Network Working Group
Internet-Draft
Intended status: Standards Track
Expires: January 9, 2017

S. Thomas
Ripple
July 08, 2016

Crypto-Conditions
draft-thomas-crypto-conditions-00

Abstract

Crypto-conditions provide a mechanism to describe a signed message such that multiple actors in a distributed system can all verify the same signed message and agree on whether it matches the description. This provides a useful primitive for event-based systems that are distributed on the Internet since we can describe events in a standard deterministic manner (represented by signed messages) and therefore define generic authenticated event handlers.

Feedback

This specification is a part of the Interledger Protocol [1] work. Feedback related to this specification should be sent to public-interledger@w3.org [2].

Status of This Memo

Copyright Notice

Table of Contents

1. Introduction

This specification describes a message format for defining distributable event descriptions (crypto-conditions) and the cryptographically verifiable event messages (fulfillments) that can be used to prove that the event occurred.

The specification defines both binary and string-based encoding for the messages.

1.1. 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].

Within this specification, the term "condition" refers to the hash of a description of a signed message.

The term "fulfillment" refers to a description of a signed message and a signed message that matches the description.

The description can be hashed and compared to a condition. If the message matches the description and the hash of the description matches the condition, we say that the fulfillment fulfills the condition.

A "hashlock" is a tuple consisting of a bytestring and its hash where the hash is published first and the publication of the corresponding bytestring acts as a one-bit, one-time signature.

1.2. Features

Crypto-conditions are a simple multi-algorithm, multi-level, multi-signature standard format for expressing conditions and fulfillments.

1.2.1. Multi-Algorithm

Crypto-conditions can support several different signature and hash algorithms and support for new ones can be added in the future.

Implementations can state their supported algorithms simply by providing a bitmask. It is easy to verify that a given implementation will be able to verify the fulfillment to a given condition, by verifying that all bits that are set in the condition's bitmask are also set in the implementation's supported features bitmask.

Any new high bit can redefine the meaning of any existing lower bits when it is set. This can be used to remove obsolete algorithms.

The bitmask is encoded as a varint to minimize space usage.

By evaluating the bitmask of a condition actors in the system can establish, even before a fulfillment is published, if they will be able to verify the fulfilment.

1.2.2. 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.

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.

1.2.3. Multi-Level

Basic multi-sign is single-level and does not support more complex trust relationships such as "I trust Alice and Bob, but only when Candice also agrees". In single level 2-of-3 Alice and Bob could sign on their own, without Candice's approval.

Crypto-conditions add that flexibility elegantly, by applying thresholds not just to signatures, but to conditions which can be

signatures or further conditions. That allows the creation of an arbitrary threshold boolean circuit of signatures.

2. Format

2.1. Binary Encoding

An description of crypto-conditions is provided in this document using Abstract Syntax Notation One (ASN.1) as defined in [itu.X680.2015]. Implementations of this spec MUST support encoding and decoding using Octet Encoding Rules (OER) as defined in [itu.X696.2015].

2.2. String Types

BASE10 Variable-length integer encoded as a base-10 (decimal) number. Implementations MUST reject encodings that are too large for them to parse. Implementations MUST be tested for overflows.

BASE16 Variable-length integer encoded as a base-16 (hexadecimal) number. Implementations MUST reject encodings that are too large for them to parse. Implementations MUST be tested for overflows. No leading zeros.

BASE64URL Base64-URL encoding. See [RFC4648] , Section 5.

2.3. Bitmask

Any system accepting crypto-conditions must be able to state its supported algorithms. It must be possible to verify that all algorithms used in a certain condition are indeed supported even if the fulfillment is not available yet.

In order to meet these design goals, we define a bitmask to express the supported primitives.

Each bit represents a different suite of features. Each type of condition depends on one or more feature suites. If an implementation supports all feature suites that a certain type depends on, the implementation MUST support that condition type. The list of known types and feature suites is the IANA maintained Crypto-Condition Type Registry (Appendix E.1) .

Conditions contain a bitmask of types which they require the implementation to support. Implementations provide a bitmask of types they support.

2.4. Condition

Below are the string and binary encoding formats for a condition. In both, the featureBitmask is the boolean OR of the feature suite bitmasks of the top-level condition type and all subcondition types, recursively.

2.4.1. String Format

Conditions are ASCII encoded as:

```
"cc:" BASE16(type) ":" BASE16(featureBitmask) ":"
    BASE64URL(fingerprint) ":" BASE10(maxFulfillmentLength)
```

2.4.2. Binary Format

Conditions are binary encoded as:

```
Condition ::= SEQUENCE {
  type ConditionType,
  featureBitmask OCTET STRING,
  fingerprint OCTET STRING,
  maxFulfillmentLength INTEGER (0..MAX)
}

ConditionType ::= INTEGER {
  preimageSha256(0),
  rsaSha256(1),
  prefixSha256(2),
  thresholdSha256(3),
  ed25519(4)
} (0..65535)
```

2.4.3. Fields

type is the numeric type identifier representing the condition type.

featureBitmask is an octet string encoding the set of feature suites an implementation must support in order to be able to successfully parse the fulfillment to this condition.

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 entire string or binary encoded condition as the key, not just the fingerprint - as different conditions of different types may have the same fingerprint. The length and contents of the fingerprint are defined by the condition type. For most condition types, the

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

maxFulfillmentLength is the maximum length of the fulfillment payload that can fulfill this condition. When a crypto-condition is submitted to an implementation, this implementation MUST verify that it will be able to process a fulfillment with a payload of size maxFulfillmentLength.

2.5. Fulfillment

Below are the string and binary encoding formats for a fulfillment.

2.5.1. String Format

Fulfillments are ASCII encoded as:

```
"cf:" BASE16(type) ":" BASE64URL(payload)
```

2.5.2. Binary Format

Fulfillments are binary encoded as:

```
Fulfillment ::= SEQUENCE {
  type ConditionType,
  payload OCTET STRING
}
```

2.5.3. Fields

type is the numeric type identifier representing the condition type. For some condition types the fulfillment will contain further subfulfillments, however the type field always represents the outermost, or top-level, type.

payload The payload is an octet string whose internal format is defined by each of the types.

3. Feature Suites

The following feature suites are defined in this version of the specification. New feature suites may be defined in the future and will be registered in the IANA maintained Crypto-Condition Type Registry (Appendix E.1)

Support for a condition type MUST depend on one or more feature suites. Future versions of this spec MAY introduce new feature bits and condition types. However, all new condition types MUST depend on

at least one of the new feature suites. This ensures that all previously created implementations correctly recognize that they do not support the new type.

Feature suites are chosen such that they represent reasonable clusters of functionality. For instance, it is reasonable to require that an implementation which supports SHA-256 in one context MUST support it in all contexts, since it already needed to implement the algorithm.

An implementation which supports a certain set of feature suites MUST accept all condition types which depend only on that set or any subset of feature suites.

3.1. SHA-256

SHA-256 is a hashing algorithm and is assigned the feature bit 2^0 = 0x01.

3.2. PREIMAGE

PREIMAGE refers to hashlock conditions and is assigned the feature bit 2^1 = 0x02.

A preimage condition is the hash of its own fulfillment. In order to fulfill a preimage condition, a valid preimage must be provided.

Preimage conditions can be used as a so-called hashlock. Since cryptographically secure hashing functions are preimage-resistant, only the original creator of a preimage condition can feasibly produce the preimage if it contains a large amount of random entropy.

3.3. PREFIX

PREFIX is a structural condition and is assigned the feature bit 2^2 = 0x04.

A prefix condition condition contains exactly one subcondition. When validated it simply prepends the message to be validated with a constant string before passing it on to the subcondition's validation function.

3.4. THRESHOLD

THRESHOLD is a structural condition and is assigned the feature bit 2^3 = 0x08.

Threshold conditions provide a way to create m-of-n threshold combinations of other conditions such that m of the n subconditions have to be fulfilled in order for the threshold condition to be fulfilled.

Weights are also supported which allow one subcondition to count as multiple fulfilled subconditions towards the threshold.

3.5. RSA-PSS

RSA-PSS is a signature algorithm and is assigned the feature bit 2^4 = 0x10.

3.6. ED25519

ED25519 is a signature algorithm and is assigned the feature bit 2^5 = 0x20.

ED25519 is a compact elliptic curve based signature algorithm.

4. Condition Types

The following condition types are defined in this version of the specification. New types may be defined in the future and will be registered in the IANA maintained Crypto-Condition Type Registry (Appendix E.1)

4.1. PREIMAGE-SHA-256

PREIMAGE-SHA-256 is assigned the type ID 0. It relies on the SHA-256 and PREIMAGE feature suites which corresponds to a feature bitmask of 0x03.

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 cryptgraphically hard to fulfill, because they would have to find a preimage for the given condition hash.

Bitcoin supports this type of condition via the OP_HASH256 operator.

4.1.1. Condition

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

4.1.2. Fulfillment

The fulfillment payload of a PREIMAGE-SHA-256 condition is the preimage.

4.2. PREFIX-SHA-256

PREFIX-SHA-256 is assigned the type ID 1. It relies on the SHA-256 and PREFIX feature suites which corresponds to a feature bitmask of 0x05.

Prefix conditions provide a way to effective narrow the scope of other conditions. A condition can be used as the fingerprint of a public key to sign an arbitrary message. By creating a prefix subcondition we can narrow the scope from signing an arbitrary message to signing a message with a specific prefix.

When a prefix fulfillment is validated against a message, it will prepend the prefix to the provided message and will use the result as the message to validate against the subfulfillment.

4.2.1. Condition

The fingerprint of a PREFIX-SHA-256 condition is the SHA-256 digest of the fingerprint contents given below:

```
PrefixSha256FingerprintContents ::= SEQUENCE {
  prefix OCTET STRING,
  condition Condition
}
```

prefix is an arbitrary octet string which will be prepended to the message during validation.

condition is the subcondition which the subfulfillment must match.

4.2.2. Fulfillment

```
PrefixSha256FulfillmentPayload ::= SEQUENCE {
  prefix OCTET STRING,
  subfulfillment Fulfillment
}
```

prefix is an arbitrary octet string which will be prepended to the message during validation.

subfulfillment is the fulfilled subcondition.

4.3. THRESHOLD-SHA-256

THRESHOLD-SHA-256 is assigned the type ID 2. It relies on the SHA-256 and THRESHOLD feature suites which corresponds to a feature bitmask of 0x09.

4.3.1. Condition

The fingerprint of a THRESHOLD-SHA-256 condition is the SHA-256 digest of the fingerprint contents given below:

```
ThresholdSha256FingerprintContents ::= SEQUENCE {
  threshold INTEGER (0..4294967295),
  subconditions SEQUENCE OF ThresholdSubcondition
}

ThresholdSubcondition ::= SEQUENCE {
  weight INTEGER (0..4294967295),
  condition Condition
}
```

The list of conditions is sorted first based on length, shortest first. Elements of the same length are sorted in lexicographic (big-endian) order, smallest first.

threshold threshold MUST be an integer in the range 1 ... 2^32 - 1. In order to fulfill a threshold condition, the weights of the provided fulfillments MUST be greater than or equal to the threshold.

subconditions is the set of subconditions, each provided as a tuple of weight and condition.

weight is the numeric weight of this subcondition, i.e. how many times it counts against the threshold.

condition is the subcondition.

4.3.2. Fulfillment

```
ThresholdSha256FulfillmentPayload ::= SEQUENCE {
  threshold INTEGER (0..4294967295),
  subfulfillments SEQUENCE OF ThresholdSubfulfillment
}

ThresholdSubfulfillment ::= SEQUENCE {
  weight INTEGER (0..4294967295) DEFAULT 1,
  condition Condition OPTIONAL,
  fulfillment Fulfillment OPTIONAL
}
```

threshold is a number and MUST be an integer in the range 1 ... 2^32 - 1. In order to fulfill a threshold condition, the weights of the provided fulfillments MUST be greater than or equal to the threshold.

subfulfillments is the set of subconditions and subfulfillments, each provided as a tuple of weight and condition or fulfillment.

weight is the numeric weight of this subcondition, i.e. how many times it counts against the threshold. It MUST be an integer in the range 1 ... 2^32 - 1.

condition is the subcondition if this subcondition is not fulfilled.

fulfillment is the subfulfillment if this subcondition is fulfilled.

4.4. RSA-SHA-256

RSA-SHA-256 is assigned the type ID 3. It relies on the SHA-256 and RSA-PSS feature suites which corresponds to a feature bitmask of 0x11.

The signature algorithm used is RSASSA-PSS as defined in PKCS#1 v2.2. [RFC3447]

4.4.1. Condition

The fingerprint of a RSA-SHA-256 condition is the SHA-256 digest of the fingerprint contents given below:

The salt length for PSS is 32 bytes.

```
RsaSha256FingerprintContents ::= SEQUENCE {
  modulus OCTET STRING (SIZE(128..512))
}
```

modulus is an octet string representing the RSA public modulus in big-endian byte order. The first byte of the modulus MUST NOT be zero.

The corresponding public exponent e is assumed to be 65537 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] .

Implementations MUST reject moduli smaller than 128 bytes (1017 bits) or greater than 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] .

4.4.2. Fulfillment

```
RsaSha256FulfillmentPayload ::= SEQUENCE {
  modulus OCTET STRING (SIZE(128..512)),
  signature OCTET STRING (SIZE(128..512))
}
```

modulus is an octet string representing the RSA public modulus in big-endian byte order. See Section 4.4.1

signature is an octet string representing the RSA signature. It MUST be encoded in big-endian byte order with the exact same number of octets as the modulus, even if this means adding leading zeros. This ensures that the fulfillment size is constant and known ahead of time. Note that the field is still binary encoded with a length prefix for consistency.

Implementations MUST verify that the signature and modulus consist of the same number of octets and that the signature is numerically less than the modulus.

The message to be signed is provided separately. If no message is provided, the message is assumed to be an octet string of length zero.

4.4.3. Implementation

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.

4.5. ED25519

ED25519 is assigned the type ID 4. It relies only on the ED25519 feature suite which corresponds to a bitmask of 0x20.

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

Note: This document is not defining the SHA-512 versions of other condition types. In addition, the Ed25519 condition type is already uniquely identified by a corresponding Ed25519 feature suite. Therefore we intentionally do not introduce a SHA-512 feature suite in this document.

4.5.1. Condition

The fingerprint of a ED25519 condition is the 32 byte Ed25519 public key. Since the public key is already very small, we do not hash it.

4.5.2. Fulfillment

```
Ed25519FulfillmentPayload ::= SEQUENCE {
  publicKey OCTET STRING (SIZE(32)),
  signature OCTET STRING (SIZE(64))
}
```

publicKey is an octet string containing the Ed25519 public key.

signature is an octet string containing the Ed25519 signature.

5. References

5.1. Normative References

[I-D.irtf-cfrg-eddsa]
Josefsson, S. and I. Liusvaara, "Edwards-curve Digital Signature Algorithm (EdDSA)", draft-irtf-cfrg-eddsa-04 (work in progress), March 2016.

[itu.X680.2015]
International Telecommunications Union, "Information technology - Abstract Syntax Notation One (ASN.1): Specification of basic notation", n.d., <http://www.itu.int/rec/T-REC-X.680-201508-I/>.

[itu.X696.2015]
International Telecommunications Union, "Information technology - ASN.1 encoding rules: Specification of Octet Encoding Rules (OER)", n.d., <http://handle.itu.int/11.1002/1000/12487>.

[RFC3447] Jonsson, J. and B. Kaliski, "Public-Key Cryptography Standards (PKCS) #1: RSA Cryptography Specifications Version 2.1", RFC 3447, DOI 10.17487/RFC3447, February 2003, <http://www.rfc-editor.org/info/rfc3447>.

[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>.

5.2. 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, <http://crypto.stackexchange.com/questions/3110/impacts-of-not-using-rsa-exponent-of-65537>.

Appendix A. Security Considerations

TODO

Appendix B. Test Values

TODO

Appendix C. ASN.1 Module

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

CryptoConditions
DEFINITIONS
AUTOMATIC TAGS ::=
BEGIN

/**
* CONTAINERS
*/

Condition ::= SEQUENCE {
type ConditionType,
featureBitmask OCTET STRING,
fingerprint OCTET STRING,
maxFulfillmentLength INTEGER (0..MAX)
}

Fulfillment ::= SEQUENCE {
type ConditionType,
payload OCTET STRING
}

ConditionType ::= INTEGER {
preimageSha256(0),
rsaSha256(1),
prefixSha256(2),
```

```
thresholdSha256(3),
ed25519(4)
} (0..65535)

/**
* FULFILLMENT PAYLOADS
*/

-- For preimage conditions, the payload equals the preimage

PrefixSha256FulfillmentPayload ::= SEQUENCE {
prefix OCTET STRING,
subfulfillment Fulfillment
}

ThresholdSha256FulfillmentPayload ::= SEQUENCE {
threshold INTEGER (0..4294967295),
subfulfillments SEQUENCE OF ThresholdSubfulfillment
}

ThresholdSubfulfillment ::= SEQUENCE {
weight INTEGER (0..4294967295) DEFAULT 1,
condition Condition OPTIONAL,
fulfillment Fulfillment OPTIONAL
}

RsaSha256FulfillmentPayload ::= SEQUENCE {
modulus OCTET STRING (SIZE(128..512)),
signature OCTET STRING (SIZE(128..512))
}

Ed25519FulfillmentPayload ::= SEQUENCE {
publicKey OCTET STRING (SIZE(32)),
signature OCTET STRING (SIZE(64))
}

/**
* FINGERPRINTS
*/

-- SHA-256 hash of the fingerprint contents
Sha256Fingerprint ::= OCTET STRING (SIZE(32)) -- digest

-- 32-byte Ed25519 public key
Ed25519Fingerprint ::= OCTET STRING (SIZE(32)) -- publicKey

/**
* FINGERPRINT CONTENTS
```

```
*
* The content that will be hashed to arrive at the fingerprint.
*/

-- The preimage type hashes the raw contents of the preimage

PrefixSha256FingerprintContents ::= SEQUENCE {
prefix OCTET STRING,
condition Condition
}

ThresholdSha256FingerprintContents ::= SEQUENCE {
threshold INTEGER (0..4294967295),
subconditions SEQUENCE OF ThresholdSubcondition
}

ThresholdSubcondition ::= SEQUENCE {
weight INTEGER (0..4294967295),
condition Condition
}

RsaSha256FingerprintContents ::= INTEGER (0..MAX) -- modulus

/**
* EXAMPLES
*/

exampleCondition Condition ::=
{
type preimageSha256,
featureBitmask '03'H,
fingerprint '
E3B0C442 98FC1C14 9AFBF4C8 996FB924 27AE41E4 649B934C A495991B 7852B855
'H,
maxFulfillmentLength 2
}

exampleFulfillment Fulfillment ::=
{
type preimageSha256,
payload '00'H
}

exampleRsaSha256FulfillmentPayload RsaSha256FulfillmentPayload ::=
{
modulus '
B30E7A93 8783BABF 836850FF 49E14F87 E3F92D5C 46E33FEC A3E4F0B2 2358580B
11765995 F4B8EEA7 FB4712C2 E1E316F7 F775A953 D232216A 169D9A64 DDC00712
```

```
 0A400B37 F2AFC077 B62FE304 DE74DE6A 119EC407 6B529C4F 6096B0BA AD4F533D
 F0173B9B 822FD85D 65FA4BEF A92D8F52 4F69CBCA 0136BD80 D095C169 AEC0E095
 '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 DECBBDE5 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
 }

 exampleEd25519FulfillmentPayload Ed25519FulfillmentPayload ::=
 {
 publicKey '
 EC172B93 AD5E563B F4932C70 E1245034 C35467EF 2EFD4D64 EBF81968 3467E2BF
 'H,
 signature '
 B62291FA D9432F8F 298B9C4A 4895DBE2 93F6FFDA 1A68DADF 0CCDEF5F 47A0C721
 2A5FEA3C DA97A3F4 C03EA9F2 E8AC1CEC 86A51D45 2127ABDB A09D1B6F 331C070A
 'H
 }

 END
```

Appendix D. Acknowledgements

The editor would like to thank the following individuals for feedback on and implementations of the specification (in alphabetical order): TODO

Appendix E. IANA Considerations

E.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 feature suite bits are registered:

Type Bit	Exp.	Hex	Condition Types
1	2^0	0x01	SHA-256
10	2^1	0x02	PREIMAGE
100	2^2	0x04	PREFIX
1000	2^3	0x08	THRESHOLD
10000	2^4	0x10	RSA
100000	2^5	0x20	ED25519

Table 1: Crypto-Condition Feature Suites

The following types are registered:

Type ID	Required Bitmask	Type Name
0	0x03	PREIMAGE-SHA-256
1	0x05	PREFIX-SHA-256
2	0x09	THRESHOLD-SHA-256
3	0x11	RSA-SHA-256
4	0x20	ED25519

Table 2: Crypto-Condition Types

Author's Address

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

Phone: -----------------
Email: stefan@ripple.com
URI: http://www.ripple.com