

Network Working Group
Internet-Draft
Intended status: Standards Track
Expires: January 19, 2018

S. Thomas
R. Reginelli
A. Hope-Bailie
Ripple
July 18, 2017

Crypto-Conditions
draft-thomas-crypto-conditions-03

Abstract

The crypto-conditions specification defines a set of encoding formats and data structures for *conditions* and *fulfillments*. A condition uniquely identifies a logical "boolean circuit" constructed from one or more logic gates, evaluated by either validating a cryptographic signature or verifying the preimage of a hash digest. A fulfillment is a data structure encoding one or more cryptographic signatures and hash digest preimages that define the structure of the circuit and provide inputs to the logic gates allowing for the result of the circuit to be evaluated.

A fulfillment is validated by evaluating that the circuit output is TRUE but also that the provided fulfillment matches the circuit fingerprint, the condition.

Since evaluation of some of the logic gates in the circuit (those that are signatures) also take a message as input the evaluation of the entire fulfillment takes an optional input message which is passed to each logic gate as required. As such the algorithm to validate a fulfillment against a condition and a message matches that of other signature schemes and a crypto-condition can serve as a sophisticated and flexible replacement for a simple signature where the condition is used as the public key and the fulfillment as the signature.

Feedback

This specification is a part of the Interledger Protocol [1] work. Feedback related to this specification should be sent to ledger@ietf.org [2].

Status of This Memo

This Internet-Draft is submitted in full conformance with the provisions of [BCP 78](#) and [BCP 79](#).

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF). Note that other groups may also distribute working documents as Internet-Drafts. The list of current Internet-Drafts is at <http://datatracker.ietf.org/drafts/current/>.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

This Internet-Draft will expire on January 19, 2018.

Copyright Notice

Copyright (c) 2017 IETF Trust and the persons identified as the document authors. All rights reserved.

This document is subject to [BCP 78](#) and the IETF Trust's Legal Provisions Relating to IETF Documents (<http://trustee.ietf.org/license-info>) in effect on the date of publication of this document. Please review these documents carefully, as they describe your rights and restrictions with respect to this document. Code Components extracted from this document must include Simplified BSD License text as described in Section 4.e of the Trust Legal Provisions and are provided without warranty as described in the Simplified BSD License.

Table of Contents

- [1.](#) Introduction [4](#)
- [2.](#) Terminology [4](#)
- [3.](#) Types [4](#)
 - [3.1.](#) Simple and Compound Types [5](#)
 - [3.2.](#) Defining and Supporting New types [5](#)
- [4.](#) Features [6](#)
 - [4.1.](#) Multi-Algorithm [6](#)
 - [4.2.](#) Multi-Signature [6](#)
 - [4.3.](#) Multi-Level [6](#)
 - [4.4.](#) Crypto-conditions as a signature scheme [7](#)
 - [4.5.](#) Crypto-conditions as a trigger in distributed systems [8](#)
 - [4.6.](#) Smart signatures [9](#)
- [5.](#) Validation of a fulfillment [9](#)
 - [5.1.](#) Subfulfillments [10](#)
- [6.](#) Deriving the Condition [10](#)
 - [6.1.](#) Conditions as Public Keys [10](#)
- [7.](#) Format [11](#)
 - [7.1.](#) Encoding Rules [11](#)
 - [7.2.](#) Condition [11](#)

7.2.1.	Fingerprint	12
7.2.2.	Cost	13
7.2.3.	Subtypes	13
7.3.	Fulfillment	14
8.	Crypto-Condition Types	15
8.1.	PREIMAGE-SHA-256	15
8.1.1.	Cost	16
8.1.2.	ASN.1	16
8.1.3.	Condition Format	16
8.1.4.	Fulfillment Format	16
8.1.5.	Validating	16
8.1.6.	Example	16
8.2.	PREFIX-SHA-256	17
8.2.1.	Cost	17
8.2.2.	ASN.1	18
8.2.3.	Condition Format	18
8.2.4.	Fulfillment Format	18
8.2.5.	Validating	19
8.2.6.	Example	19
8.3.	THRESHOLD-SHA-256	20
8.3.1.	Cost	20
8.3.2.	ASN.1	20
8.3.3.	Condition Format	20
8.3.4.	Fulfillment Format	21
8.3.5.	Validating	21
8.3.6.	Example	21
8.4.	RSA-SHA-256	23
8.4.1.	RSA Keys	23
8.4.2.	Cost	24
8.4.3.	ASN.1	24
8.4.4.	Condition Format	24
8.4.5.	Fulfillment Format	24
8.4.6.	Validating	25
8.4.7.	Example	25
8.5.	ED25519-SHA256	26
8.5.1.	Cost	27
8.5.2.	ASN.1	27
8.5.3.	Condition Format	27
8.5.4.	Fulfillment	27
8.5.5.	Validating	27
8.5.6.	Example	28
9.	URI Encoding Rules	28
9.1.	Condition URI Format	28
9.2.	New URI Parameter Definitions	29
9.2.1.	Parameter: Fingerprint Type (fpt)	29
9.2.2.	Parameter: Cost (cost)	29
9.2.3.	Parameter: Subtypes (subtypes)	29
10.	Example Condition	29

- [11. References](#) [30](#)
- [11.1. Normative References](#) [30](#)
- [11.2. Informative References](#) [31](#)
- [Appendix A. Security Considerations](#) [32](#)
- [Appendix B. Test Values](#) [33](#)
- [Appendix C. ASN.1 Module](#) [33](#)
- [Appendix D. IANA Considerations](#) [35](#)
- [D.1. Crypto-Condition Type Registry](#) [35](#)
- Authors' Addresses [35](#)

1. Introduction

Crypto-conditions is a scheme for composing signature-like structures from one or more existing signature schemes or hash digest primitives. It defines a mechanism for these existing primitives to be combined and grouped to create complex signature arrangements but still maintain the useful properties of a simple signature, most notably, that a deterministic algorithm exists to verify the signature against a message given a public key.

Using crypto-conditions, existing primitives such as RSA and ED25519 signature schemes and SHA256 digest algorithms can be used as logic gates to construct complex boolean circuits which can then be used as a compound signature. The validation function for these compound signatures takes as input the fingerprint of the circuit, called the condition, the circuit definition and minimum required logic gates with their inputs, called the fulfillment, and a message.

The function returns a boolean indicating if the compound signature is valid or not. This property of crypto-conditions means they can be used in most scenarios as a replacement for existing signature schemes which also take as input, a public key (the condition), a signature (the fulfillment), and a message and return a boolean result.

2. Terminology

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in [[RFC2119](#)].

3. Types

Crypto-conditions are a standard format for expressing conditions and fulfillments. The format supports multiple algorithms, including different hash functions and cryptographic signing schemes. Crypto-conditions can be nested in multiple levels, with each level possibly having multiple signatures.

The different types of crypto-conditions each have different internal structures and employ different cryptographic algorithms as primitives.

3.1. Simple and Compound Types

Two categories of crypto-condition type exist. Simple crypto-conditions provide a standard encoding of common cryptographic primitives with hardcoded parameters, e.g. RSA and ED25519 signature or SHA256 hash digests. As such, simple types that use the same underlying scheme (e.g. SHA) with different parameters (e.g. 256 or 512 bits) are considered different crypto-condition types.

As an example, the types defined in this version of the specification all use the SHA-256 digest algorithm to generate the condition fingerprint. If a future version were to introduce SHA-512 as an alternative this would require that new types be defined for each existing type that must have its condition generated using SHA-512.

Compound crypto-conditions contain one or more sub-crypto-conditions. The compound crypto-condition will evaluate to TRUE or FALSE based on the output of the evaluation of the sub-crypto-conditions. In this way compound crypto-conditions are used to construct branches of a boolean circuit.

To validate a compound crypto-condition all sub-crypto-conditions are provided in the fulfillment so that the fingerprint of the compound condition can be generated. However, some of these sub-crypto-conditions may be sub-fulfillments and some may be sub-conditions, depending on the type and properties of the compound crypto-condition.

As an example, in the case of an m-of-n signature scheme, only m sub-fulfillments are needed to validate the compound signature, but the remaining n-m sub-conditions must still be provided to validate that the complete fulfillment matches the originally provided condition. This is an important feature for multi-party signing, when not all parties are ready to provide fulfillment yet all parties still desire fulfillment of the overall condition if enough counter-parties do provide fulfillment.

3.2. Defining and Supporting New types

The crypto-conditions format has been designed so that it can be expanded. For example, you can add new cryptographic signature schemes or hash functions. This is important because advances in cryptography frequently render old algorithms insecure or invent newer, more effective algorithms.

Implementations are not required to support all condition types therefore it is necessary to indicate which types an implementation must support in order to validate a fulfillment. For this reason, compound conditions are encoded with an additional field, subtypes, indicating the set of types and subtypes of all sub-crypto-conditions.

4. Features

Crypto-conditions offer many of the features required of a regular signature scheme but also others which make them useful in a variety of new use cases.

4.1. Multi-Algorithm

Each condition type uses one or more cryptographic primitives such as digest or signature algorithms. Compound types may contain sub-crypto-conditions of any type and indicate the set of underlying types in the subtypes field of the condition

To verify that a given implementation can verify a fulfillment for a given condition, implementations **MUST** ensure they are able to validate fulfillments of all types indicated in the subtypes field of a compound condition. If an implementation encounters an unknown type it **MUST** reject the condition as it will almost certainly be unable to validate the fulfillment.

4.2. Multi-Signature

Crypto-conditions can abstract away many of the details of multi-sign. When a party provides a condition, other parties can treat it opaquely and do not need to know about its internal structure. That allows parties to define arbitrary multi-signature setups without breaking compatibility. That said, it is important that implementations must inspect the types and subtypes of any crypto-conditions they encounter to ensure they do not pass on a condition they will not be able to verify at a later stage.

In many instances protocol designers can use crypto-conditions as a drop-in replacement for public key signature algorithms and add multi-signature support to their protocols without adding any additional complexity.

4.3. Multi-Level

Crypto-conditions elegantly support weighted multi-signatures and multi-level signatures. A threshold condition has a number of subconditions, and a target threshold. Each subcondition can be a

signature or another threshold condition. This provides flexibility in forming complex conditions.

For example, consider a threshold condition that consists of two subconditions, one each from Wayne and Alf. Alf's condition can be a signature condition while Wayne's condition is a threshold condition, requiring both Claude and Dan to sign for him.

Multi-level signatures allow more complex relationships than simple M-of-N signing. For example, a weighted condition can support an arrangement of subconditions such as, "Either Ron, Mac, and Ped must approve; or Smithers must approve."

4.4. Crypto-conditions as a signature scheme

Crypto-conditions is a signature scheme for compound signatures which has similar properties to most other signature schemes, such as:

1. Validation of the signature (the fulfillment) is done using a public key (the condition) and a message as input
2. The same public key can be used to validate multiple different signatures, each against a different message
3. It is not possible to derive the signature from the public key

However, the scheme also has a number of features that make it unique such as:

1. It is possible to derive the same public key from any valid signature without the message
2. It is possible for the same public key and message to be used to validate multiple signatures. For example, the fulfillment of an m-of-n condition will be different for each combination of n signatures.
3. Composite signatures use one or more other signatures as components allowing for recursive signature validation logic to be defined.
4. A valid signature can be produced using different combinations of private keys if the structure of the compound signature requires only specific combinations of internal signatures to be valid (m of n signature scheme).

4.5. Crypto-conditions as a trigger in distributed systems

One of the challenges facing a distributed system is achieving atomic execution of a transaction across the system. A common pattern for solving this problem is two-phase commit in which the most time and resource-consuming aspects of the transaction are prepared by all participants following which a simple trigger is sufficient to either commit or abort the transaction. Described in more abstract terms, the system consists of a number of participants that have prepared a transaction pending the fulfillment of a predefined condition.

Crypto-conditions defines a mechanism for expressing these triggers as pairs of unique trigger identifiers (conditions) and cryptographically verifiable triggers (fulfillments) that can be deterministically verified by all participants.

It is also important that all participants in such a distributed system are able to evaluate, prior to the trigger being fired, that they will be capable of verifying the trigger. Determinism is useless if validation of the trigger requires algorithms or resources that are not available to all participants.

Therefore conditions may be used as **distributable event descriptions** in the form of a `_fingerprint_`, but also `_event meta-data_` that allows the event verification system to determine if they have the necessary capabilities (such as required crypto-algorithms) and resources (such as heap size or memory) to verify the event notification later.

Fulfillments are therefore **cryptographically verifiable event notifications** that can be used to verify the event occurred but also that it matches the given description.

When using crypto-conditions as a trigger it will often make sense for the message that is used for validation to be empty to match the signature of the trigger processing system's API. This makes crypto-conditions compatible with systems that use simple hash-locks as triggers.

If a PKI signature scheme is being used for the triggers this would require a new key pair for each trigger which is impractical. Therefore the PREFIX compound type wraps a sub-crypto-condition with a message prefix that is applied to the message before signature validation. In this way a unique condition can be derived for each trigger even if the same key pair is re-used with an empty message.

4.6. Smart signatures

In the Interledger protocol, fulfillments provide non-repudiable proof that a transaction has been completed on a ledger. They are simple messages that can be easily shared with other ledgers. This allows ledgers to escrow funds or hold a transfer conditionally, then execute the transfer automatically when the ledger sees the fulfillment of the stated condition. In this way the Interledger protocol synchronizes multiple transfers on distinct ledgers in an almost atomic end-to-end transaction.

Crypto-conditions may also be useful in other contexts where a system needs to make a decision based on predefined criteria, and the proof from a trusted oracle(s) that the criteria have been met, such as smart contracts.

The advantage of using crypto-conditions for such use cases as opposed to a turing complete contract scripting language is the fact that the outcome of a crypto-condition validation is deterministic across platforms as long as the underlying cryptographic primitives are correctly implemented.

5. Validation of a fulfillment

Validation of a fulfillment (F) against a condition (C) and a message (M), in the majority of cases, follows these steps:

1. The implementation must derive a condition from the fulfillment and ensure that the derived condition (D) matches the given condition (C).
2. If the fulfillment is a simple crypto-condition AND is based upon a signature scheme (such as RSA-PSS or ED25519) then any signatures in the fulfillment (F) must be verified, using the appropriate signature verification algorithm, against the corresponding public key, also provided in the fulfillment and the message (M) (which may be empty).
3. If the fulfillment is a compound crypto-condition then the sub-fulfillments MUST each be validated. In the case of the PREFIX-SHA-256 type the sub-fulfillment MUST be valid for F to be valid and in the case of the THRESHOLD-SHA-256 type the number of valid sub-fulfillments must be equal or greater than the threshold defined in F.

If the derived condition (D) matches the input condition (C) AND the boolean circuit defined by the fulfillment evaluates to TRUE then the fulfillment (F) fulfills the condition (C).

A more detailed validation algorithm for each crypto-condition type is provided with the details of the type later in this document. In each case the notation F.x or C.y implies; the decoded value of the field named x of the fulfillment and the decoded value of the field named y of the Condition respectively.

5.1. Subfulfillments

In validating a fulfillment for a compound crypto-condition it is necessary to validate one or more sub-fulfillments per step 3 above. In this instance the condition for one or more of these sub-fulfillments is often not available for comparison with the derived condition. Implementations MUST skip the first fulfillment validation step as defined above and only perform steps 2 and 3 of the validation.

The message (M) used to validate sub-fulfillments is the same message (M) used to validate F however in the case of the PREFIX-SHA-256 type this is prefixed with F.prefix before validation of the sub-fulfillment is performed.

6. Deriving the Condition

Since conditions provide a unique fingerprint for fulfillments it is important that a deterministic algorithm is used to derive a condition. For each crypto-condition type details are provided on how to:

1. Assemble the fingerprint content and calculate the hash digest of this data.
2. Calculate the maximum cost of validating a fulfillment

For compound types the fingerprint content will contain the complete, encoded, condition for all sub-crypto-conditions. Implementations MUST abide by the ordering rules provided when assembling the fingerprint content.

When calculating the fingerprint of a compound crypto-condition implementations MUST first derive the condition for all sub-fulfillments and include these conditions when assembling the fingerprint content.

6.1. Conditions as Public Keys

Since the condition is just a fingerprint and meta-data about the crypto-condition it can be transmitted freely in the same way a

public key is shared publicly. It's not possible to derive the fulfillment from the condition.

7. Format

A description of crypto-conditions is provided in this document using Abstract Syntax Notation One (ASN.1) as defined in [[itu.X680.2015](#)].

7.1. Encoding Rules

Implementations of this specification MUST support encoding and decoding using Distinguished Encoding Rules (DER) as defined in [[itu.X690.2015](#)]. This is the canonical encoding format.

Alternative encodings may be used to represent top-level conditions and fulfillments but to ensure a deterministic outcome in producing the condition fingerprint content, including any sub-conditions, MUST be DER encoded prior to hashing.

The exception is the PREIMAGE-SHA-256 condition where the fingerprint content is the raw preimage which is not encoded prior to hashing. This is to allow a PREIMAGE-SHA-256 crypto-condition to be used in systems where "hash-locks" are already in use.

7.2. Condition

The binary encoding of conditions differs based on their type. All types define at least a fingerprint and cost sub-field. Some types, such as the compound condition types, define additional sub-fields that are required to convey essential properties of the crypto-condition (such as the sub-types used by sub-conditions in the case of the compound types).

Each crypto-condition type has a type ID. The list of known types is the IANA-maintained Crypto-Condition Type Registry (Appendix D.1).

Conditions are encoded as follows:


```
Condition ::= CHOICE {
  preimageSha256   [0] SimpleSha256Condition,
  prefixSha256    [1] CompoundSha256Condition,
  thresholdSha256 [2] CompoundSha256Condition,
  rsaSha256       [3] SimpleSha256Condition,
  ed25519Sha256  [4] SimpleSha256Condition
}

SimpleSha256Condition ::= SEQUENCE {
  fingerprint      OCTET STRING (SIZE(32)),
  cost             INTEGER (0..4294967295)
}

CompoundSha256Condition ::= SEQUENCE {
  fingerprint      OCTET STRING (SIZE(32)),
  cost             INTEGER (0..4294967295),
  subtypes         ConditionTypes
}

ConditionTypes ::= BIT STRING {
  preImageSha256   (0),
  prefixSha256    (1),
  thresholdSha256 (2),
  rsaSha256       (3),
  ed25519Sha256  (4)
}
```

[7.2.1.](#) **Fingerprint**

The fingerprint is an octet string uniquely representing the condition with respect to other conditions *of the same type*.

Implementations which index conditions MUST use the complete encoded condition as the key, not just the fingerprint - as different conditions of different types may have the same fingerprint.

For most condition types, the fingerprint is a cryptographically secure hash of the data which defines the condition, such as a public key.

For types that use PKI signature schemes, the signature is intentionally not included in the content that is used to compose the fingerprint. This means the fingerprint can be calculated without needing to know the message or having access to the private key.

Future types may use different functions to produce the fingerprint, which may have different lengths, therefore the field is encoded as a variable length string.

7.2.2. Cost

For each type, a cost function is defined which produces a deterministic cost value based on the properties of the condition.

The cost functions are designed to produce a number that will increase rapidly if the structure and properties of a crypto-condition are such that they increase the resource requirements of a system that must validate the fulfillment.

The constants used in the cost functions are selected in order to provide some consistency across types for the cost value and the expected "real cost" of validation. This is not an exact science given that some validations will require signature verification (such as RSA and ED25519) and others will simply require hashing and storage of large values therefore the cost functions are roughly configured (through selection of constants) to be the number of bytes that would need to be processed by the SHA-256 hash digest algorithm to produce the equivalent amount of work.

The goal is to produce an indicative number that implementations can use to protect themselves from attacks involving crypto-conditions that would require massive resources to validate (denial of service type attacks).

Since dynamic heuristic measures can't be used to achieve this a deterministic value is required that can be produced consistently by any implementation, therefore for each crypto-condition type, an algorithm is provided for consistently calculating the cost.

Implementations **MUST** determine a safe cost ceiling based on the expected cost value of crypto-conditions they will need to process. When a crypto-condition is submitted to an implementation, the implementation **MUST** verify that it will be able to process a fulfillment with the given cost (i.e. the cost is lower than the allowed ceiling) and reject it if not.

Cost function constants have been rounded to numbers that have an efficient base-2 representation to facilitate efficient arithmetic operations.

7.2.3. Subtypes

Subtypes is a bitmap that indicates the set of types an implementation must support in order to be able to successfully validate the fulfillment of this condition. This is the set of types and subtypes of all sub-crypto-conditions, recursively excluding the type of the root crypto-condition.

It must be possible to verify that all types used in a crypto-condition are supported (including the types and subtypes of any sub-crypto-conditions) even if the fulfillment is not available to be analysed yet. Therefore, all compound conditions set the bits in this bitmap that correspond to the set of types and subtypes of all sub-crypto-conditions.

The field is encoded as a variable length BIT STRING, as defined in ASN.1, to accommodate new types that may be defined.

Each bit in the bitmap represents a type from the list of known types in the IANA-maintained Crypto-Condition Type Registry (Appendix D.1) and the bit corresponding to each type is the bit at position X where X is the type ID of the type.

The presence of one or more sub-crypto-conditions of a specific type is indicated by setting the numbered bit corresponding to the type ID of that type.

In DER encoding, the bits in a bitstring are numbered from the MOST significant bit (bit 0) to least significant (bit 7) of the first byte and then continue with the MOST significant bit (bit 8) of the next byte, and so on. For example, a compound condition that contains an ED25519-SHA-256 crypto-condition as a sub-crypto-condition will set the bit at position 4 and the BITSTRING will be DER encoded with an appropriate tag byte followed by the three bytes 0x02 0x03 and 0x80, where 0x02 indicates the length (2 bytes, the first being the padding indicator), 0x03 indicates that there are 3 padding bits in the last byte and 0x80 indicates the 5 bits in the string are set to 00001.

7.3. Fulfillment

The ASN.1 definition for fulfillments is defined as follows:


```
Fulfillment ::= CHOICE {
  preimageSha256    [0] PreimageFulfillment ,
  prefixSha256     [1] PrefixFulfillment,
  thresholdSha256  [2] ThresholdFulfillment,
  rsaSha256        [3] RsaSha256Fulfillment,
  ed25519Sha256   [4] Ed25519Sha512Fulfillment
}
```

```
PreimageFulfillment ::= SEQUENCE {
  preimage          OCTET STRING
}
```

```
PrefixFulfillment ::= SEQUENCE {
  prefix            OCTET STRING,
  maxLength         INTEGER (0..4294967295),
  subfulfillment   Fulfillment
}
```

```
ThresholdFulfillment ::= SEQUENCE {
  subfulfillments  SET OF Fulfillment,
  subconditions    SET OF Condition
}
```

```
RsaSha256Fulfillment ::= SEQUENCE {
  modulus          OCTET STRING,
  signature        OCTET STRING
}
```

```
Ed25519Sha512Fulfillment ::= SEQUENCE {
  publicKey        OCTET STRING (SIZE(32)),
  signature        OCTET STRING (SIZE(64))
}
```

8. Crypto-Condition Types

The following condition types are defined in this version of the specification. While support for additional crypto-condition types may be added in the future and will be registered in the IANA maintained Crypto-Condition Type Registry (Appendix D.1), no other types are supported by this specification.

8.1. PREIMAGE-SHA-256

PREIMAGE-SHA-256 is assigned the type ID 0. It relies on the availability of the SHA-256 digest algorithm.

This type of condition is also called a "hashlock". By creating a hash of a difficult-to-guess 256-bit random or pseudo-random integer

it is possible to create a condition which the creator can trivially fulfill by publishing the random value. However, for anyone else, the condition is cryptographically hard to fulfill, because they would have to find a preimage for the given condition hash.

Implementations MUST ignore any input message when validating a PREIMAGE-SHA-256 fulfillment as the validation of this crypto-condition type only requires that the SHA-256 digest of the preimage, taken from the fulfillment, matches the fingerprint, taken from the condition.

8.1.1. Cost

The cost is the size, in bytes, of the *unencoded* preimage.

cost = preimage length

8.1.2. ASN.1

```
-- Condition Fingerprint
-- The PREIMAGE-SHA-256 condition fingerprint content is not DER encoded
-- The fingerprint content is the preimage

-- Fulfillment
PreimageFulfillment ::= SEQUENCE {
  preimage          OCTET STRING
}
```

8.1.3. Condition Format

The fingerprint of a PREIMAGE-SHA-256 condition is the SHA-256 hash of the *unencoded* preimage.

8.1.4. Fulfillment Format

The fulfillment simply contains the preimage (encoded into a SEQUENCE of one element for consistency).

8.1.5. Validating

A PREIMAGE-SHA-256 fulfillment is valid iff C.fingerprint is equal to the SHA-256 hash digest of F.

8.1.6. Example


```
examplePreimageCondition Condition ::=
  preimageSha256 : {
    fingerprint '7F83B165 7FF1FC53 B92DC181 48A1D65D FC2D4B1F A3D67728 4ADDD200
126D9069'H,
    cost          12
  }
```

```
examplePreimageFulfillment Fulfillment ::=
  preimageSha256 : {
    preimage '48656C6C 6F20576F 726C6421'H
  }
```

8.2. PREFIX-SHA-256

PREFIX-SHA-256 is assigned the type ID 1. It relies on the availability of the SHA-256 digest algorithm and any other algorithms required by its sub-crypto-condition as it is a compound crypto-condition type.

Prefix crypto-conditions provide a way to narrow the scope of other crypto-conditions that are used inside the prefix crypto-condition as a sub-crypto-condition.

Because a condition is the fingerprint of a public key, by creating a prefix crypto-condition that wraps another crypto-condition we can narrow the scope from signing an arbitrary message to signing a message with a specific prefix.

We can also use the prefix condition in contexts where there is an empty message used for validation of the fulfillment so that we can reuse the same key pair for multiple crypto-conditions, each with a different prefix, and therefore generate a unique condition and fulfillment each time.

Implementations MUST prepend the prefix to the provided message and will use the resulting value as the message to validate the sub-fulfillment.

8.2.1. Cost

The cost is the size, in bytes, of the *unencoded* prefix, plus the maximum message that will be accepted to be prefixed and validated by the subcondition, plus the cost of the sub-condition, plus the constant 1024.

```
cost = prefix.length (in bytes) + max_message_length + subcondition_cost + 1024
```


8.2.2. ASN.1

```
-- Condition Fingerprint
PrefixFingerprintContents ::= SEQUENCE {
    prefix          OCTET STRING,
    maxMessageLength  INTEGER (0..4294967295),
    subcondition     Condition
}

-- Fulfillment
PrefixFulfillment ::= SEQUENCE {
    prefix          OCTET STRING,
    maxMessageLength  INTEGER (0..4294967295),
    subfulfillment   Fulfillment
}
```

8.2.3. Condition Format

The fingerprint of a PREFIX-SHA-256 condition is the SHA-256 digest of the DER encoded fingerprint contents which are a SEQUENCE of:

prefix An arbitrary octet string which will be prepended to the message during validation of the sub-fulfillment.

maxMessageLength The maximum size, in bytes, of the message that will be accepted during validation of the fulfillment of this condition.

subcondition The condition derived from the sub-fulfillment of this crypto-condition.

8.2.4. Fulfillment Format

The fulfillment of a PREFIX-SHA-256 crypto-condition is a PrefixFulfillment which is a SEQUENCE of:

prefix An arbitrary octet string which will be prepended to the message during validation of the sub-fulfillment.

maxMessageLength The maximum size, in bytes, of the message that will be accepted during validation of the fulfillment of this condition.

subfulfillment A fulfillment that will be verified against the prefixed message.

8.2.5. Validating

A PREFIX-SHA-256 fulfillment is valid iff:

1. The size of M, in bytes, is less than or equal to F.maxMessageLength AND
2. F.subfulfillment is valid, where the message used for validation of f is M prefixed by F.prefix AND
3. D is equal to C

8.2.6. Example

```
examplePrefixCondition Condition ::=
  prefixSha256 : {
    fingerprint 'BB1AC526 0C0141B7 E54B26EC 2330637C 5597BF81 1951AC09 E744AD20
FF77E287'H,
    cost          1024,
    subtypes     { preimageSha256 }
  }
```

```
examplePrefixFulfillment Fulfillment ::=
  prefixSha256 : {
    prefix          'H,
    maxMessageLength 0,
    subfulfillment  preimageSha256 : { preimage 'H }
  }
```

```
examplePrefixFingerprintContents PrefixFingerprintContents ::= {
  prefix          'H,
  maxMessageLength 0,
  subcondition    preimageSha256 : {
    fingerprint
'E3B0C44298FC1C149AFBF4C8996FB92427AE41E4649B934CA495991B7852B855'H,
    cost          0
  }
}
```

Note that the example given, while useful to demonstrate the structure, has less practical security value than the use of an RSA-SHA-256 or ED25519-SHA-256 subfulfillment. Since the subfulfillment is a PREIMAGE-SHA-256, the validation of which ignores the incoming message, as long as the prefix, maxMessageLength and preimage provided in the subfulfillment are correct, the parent PREFIX-SHA-256 fulfillment will validate.

In this case, wrapping the PREIMAGE-SHA-256 crypto-condition in the PREFIX-SHA-256 crypto-condition, has the effect of enforcing a

message length of 0 bytes.

Thomas, et al.

Expires January 19, 2018

[Page 19]

Note also, any change to the PREFIX-SHA-256 crypto-condition's prefix and maxMessageLength values result in a different fingerprint value, effectively namespacing the underlying preimage and re-hashing it. The result is a new crypto-condition with a new and unique fingerprint with no change to the underlying sub-crypto-condition.

8.3. THRESHOLD-SHA-256

THRESHOLD-SHA-256 is assigned the type ID 2. It relies on the availability of the SHA-256 digest algorithm and any other algorithms required by any of its sub-crypto-conditions as it is a compound crypto-condition type.

8.3.1. Cost

The cost is the sum of the F.threshold largest cost values of all sub-conditions, added to 1024 times the total number of sub-conditions.

$$\text{cost} = (\text{sum of largest } F.\text{threshold} \text{ subcondition.cost values}) + 1024 * F.\text{subconditions.count}$$

For example, if a threshold crypto-condition contains 5 sub-conditions with costs of 64, 64, 82, 84 and 84 and has a threshold of 3, the cost is equal to the sum of the largest three sub-condition costs ($82 + 84 + 84 = 250$) plus 1024 times the number of sub-conditions ($1024 * 5 = 5120$): 5370

8.3.2. ASN.1

```
-- Condition Fingerprint
ThresholdFingerprintContents ::= SEQUENCE {
    threshold          INTEGER (1..65535),
    subconditions      SET OF Condition
}

-- Fulfillment
ThresholdFulfillment ::= SEQUENCE {
    subfulfillments    SET OF Fulfillment,
    subconditions      SET OF Condition
}
```

8.3.3. Condition Format

The fingerprint of a THRESHOLD-SHA-256 condition is the SHA-256 digest of the DER encoded fingerprint contents which are a SEQUENCE of:

threshold A number that MUST be an integer in the range 1 ... 65535. In order to fulfill a threshold condition, the count of the sub-fulfillments MUST be equal to the threshold.

subconditions The set of sub-conditions, F.threshold of which MUST be satisfied by valid sub-fulfillments provided in the fulfillment. The SET of DER encoded sub-conditions is sorted according to the DER encoding rules for a SET, in lexicographic (big-endian) order, smallest first as defined in section 11.6 of [[itu.X690.2015](#)].

8.3.4. Fulfillment Format

The fulfillment of a THRESHOLD-SHA-256 crypto-condition is a ThresholdFulfillment which is a SEQUENCE of:

subfulfillments A SET OF fulfillments. The number of elements in this set is equal to the threshold therefore implementations must use the length of this SET as the threshold value when deriving the fingerprint of this crypto-condition.

subconditions A SET OF conditions. This is the list of unfulfilled sub-conditions. This list must be combined with the list of conditions derived from the subfulfillments and the combined list, sorted, and used as the subconditions value when deriving the fingerprint of this crypto-condition.

This may be an empty list.

8.3.5. Validating

A THRESHOLD-SHA-256 fulfillment is valid iff :

1. All F.subfulfillments are valid.
2. D is equal to C.

8.3.6. Example

```
exampleThresholdCondition Condition ::=
  thresholdSha256 : {
    fingerprint 'B4B84136 DF48A71D 73F4985C 04C6767A 778ECB65 BA7023B4 506823BE
EE7631B9'H,
    cost          1024,
    subtypes     { preimageSha256 }
  }
```

```
exampleThresholdFulfillment Fulfillment ::=
  thresholdSha256 : {
```



```
    subfulfillments { preimageSha256 : { preimage 'H' } },
    subconditions   { }
  }
```

```
exampleThresholdFingerprintContents ThresholdFingerprintContents ::= {
  threshold 1,
  subconditions {
    preimageSha256 : {
      fingerprint 'E3B0C442 98FC1C14 9AFBF4C8 996FB924 27AE41E4 649B934C
A495991B 7852B855'H,
      cost        0
    }
  }
}
```

```
exampleThresholdCondition2 Condition ::=
  thresholdSha256 : {
    fingerprint '5A218ECE 7AC4BC77 157F04CB 4BC8DFCD 5C9D225A 55BD0AA7 60BCA2A4
F1773DC6'H,
    cost        2060,
    subtypes   { preimageSha256 }
  }
```

```
exampleThresholdFulfillment2 Fulfillment ::=
  thresholdSha256 : {
    subfulfillments { preimageSha256 : { preimage 'H' } },
    subconditions {
      preimageSha256 : {
        fingerprint '7F83B165 7FF1FC53 B92DC181 48A1D65D FC2D4B1F A3D67728
4ADDD200 126D9069'H,
        cost        12
      }
    }
  }
```

```
exampleThresholdFingerprintContents2 ThresholdFingerprintContents ::= {
  threshold 1,
  subconditions {
    preimageSha256 : {
      fingerprint 'E3B0C442 98FC1C14 9AFBF4C8 996FB924 27AE41E4 649B934C
A495991B 7852B855'H,
      cost        0
    },
    preimageSha256 : {
      fingerprint '7F83B165 7FF1FC53 B92DC181 48A1D65D FC2D4B1F A3D67728
4ADDD200 126D9069'H,
      cost        12
    }
  }
```

}
}

Thomas, et al.

Expires January 19, 2018

[Page 22]

8.4. RSA-SHA-256

RSA-SHA-256 is assigned the type ID 3. It relies on the SHA-256 digest algorithm and the RSA-PSS signature scheme.

The signature algorithm used is RSASSA-PSS as defined in PKCS#1 v2.2. [[RFC8017](#)]

Implementations MUST NOT use the default RSASSA-PSS-params. Implementations MUST use the SHA-256 hash algorithm and therefore, the same algorithm in the mask generation algorithm, as recommended in [[RFC8017](#)]. The algorithm parameters to use, as defined in [[RFC4055](#)] are:

```
pkcs-1 OBJECT IDENTIFIER ::= { iso(1) member-body(2) us(840) rsadsi(113549)
pkcs(1) 1 }
```

```
id-sha256 OBJECT IDENTIFIER ::= { joint-iso-itu-t(2) country(16) us(840)
organization(1) gov(101) csor(3) nistalgorithm(4) hashalgs(2) 1 }
```

```
sha256Identifier AlgorithmIdentifier ::= {
  algorithm          id-sha256,
  parameters         nullParameters
}
```

```
id-mgf1 OBJECT IDENTIFIER ::= { pkcs-1 8 }
```

```
mgf1SHA256Identifier AlgorithmIdentifier ::= {
  algorithm          id-mgf1,
  parameters         sha256Identifier
}
```

```
rSASSA-PSS-SHA256-Params RSASSA-PSS-params ::= {
  hashAlgorithm      sha256Identifier,
  maskGenAlgorithm   mgf1SHA256Identifier,
  saltLength         20,
  trailerField       1
}
```

8.4.1. RSA Keys

To optimize the RsaFulfillment, and enforce a public exponent value of 65537, only the RSA Public Key modulus is stored in the RsaFingerprintContents and RsaFulfillment.

The modulus is stored as an OCTET STRING representing an unsigned integer (i.e. no sign byte) in big-endian byte-order, the most significant byte being the first in the string.

Implementations MUST use moduli greater than 128 bytes (1017 bits) and smaller than or equal to 512 bytes (4096 bits.) Large moduli slow down signature verification which can be a denial-of-service vector. DNSSEC also limits the modulus to 4096 bits [[RFC3110](#)]. OpenSSL supports up to 16384 bits [[OPENSSL-X509-CERT-EXAMPLES](#)].

Implementations MUST use the value 65537 for the public exponent e as recommended in [[RFC4871](#)]. Very large exponents can be a DoS vector [[LARGE-RSA-EXPONENTS](#)] and 65537 is the largest Fermat prime, which has some nice properties [[USING-RSA-EXPONENT-OF-65537](#)].

The recommended modulus size as of 2016 is 2048 bits [[KEYLENGTH-RECOMMENDATION](#)]. In the future we anticipate an upgrade to 3072 bits which provides approximately 128 bits of security [[NIST-KEYMANAGEMENT](#)] (p. 64), about the same level as SHA-256.

[8.4.2.](#) Cost

The cost is the square of the RSA key modulus size (in bytes).

cost = (modulus size in bytes) ^ 2

[8.4.3.](#) ASN.1

```
-- Condition Fingerprint
RsaFingerprintContents ::= SEQUENCE {
    modulus          OCTET STRING
}

-- Fulfillment
RsaSha256Fulfillment ::= SEQUENCE {
    modulus          OCTET STRING,
    signature        OCTET STRING
}
```

[8.4.4.](#) Condition Format

The fingerprint of an RSA-SHA-256 condition is the SHA-256 digest of the DER encoded fingerprint contents which is a SEQUENCE of a single element, the modulus of the RSA Key Pair.

[8.4.5.](#) Fulfillment Format

The fulfillment of an RSA-SHA-256 crypto-condition is an RsaSha256Fulfillment which is a SEQUENCE of:

modulus The modulus of the RSA key pair used to sign and verify the signature provided.

signature An octet string representing the RSA signature on the message M.

Implementations MUST verify that the signature is numerically less than the modulus.

Note that the message that has been signed is provided separately. If no message is provided, the message is assumed to be an octet string of length zero.

8.4.6. Validating

An RSA-SHA-256 fulfillment is valid iff :

1. F.signature is valid for the message M, using the RSA public key with modulus = F.modulus and exponent = 65537 for verification.
2. D is equal to C.

8.4.7. Example


```
exampleRsaCondition Condition ::=
```

```
  rsaSha256 : {
    fingerprint 'B31FA820 6E4EA7E5 15337B3B 33082B87 76518010 85ED84FB 4DAEB247
BF698D7F'H,
    cost          65536
  }
```

```
exampleRsaSha256Fulfillment Fulfillment ::=
```

```
  rsaSha256 : {
    modulus      'E1EF8B24 D6F76B09 C81ED775 2AA262F0 44F04A87 4D43809D 31CEA612
F99B0C97 A8B43741
                  53E3EEF3 D6661684 3E0E41C2 93264B71 B6173DB1 CF0D6CD5 58C58657
706FCF09 7F704C48
                  3E59CBFD FD5B3EE7 BC80D740 C5E0F047 F3E85FC0 D7581577 6A6F3F23
C5DC5E79 7139A688
                  2E38336A 4A5FB361 37620FF3 663DBAE3 28472801 862F72F2 F87B202B
9C89ADD7 CD5B0A07
                  6F7C53E3 5039F67E D17EC815 E5B4305C C6319706 8D5E6E57 9BA6DE5F
4E3E57DF 5E4E072F
                  F2CE4C66 EB452339 73875275 9639F025 7BF57DBD 5C443FB5 158CCE0A
3D36ADC7 BA01F33A
                  0BB6DBB2 BF989D60 7112F234 4D993E77 E563C1D3 61DEDF57 DA96EF2C
FC685F00 2B638246
                  A5B309B9'H,
    signature    '48E8945E FE007556 D5BF4D5F 249E4808 F7307E29 511D3262 DAEF61D8
8098F9AA 4A8BC062
                  3A8C9757 38F65D6B F459D543 F289D73C BC7AF4EA 3A33FBF3 EC444044
7911D722 94091E56
                  1833628E 49A772ED 608DE6C4 4595A91E 3E17D6CF 5EC3B252 8D63D2AD
D6463989 B12EEC57
                  7DF64709 60DF6832 A9D84C36 0D1C217A D64C8625 BDB594FB 0ADA086C
DECBDE5 80D424BF
                  9746D2F0 C312826D BBB00AD6 8B52C4CB 7D47156B A35E3A98 1C973863
792CC80D 04A18021
                  0A524158 65B64B3A 61774B1D 3975D78A 98B0821E E55CA0F8 6305D425
29E10EB0 15CEFD40
                  2FB59B2A BB8DEEE5 2A6F2447 D2284603 D219CD4E 8CF9CFFD D5498889
C3780B59 DD6A57EF
                  7D732620'H
  }
```

```
exampleRsaFingerprintContents RsaFingerprintContents ::= {
```

```
  modulus      'E1EF8B24 D6F76B09 C81ED775 2AA262F0 44F04A87 4D43809D 31CEA612
F99B0C97 A8B43741
                  53E3EEF3 D6661684 3E0E41C2 93264B71 B6173DB1 CF0D6CD5 58C58657
706FCF09 7F704C48
                  3E59CBFD FD5B3EE7 BC80D740 C5E0F047 F3E85FC0 D7581577 6A6F3F23
C5DC5E79 7139A688
```

```
2E38336A 4A5FB361 37620FF3 663DBAE3 28472801 862F72F2 F87B202B
9C89ADD7 CD5B0A07
6F7C53E3 5039F67E D17EC815 E5B4305C C6319706 8D5E6E57 9BA6DE5F
4E3E57DF 5E4E072F
F2CE4C66 EB452339 73875275 9639F025 7BF57DBD 5C443FB5 158CCE0A
3D36ADC7 BA01F33A
0BB6DBB2 BF989D60 7112F234 4D993E77 E563C1D3 61DEDF57 DA96EF2C
FC685F00 2B638246
A5B309B9'H
}
```

8.5. ED25519-SHA256

ED25519-SHA-256 is assigned the type ID 4. It relies on the SHA-256 and SHA-512 digest algorithms and the ED25519 signature scheme.

The exact algorithm and encodings used for the public key and signature are defined in [[I-D.irtf-cfrg-eddsa](#)] as Ed25519. SHA-512 is used as the hashing function for this signature scheme.

[8.5.1.](#) Cost

The public key and signature are a fixed size therefore the cost for an ED25519 crypto-condition is fixed at 131072.

```
cost = 131072
```

[8.5.2.](#) ASN.1

```
-- Condition Fingerprint
Ed25519Sha512Fulfillment ::= SEQUENCE {
    publicKey          OCTET STRING (SIZE(32)),
    signature          OCTET STRING (SIZE(64))
}

-- Fulfillment
Ed25519FingerprintContents ::= SEQUENCE {
    publicKey          OCTET STRING (SIZE(32))
}
```

[8.5.3.](#) Condition Format

The fingerprint of an ED25519-SHA-256 condition is the SHA-256 digest of the DER encoded Ed25519 public key included as the only value within a SEQUENCE. While the public key is already very small and constant size, we wrap it in a SEQUENCE type and hash it for consistency with the other types.

[8.5.4.](#) Fulfillment

The fulfillment of an ED25519-SHA-256 crypto-condition is an Ed25519Sha512Fulfillment which is a SEQUENCE of:

publicKey An octet string containing the Ed25519 public key.

signature An octet string containing the Ed25519 signature.

[8.5.5.](#) Validating

An ED25519-SHA-256 fulfillment is valid iff :

1. F.signature is valid for the message M, given the ED25519 public key F.publicKey.
2. D is equal to C.

8.5.6. Example

```
exampleEd25519Condition Condition ::=
  ed25519Sha256 : {
    fingerprint '799239AB A8FC4FF7 EABFBC4C 44E69E8B DFED9933 24E12ED6 4792ABE2
89CF1D5F'H,
    cost 131072
  }
```

```
exampleEd25519Fulfillment Fulfillment ::=
  ed25519Sha256 : {
    publicKey 'D75A9801 82B10AB7 D54BFED3 C964073A 0EE172F3 DAA62325 AF021A68
F707511A'H,
    signature 'E5564300 C360AC72 9086E2CC 806E828A 84877F1E B8E5D974 D873E065
22490155
                    5FB88215 90A33BAC C61E3970 1CF9B46B D25BF5F0 595BBE24 65514143
8E7A100B'H
  }
```

```
exampleEd25519FingerprintContents Ed25519FingerprintContents ::= {
  publicKey 'D75A9801 82B10AB7 D54BFED3 C964073A 0EE172F3 DAA62325 AF021A68
F707511A'H
}
```

9. URI Encoding Rules

Conditions can be encoded as URIs per the rules defined in the Named Information specification, [\[RFC6920\]](#). There are no URI encoding rules for fulfillments.

Applications that require a string encoding for fulfillments MUST use an appropriate string encoding of the DER encoded binary representation of the fulfillment. No string encoding is defined in this specification. For consistency with the URI encoding of conditions, BASE64URL is recommended as described in [\[RFC4648\]](#), [Section 5](#).

The URI encoding is only used to encode top-level conditions and never for sub-conditions. The binary encoding is considered the canonical encoding.

9.1. Condition URI Format

Conditions are represented as URIs using the rules defined in [\[RFC6920\]](#) where the object being hashed is the DER encoded fingerprint content of the condition as described for the specific condition type.

While [\[RFC6920\]](#) allows for truncated hashes, implementations using

the Named Information URI schemes for crypto-conditions MUST only use untruncated SHA-256 hashes (Hash Name: sha-256, ID: 1 from the "Named Information Hash Algorithm Registry" defined in [[RFC6920](#)]).

9.2. New URI Parameter Definitions

[RFC6920] established the IANA registry of "Named Information URI Parameter Definitions". This specification defines three new definitions that are added to that registry and passed in URI encoded conditions as query string parameters.

9.2.1. Parameter: Fingerprint Type (fpt)

The type parameter indicates the type of condition that is represented by the URI. The value MUST be one of the names from the Crypto-Condition Type Registry (Appendix D.1).

9.2.2. Parameter: Cost (cost)

The cost parameter is the cost of the condition that is represented by the URI.

9.2.3. Parameter: Subtypes (subtypes)

The subtypes parameter indicates the types of conditions that are subtypes of the condition represented by the URI. The value MUST be a comma separated list of names from the Crypto-Condition Type Registry (Appendix D.1).

The list MUST exclude the type of the root crypto-condition. i.e. The value of the fpt parameter should not appear in the list of types provided as the value of the subtypes parameter.

The list MUST be ordered by the type id value of each type, in ascending order. i.e. preimage-sha-256 MUST appear before prefix-sha-256.

The commas in the list should be treated as reserved characters per [RFC3986] and not be percent encoded when used as list delimiters in the subtypes parameter.

10. Example Condition

An example condition (PREIMAGE-SHA-256):

```
0x00000000 A0 25 80 20 7F 83 B1 65 7F F1 FC 53 B9 2D C1 81 .%. ....e...S.-...
0x00000010 48 A1 D6 5D FC 2D 4B 1F A3 D6 77 28 4A DD D2 00 H..].-K...w(J...
0x00000020 12 6D 90 69 81 01 0C                               .m.i...
```

```
ni:///sha-256;f40xZX_x_F05LcGBSKHWXfwtSx-j1ncoSt3SABJtkGk?fpt=preimage-
sha-256&cost=12
```

The example has the following attributes:

Field	Value	Description
scheme	"ni:///"	The named information scheme.
hash function name	"sha-256"	The fingerprint is hashed with the SHA-256 digest function
fingerprint	"f40xZX_x_F05LcGBSKHWXfwtSx-j1ncoSt3SABJtkGk"	The fingerprint for this condition.
type	"preimage-sha-256"	This is a PREIMAGE-SHA-256 (Section 8.1) condition.
cost	"12"	The fulfillment payload is 12 bytes long, therefore the cost is 12.

11. References

11.1. Normative References

[I-D.irtf-cfrg-eddsa]

Josefsson, S. and I. Liusvaara, "Edwards-curve Digital Signature Algorithm (EdDSA)", [draft-irtf-cfrg-eddsa-08](#) (work in progress), August 2016.

[itu.X680.2015]

International Telecommunications Union, "Information technology - Abstract Syntax Notation One (ASN.1): Specification of basic notation", August 2015, <http://handle.itu.int/11.1002/1000/12479>.

- [itu.X690.2015]
International Telecommunications Union, "Information technology - ASN.1 encoding rules: Specification of Basic Encoding Rules (BER), Canonical Encoding Rules (CER) and Distinguished Encoding Rules (DER)", August 2015, <<http://handle.itu.int/11.1002/1000/12483>>.
- [RFC3280] Housley, R., Polk, W., Ford, W., and D. Solo, "Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile", [RFC 3280](#), DOI 10.17487/RFC3280, April 2002, <<http://www.rfc-editor.org/info/rfc3280>>.
- [RFC3986] Berners-Lee, T., Fielding, R., and L. Masinter, "Uniform Resource Identifier (URI): Generic Syntax", STD 66, [RFC 3986](#), DOI 10.17487/RFC3986, January 2005, <<http://www.rfc-editor.org/info/rfc3986>>.
- [RFC4055] Schaad, J., Kaliski, B., and R. Housley, "Additional Algorithms and Identifiers for RSA Cryptography for use in the Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile", [RFC 4055](#), DOI 10.17487/RFC4055, June 2005, <<http://www.rfc-editor.org/info/rfc4055>>.
- [RFC4648] Josefsson, S., "The Base16, Base32, and Base64 Data Encodings", [RFC 4648](#), DOI 10.17487/RFC4648, October 2006, <<http://www.rfc-editor.org/info/rfc4648>>.
- [RFC6920] Farrell, S., Kutscher, D., Dannewitz, C., Ohlman, B., Keranen, A., and P. Hallam-Baker, "Naming Things with Hashes", [RFC 6920](#), DOI 10.17487/RFC6920, April 2013, <<http://www.rfc-editor.org/info/rfc6920>>.
- [RFC8017] Moriarty, K., Ed., Kaliski, B., Jonsson, J., and A. Rusch, "PKCS #1: RSA Cryptography Specifications Version 2.2", [RFC 8017](#), DOI 10.17487/RFC8017, November 2016, <<http://www.rfc-editor.org/info/rfc8017>>.

11.2. Informative References

- [KEYLENGTH-RECOMMENDATION]
"BlueKrypt - Cryptographic Key Length Recommendation", September 2015, <<https://www.keylength.com/en/compare/>>.

[LARGE-RSA-EXPONENTS]

"Imperial Violet - Very large RSA public exponents (17 Mar 2012)", March 2012,
<<https://www.imperialviolet.org/2012/03/17/rsados.html>>.

[NIST-KEYMANAGEMENT]

and , "NIST - Recommendation for Key Management - Part 1 - General (Revision 3)", July 2012,
<http://csrc.nist.gov/publications/nistpubs/800-57/sp800-57_part1_rev3_general.pdf>.

[OPENSSL-X509-CERT-EXAMPLES]

"OpenSSL - X509 certificate examples for testing and verification", July 2012,
<<http://fm4dd.com/openssl/certexamples.htm>>.

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", [BCP 14](#), [RFC 2119](#), DOI 10.17487/RFC2119, March 1997,
<<http://www.rfc-editor.org/info/rfc2119>>.

[RFC3110] Eastlake 3rd, D., "RSA/SHA-1 SIGs and RSA KEYS in the Domain Name System (DNS)", [RFC 3110](#), DOI 10.17487/RFC3110, May 2001, <<http://www.rfc-editor.org/info/rfc3110>>.

[RFC4871] Allman, E., Callas, J., Delany, M., Libbey, M., Fenton, J., and M. Thomas, "DomainKeys Identified Mail (DKIM) Signatures", [RFC 4871](#), DOI 10.17487/RFC4871, May 2007,
<<http://www.rfc-editor.org/info/rfc4871>>.

[USING-RSA-EXPONENT-OF-65537]

"Cryptography - StackExchange - Impacts of not using RSA exponent of 65537", November 2014,
<<https://crypto.stackexchange.com/questions/3110/impacts-of-not-using-rsa-exponent-of-65537>>.

Appendix A. Security Considerations

This specification has a normative dependency on a number of other specifications with extensive security considerations therefore the considerations defined for SHA-256 hashing and RSA signatures in [[RFC8017](#)] and [[RFC4055](#)] and for ED25519 signatures in [[I-D.irtf-cfrg-eddsa](#)] must be considered.

The cost and subtypes values of conditions are provided to allow implementations to evaluate their ability to validate a fulfillment for the given condition later.

[Appendix B](#). Test Values

This section to be expanded in a later draft.

For now, see the test cases for the reference implementation:

<https://github.com/interledger/five-bells-condition/tree/master/test>

[Appendix C](#). ASN.1 Module

```
--<ASN1.PDU CryptoConditions.Condition, CryptoConditions.Fulfillment>--

Crypto-Conditions DEFINITIONS AUTOMATIC TAGS ::= BEGIN

-- Conditions

Condition ::= CHOICE {
  preimageSha256  [0] SimpleSha256Condition,
  prefixSha256   [1] CompoundSha256Condition,
  thresholdSha256 [2] CompoundSha256Condition,
  rsaSha256      [3] SimpleSha256Condition,
  ed25519Sha256 [4] SimpleSha256Condition
}

SimpleSha256Condition ::= SEQUENCE {
  fingerprint  OCTET STRING (SIZE(32)),
  cost         INTEGER (0..4294967295)
}

CompoundSha256Condition ::= SEQUENCE {
  fingerprint  OCTET STRING (SIZE(32)),
  cost         INTEGER (0..4294967295),
  subtypes     ConditionTypes
}

ConditionTypes ::= BIT STRING {
  preImageSha256  (0),
  prefixSha256   (1),
  thresholdSha256 (2),
  rsaSha256      (3),
  ed25519Sha256  (4)
}

-- Fulfillments

Fulfillment ::= CHOICE {
  preimageSha256  [0] PreimageFulfillment ,
  prefixSha256   [1] PrefixFulfillment,
  thresholdSha256 [2] ThresholdFulfillment,
```



```
    rsaSha256          [3] RsaSha256Fulfillment,  
    ed25519Sha256     [4] Ed25519Sha512Fulfillment  
}
```

```
PreimageFulfillment ::= SEQUENCE {  
    preimage          OCTET STRING  
}
```

```
PrefixFulfillment ::= SEQUENCE {  
    prefix            OCTET STRING,  
    maxMessageLength INTEGER (0..4294967295),  
    subfulfillment   Fulfillment  
}
```

```
ThresholdFulfillment ::= SEQUENCE {  
    subfulfillments  SET OF Fulfillment,  
    subconditions    SET OF Condition  
}
```

```
RsaSha256Fulfillment ::= SEQUENCE {  
    modulus           OCTET STRING,  
    signature         OCTET STRING  
}
```

```
Ed25519Sha512Fulfillment ::= SEQUENCE {  
    publicKey         OCTET STRING (SIZE(32)),  
    signature         OCTET STRING (SIZE(64))  
}
```

-- Fingerprint Content

-- The PREIMAGE-SHA-256 condition fingerprint content is not DER encoded
-- The fingerprint content is the preimage

```
PrefixFingerprintContents ::= SEQUENCE {  
    prefix            OCTET STRING,  
    maxMessageLength INTEGER (0..4294967295),  
    subcondition      Condition  
}
```

```
ThresholdFingerprintContents ::= SEQUENCE {  
    threshold         INTEGER (1..65535),  
    subconditions     SET OF Condition  
}
```

```
RsaFingerprintContents ::= SEQUENCE {  
    modulus           OCTET STRING  
}
```



```
Ed25519FingerprintContents ::= SEQUENCE {
  publicKey          OCTET STRING (SIZE(32))
}
```

END

Appendix D. IANA Considerations

D.1. Crypto-Condition Type Registry

The following initial entries should be added to the Crypto-Condition Type registry to be created and maintained at (the suggested URI) <http://www.iana.org/assignments/crypto-condition-types>:

The following types are registered:

Type ID	Type Name
0	PREIMAGE-SHA-256
1	PREFIX-SHA-256
2	THRESHOLD-SHA-256
3	RSA-SHA-256
4	ED25519

Table 1: Crypto-Condition Types

Authors' Addresses

Stefan Thomas
 Ripple
 300 Montgomery Street
 San Francisco, CA 94104
 US

Phone: -----
 Email: stefan@ripple.com
 URI: <https://www.ripple.com>

Rome Reginelli
Ripple
300 Montgomery Street
San Francisco, CA 94104
US

Phone: -----
Email: rome@ripple.com
URI: <https://www.ripple.com>

Adrian Hope-Bailie
Ripple
300 Montgomery Street
San Francisco, CA 94104
US

Phone: -----
Email: adrian@ripple.com
URI: <https://www.ripple.com>

