contains several images.

Note indeed that geolocation information may be encountered in unusual (non media) file formats.

8.9. Prop `kx.exclude` (subversion 1)

When present, this props bumps the subversion to at least "2.1" (statement version 2, subversion 1).

The `value2` is a string consisting of two numbers designating the start offset and the length of an area of the file that will not be included in the IDX computation. These numbers are in hexadecimal, and can be padded with either 0 or spaces. They are separated by a dash (`-`).

They are followed by a reason for the exclusion (also separated by a dash). This should be a short string that justify why this area is excluded.

```
keeex prop "kx.exclude", " 12bc-4-CRC section" xeeek
```

The above statement would skip the area 0x12bc to 0x12c0 (four bytes) of the file with the reason "CRC section". Multiple exclusion zones can be provided, although care must be taken to only exclude the minimum area required, and with valid justification, to not allow unauthorized modifications of the file. A prime example of exclusion zone would be other digital signature or integrity mechanisms from the underlying file format, such as CRC checksums.

8.10. Prop `kx.format`

File format injection method. Some late file alteration can happen, when updating a signature of changing the content of a reserved statement. In those cases, the appropriate post-processing function must be called if needed. If `kx.format` is present, it is an indication for the tool doing the change to use the appropriate post-processing function in those cases.

8.11. Prop `kx.copyright`

Bind a copyright notice to data. This can be present multiple times when needed. There are two forms for the value of the copyright notice:

- a plain string, that will be reproduced as-is
- a JSON object with the properties of a copyright notice

In the case of a JSON object, the following interface is considered:

```
interface YearRange {
  firstYear: number;
  lastYear: number;
}

interface CopyrightNoticePropValue {
  type: "notice";
  /** Optional note added at the end of the copyright notice
  note?: string;
  owner: string;
  /** Indicate that the content is a sound recording */
  phonorecord?: boolean;
  publishDate: number | string | YearRange;
}

enum CCLicense {
  attribution = "CC BY",
  attributionShareAlike = "CC BY-SA",
  attributionNonCommercial = "CC BY-NC",
  attributionNonCommercialSA = "CC BY-NC-SA",
  attributionNoDerivatives = "CC BY-ND",
  attributionNonCommNoDerivatives = "CC BY-NC-ND",
  publicDomain = "CC0",
}

interface CopyrightCCPropValue {
  type: "creativecommons";
  license: CCLicense;
  workName?: string;
  owner?: string;
  profileUrl?: string;
  workUrl?: string;
  publishDate?: number | string | YearRange;
}

type CopyrightPropValue =
| CopyrightNoticePropValue
| CopyrightCCPropValue;
```

A typical human representation of such statement would be:

```
© 2020-2025 Wile E. Coyote
```

Specification of what to include are taken from the following sources:

- Copyright Alliance
- epiphany.law
- photocopyrightlaw.com
- Creative Commons license picker
- Creative Commons wikipedia page

8.12. Prop `kx.classification`

This property defines the data's access level. This can be from a range of pre-defined classification levels, or a plain custom string. There can be multiple entries, and each entry can indicate what they relate to.

Note that this property can be found in two ways: hardcoded in the file, or as a dynamic property. In the case of a dynamic property, the initial value may also be present in the file, but the remote ledger is considered to have the current value if it can be reached.

Each entry should follow this interface:

```
enum BasicClassificationLevel {
  notClassified = "NC (not classified)",
  public = "C0 (public)",
  internal = "C1 (internal)",
  confidential = "C2 (confidential)",
  restricted = "C3 (restricted or secret)",
  topSecret = "C4 (top secret)",
}

interface BasicClassificationPropValue {
  type: "basic";
  level: BasicClassificationLevel;
  /** Free-form string to indicate which part of the data is classified */
  range?: string;
  /** Description of expected recipients */
  recipients?: string;
}

enum TlpClassificationLevel {
  red = "TLP:RED",
  amber = "TLP:AMBER",
  amberStrict = "TLP:AMBER+STRICT",
  green = "TLP:GREEN",
  clear = "TLP:CLEAR",
}

interface TlpClassificationPropValue {
  type: "tlp";
  level: TlpClassificationLevel;
  /** Free-form string to indicate which part of the data is classified */
  range?: string;
  /** Description of expected recipients */
  recipients?: string;
}

type ClassificationPropValue =
| BasicClassificationPropValue
| TlpClassificationPropValue;
```

Source for the classification levels:

- TLP
- Article random

8.13. Prop `kx.usageRestrictions`

Determine some restrictions/permissions on the data. This property will likely evolve over time to accommodate more options.

The basic form follows this interface:

```
enum RestrictionValue {
  allowed = "allowed",
  notAllowed = "notAllowed",
}

enum UsageFeature {
  aiTraining = "ai_training",
  aiGenerativeTraining = "ai_generative_training",
  dataMining = "data_mining",
}

interface UsageRestriction {
  restriction: RestrictionValue;
  note?: string;
}

interface UsageRestrictionsPropValue {
  /** The general restriction to consider for unspecified fields */
  defaultMode?: RestrictionValue;
  note?: string;
  feature?: Record<UsageFeature, UsageRestriction>;
}
```

Reference: IPTC GenAI opt-out best practice

9. ANNEX 4 Computing a multi-hash output

Multi-hash is a combination method that involves different hash algorithms to produce a single hash output. A selection of hash algorithms are picked and used according to the description below. The goal is to provide some level of protection against weakness in a single algorithm for long-term applications (by weakness, we consider situations were an attacker could produce a different input that would have the same hash output).

9.1. Basic layout

Assuming three hash functions named H1, H2 and H3, each producing varying length of outputs. The input data is named d, and the output hash is H. Concatenation of buffers is

noted `|`.

The hash identifier (`hashIdent`) is comprised of the string representation of the algorithms names as defined in the "Hash identifier" section below, joined with the character `<`.

We compute H this way (note that the order of the hash algorithms is important):

```
D=hashIdent|d
h3=H3(D)
h2=H2(D|h3)
h1=H1(D|h3|h2)
H=h1=H1(hashIdent|d|H3(hashIdent|d)|H2(hashIdent|d|H3(hashIdent|d)))
```

This allows a semi-parallel computation of all hashes, where all algorithms process data input in parallel and only the final step is dependent on other hashes.

The meta algorithm itself:

```
1. Process all input data with each individual hash algorithms
2. For n=[N..2], N being the number of hash algorithms considered, the output
of Hn is appended in
   every subsequent hash Hm where m=[n-1..1]
3. The output of hash algorithm 1 is the final output
```

9.2. Hash identifier

The computation requires knowing in a deterministic way the sequence of hash algorithm to use. This sequence is defined as a string containing the concatenation of each algorithm separated by a `<` character.

The list of currently defined algorithms names is:

- `sha224`
- `sha256`
- `sha512`
- `ripemd160`
- `sha3-224`
- `sha3-256`
- `sha3-384`
- `sha3-512`
- `keccak224`
- `keccak256`

- `keccak384`
- `keccak512`

The string is made of these identifier, and interpreted as an UTF-8 string for the sake of conversion in case a future algorithm name do use wide characters.

So for example, an algorithm whose identifier is `sha3-256<sha256` would compute the following hash value from a data `d`:

```
hashIdent=`sha3-256<sha256`
D="sha3_256<sha256" | d
H2=SHA256(D)
H1=SHA3_256(D | H2)
H=H1

or else H=SHA3_256(D | SHA256(D))

or else H=SHA3_256(`sha3-256<sha256` | d | SHA256(`sha3-256<sha256` | d))
```

9.3. Quick analysis

Assuming a combination of three hash functions, the dependency cycles are as follow:

- Output of H3 depends on D
- Output of H2 depends on D and H3
- Output of H1 (H) depends on D and H2 and H3

A weakness in only one hash algorithm would change the output H, since the same D value is used in all of them. To keep the same value of H with a change in D would require that all hash algorithms have the same behavior against the change in D, which is highly unlikely assuming the chosen algorithms have different core operations.

9.4. Choice of algorithms

An ideal solution is be to use wildly different algorithms from different hash function "families" such as SHA2 and SHA3.

Performance tests show that 'sha3-256' performances in the browser (in pure JavaScript) are acceptable, so the V2 settles for a default of two algorithms, `'sha3-256<sha256'` ('sha3-256' being the final output).

Cheerful thanks from the KeeeX team for reading