Crypto Forum
Internet-Draft
Intended status: Informational
Expires: 31 August 2026

E. Lundberg, Ed.
J. Bradley
Yubico

27 February 2026

The Asynchronous Remote Key Generation (ARKG) algorithm
draft-bradleylundberg-cfrg-arkg-10

Abstract

Asynchronous Remote Key Generation (ARKG) is an abstract algorithm that enables delegation of asymmetric public key generation without giving access to the corresponding private keys. This capability enables a variety of applications: a user agent can generate pseudonymous public keys to prevent tracking; a message sender can generate ephemeral recipient public keys to enhance forward secrecy; two paired authentication devices can each have their own private keys while each can register public keys on behalf of the other.

This document provides three main contributions: a specification of the generic ARKG algorithm using abstract primitives; a set of formulae for instantiating the abstract primitives using concrete primitives; and an initial set of fully specified concrete ARKG instances. We expect that additional instances will be defined in the future.

About This Document

This note is to be removed before publishing as an RFC.

Status information for this document may be found at https://datatracker.ietf.org/doc/draft-bradleylundberg-cfrg-arkg/.

Source for this draft and an issue tracker can be found at https://github.com/Yubico/arkg-rfc.

Status of This Memo

Copyright Notice

Table of Contents

1. Introduction

Asynchronous Remote Key Generation (ARKG) introduces a mechanism to generate public keys without access to the corresponding private keys. Such a mechanism is useful for many scenarios when a new public key is needed but the private key holder is not available to perform the key generation. This may occur when private keys are stored in a hardware security device, which may be unavailable or locked at the time a new public key is needed.

Some motivating use cases of ARKG include:

* *Single-use asymmetric keys*: Envisioned for the European Union's digital identity framework, which is set to use single-use asymmetric keys to prevent colluding verifiers from using public keys as correlation handles. Each digital identity credential would thus be issued with a single-use proof-of-possession key, used only once to present the credential to a verifier. ARKG empowers both online and offline usage scenarios: for offline scenarios, ARKG enables pre-generation of public keys for single-use credentials without needing to access the hardware security device that holds the private keys. For online scenarios, ARKG gives the credential issuer assurance that all derived private

keys are bound to the same secure hardware element. In both cases, application performance may be improved since public keys can be generated in a general-purpose execution environment instead of a secure enclave.

* *Enhanced forward secrecy*: The use of ARKG can facilitate forward secrecy in certain contexts. For instance, section 8.5.4 of RFC 9052 (https://www.rfc-editor.org/rfc/rfc9052.html#name-direct-key-agreement) notes that "Since COSE is designed for a store-and-forward environment rather than an online environment, [...] forward secrecy (see [RFC4949]) is not achievable. A static key will always be used for the receiver of the COSE object." As opposed to workarounds like exchanging a large number of keys in advance, ARKG enables the the sender to generate ephemeral recipient public keys on demand.

* *Backup key generation*: For example, the W3C Web Authentication API [WebAuthn] (WebAuthn) generates a new key pair for each account on each web site. ARKG could allow for simultaneously generating a backup public key when registering a new public key. A primary authenticator could generate both a key pair for itself and a public key for a paired backup authenticator. The backup authenticator only needs to be paired with the primary authenticator once, and can then be safely stored until it is needed.

ARKG consists of three procedures:

* *Initialization*: The _delegating party_ generates a _seed pair_ and discloses the _public seed_ to a _subordinate party_, while securely retaining the _private seed_.

* *Public key generation*: The subordinate party uses the public seed to autonomously generate a new public key along with a unique _key handle_ for the public key. This can be repeated any number of times.

* *Private key derivation*: The delegating party uses a key handle and the private seed to derive the private key corresponding to the public key generated along with the key handle. This can be repeated with any number of key handles.

Notably, ARKG can be built entirely using established cryptographic primitives. The required primitives are a public key blinding scheme and a key encapsulation mechanism (KEM), which may in turn use a key derivation function (KDF) and a message authentication code (MAC) scheme. Both conventional primitives and quantum-resistant alternatives exist that meet these requirements. [ASIACCS_SteWil24]

1.1. Requirements Language

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "NOT RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in BCP 14 [RFC2119] [RFC8174] when, and only when, they appear in all capitals, as shown here.

1.2. Notation

The following notation is used throughout this document:

* The symbol || represents octet string concatenation.

* Literal text strings and octet strings are denoted using the CDDL syntax defined in Section 3.1 of [RFC8610].

* Elliptic curve operations are written in additive notation: + denotes point addition, i.e., the curve group operation; * denotes point multiplication, i.e., repeated point addition; and + also denotes scalar addition modulo the curve order. * has higher precedence than +, i.e., a + b * C is equivalent to a + (b * C).

* LEN(x) is the length, in octets, of the octet string x.

* The function I2OSP converts a nonnegative integer into an octet string as defined in Section 4.1 of [RFC8017].

2. The Asynchronous Remote Key Generation (ARKG) algorithm

The ARKG algorithm consists of three functions, each performed by one of two participants: the _delegating party_ or the _subordinate party_. The delegating party generates an ARKG _seed pair_ and emits the _public seed_ to the subordinate party while keeping the _private seed_ secret. The subordinate party can then use the public seed to generate derived public keys and _key handles_, and the delegating party can use the private seed and a key handle to derive the corresponding private key.

This construction of ARKG is fully deterministic, extracting input entropy as explicit parameters, as opposed to the internal random sampling typically used in the academic literature [CCS_FGKLMN20] [Wilson2023] [AC_BreCleFis24]. Implementations MAY choose to instead implement the ARKG-Derive-Seed and KEM-Encaps functions as nondeterministic procedures omitting their respective ikm parameters and sampling random entropy internally; this choice does not affect interoperability between the functions ARKG-Derive-Seed, ARKG-Derive-Public-Key and ARKG-Derive-Private-Key.

The following subsections define the abstract instance parameters used to construct the three ARKG functions, followed by the definitions of the three ARKG functions.

2.1. Instance parameters

ARKG is composed of a suite of other algorithms. The parameters of an ARKG instance are:

* BL: An asymmetric key blinding scheme [Wilson2023], consisting of:

  - Function BL-Generate-Keypair() -> (pk, sk): Generate a blinding key pair.

    Input consists of input keying material entropy ikm.

    Output consists of a blinding public key pk and a blinding private key sk.

  - Function BL-PRF(ikm_tau, ctx) -> tau: Derive a pseudorandom blinding factor.

    Input consists of input entropy ikm_tau and a domain separation parameter ctx.

    Output consists of the blinding factor tau.

  - Function BL-Blind-Public-Key(pk, tau) -> pk_tau: Deterministically compute a blinded public key.

    Input consists of a blinding public key pk, and a blinding factor tau.

    Output consists of the blinded public key pk_tau.

  - Function BL-Blind-Private-Key(sk, tau) -> sk_tau: Deterministically compute a blinded private key.

    Input consists of a blinding private key sk, and the blinding factor tau.

    Output consists of the blinded private key sk_tau.

  ikm is an opaque octet string of a suitable length as defined by the ARKG instance. ikm_tau is an opaque octet string generated as the k output of KEM-Encaps and KEM-Decaps. ctx is an opaque octet string of arbitrary length.

The representations of pk and pk_tau are defined by the protocol that invokes ARKG. The representations of sk, tau and sk_tau are undefined implementation details.

See [Wilson2023] for definitions of security properties required of the key blinding scheme BL.

* KEM: A key encapsulation mechanism [Shoup], consisting of the functions:

  - KEM-Derive-Key-Pair(ikm) -> (pk, sk): Derive a key encapsulation key pair.

    Input consists of input keying material entropy ikm.

    Output consists of public key pk and private key sk.

  - KEM-Encaps(pk, ikm, ctx) -> (k, c): Derive a key encapsulation.

    Input consists of an encapsulation public key pk, input entropy ikm and a domain separation parameter ctx.

    Output consists of a shared secret k and an encapsulation ciphertext c.

  - KEM-Decaps(sk, c, ctx) -> k: Decapsulate a shared secret.

    Input consists of encapsulation private key sk, encapsulation ciphertext c and a domain separation parameter ctx.

    Output consists of the shared secret k on success, or an error otherwise.

  ikm is an opaque octet string of a suitable length as defined by the ARKG instance. k, c and ctx are opaque octet strings of arbitrary length. The representation of pk is defined by the protocol that invokes ARKG. The representation of sk is an undefined implementation detail.

  The KEM MUST guarantee integrity of the ciphertext, meaning that knowledge of the public key pk and the domain separation parameter ctx is required in order to create any ciphertext c that can be successfully decapsulated by the corresponding private key sk. Section 3.2 describes a general formula for how any KEM can be adapted to include this guarantee. Section 9.1 discusses the reasons for this requirement.

See [ASIACCS_SteWil24] for definitions of additional security properties required of the key encapsulation mechanism KEM.

A concrete ARKG instantiation MUST specify the instantiation of each of the above functions.

The output keys of the BL scheme are also the output keys of the ARKG instance as a whole. For example, if BL-Blind-Public-Key and BL-Blind-Private-Key output ECDSA keys, then the ARKG instance will also output ECDSA keys.

We denote a concrete ARKG instance by the pattern ARKG-NAME, substituting for NAME some description of the chosen instantiation for BL and KEM. Note that this pattern cannot in general be unambiguously parsed; implementations MUST NOT attempt to construct an ARKG instance by parsing such a pattern string. Concrete ARKG instances MUST always be identified by lookup in a registry of fully specified ARKG instances. This is to prevent usage of algorithm combinations that may be incompatible or insecure.

2.2. The function ARKG-Derive-Seed

This function is performed by the delegating party. The delegating party derives the ARKG seed pair (pk, sk) and keeps the private seed sk secret, while the public seed pk is provided to the subordinate party. The subordinate party will then be able to derive public keys on behalf of the delegating party.

```
ARKG-Derive-Seed(ikm_bl, ikm_kem) -> (pk, sk)
    ARKG instance parameters:
        BL        A key blinding scheme.
        KEM       A key encapsulation mechanism.

    Inputs:
        ikm_bl    Input keying material entropy for BL.
        ikm_kem   Input keying material entropy for KEM.

    Output:
        (pk, sk)  An ARKG seed pair with public seed pk
                    and private seed sk.

    The output (pk, sk) is calculated as follows:

    (pk_bl,  sk_bl)  = BL-Derive-Key-Pair(ikm_bl)
    (pk_kem, sk_kem) = KEM-Derive-Key-Pair(ikm_kem)
    pk = (pk_bl, pk_kem)
    sk = (sk_bl, sk_kem)
```

2.2.1. Nondeterministic variants

Applications that do not need a deterministic interface MAY choose to instead implement ARKG-Derive-Seed, KEM-Derive-Key-Pair and BL-Derive-Key-Pair as nondeterministic procedures omitting their respective ikm parameters and sampling random entropy internally; this choice does not affect interoperability with ARKG-Derive-Public-Key and ARKG-Derive-Private-Key.

2.3. The function ARKG-Derive-Public-Key

This function is performed by the subordinate party, which holds the ARKG public seed pk = (pk_bl, pk_kem). The resulting public key pk' can be provided to external parties to use in asymmetric cryptography protocols, and the resulting key handle kh can be used by the delegating party to derive the private key corresponding to pk'.

This function may be invoked any number of times with the same public seed, using different ikm or ctx arguments, in order to generate any number of public keys.

```
ARKG-Derive-Public-Key((pk_bl, pk_kem), ikm, ctx) -> (pk', kh)
    ARKG instance parameters:
        BL        A key blinding scheme.
        KEM       A key encapsulation mechanism.

    Inputs:
        pk_bl     A key blinding public key.
        pk_kem    A key encapsulation public key.
        ikm       Input entropy for KEM encapsulation.
        ctx       An octet string of length at most 64,
                    containing optional context and
                    application specific information
                    (can be a zero-length string).

    Output:
        pk'       A blinded public key.
        kh        A key handle for deriving the blinded
                    private key sk' corresponding to pk'.

    The output (pk', kh) is calculated as follows:

    if LEN(ctx) > 64:
        Abort with an error.

    ctx'    = I2OSP(LEN(ctx), 1) || ctx
    ctx_bl  = 'ARKG-Derive-Key-BL.'  || ctx'
    ctx_kem = 'ARKG-Derive-Key-KEM.' || ctx'

    (ikm_tau, c) = KEM-Encaps(pk_kem, ikm, ctx_kem)
    tau = BL-PRF(ikm_tau, ctx_bl)
    pk' = BL-Blind-Public-Key(pk_bl, tau)

    kh = c
```

If this procedure aborts due to an error, the procedure can safely be retried with the same (pk_bl, pk_kem) and ctx arguments but a new ikm argument.

See Section 2.5 for guidance on using ctx arguments longer than 64 bytes.

2.3.1. Nondeterministic variants

Applications that do not need a deterministic interface MAY choose to instead implement ARKG-Derive-Public-Key and KEM-Encaps as nondeterministic procedures omitting their respective ikm parameter and sampling random entropy internally; this choice does not affect interoperability with ARKG-Derive-Private-Key.

BL-PRF and BL-Blind-Public-Key must always be deterministic for compatibility with ARKG-Derive-Private-Key.

2.4. The function ARKG-Derive-Private-Key

This function is performed by the delegating party, which holds the ARKG private seed (sk_bl, sk_kem). The resulting private key sk' can be used in asymmetric cryptography protocols to prove possession of sk' to an external party that has the corresponding public key.

This function may be invoked any number of times with the same private seed, in order to derive the same or different private keys any number of times.

```
ARKG-Derive-Private-Key((sk_bl, sk_kem), kh, ctx) -> sk'
    ARKG instance parameters:
        BL        A key blinding scheme.
        KEM       A key encapsulation mechanism.

    Inputs:
        sk_bl     A key blinding private key.
        sk_kem    A key encapsulation private key.
        kh        A key handle output from ARKG-Derive-Public-Key.
        ctx       An octet string of length at most 64,
                    containing optional context and
                    application specific information
                    (can be a zero-length string).

    Output:
        sk'       A blinded private key.

    The output sk' is calculated as follows:

    if LEN(ctx) > 64:
        Abort with an error.

    ctx'    = I2OSP(LEN(ctx), 1) || ctx
    ctx_bl  = 'ARKG-Derive-Key-BL.'  || ctx'
    ctx_kem = 'ARKG-Derive-Key-KEM.' || ctx'

    ikm_tau = KEM-Decaps(sk_kem, kh, ctx_kem)
    If decapsulation failed:
        Abort with an error.

    tau = BL-PRF(ikm_tau, ctx_bl)
    sk' = BL-Blind-Private-Key(sk_bl, tau)
```

Errors in this procedure are typically unrecoverable. For example, KEM-Decaps may fail to decapsulate the KEM ciphertext kh if it fails an integrity check. ARKG instantiations SHOULD be chosen in a way that such errors are impossible if kh was generated by an honest and correct implementation of ARKG-Derive-Public-Key. Incorrect or malicious implementations of ARKG-Derive-Public-Key do not degrade the security of an honest and correct implementation of ARKG-Derive-Private-Key. See also Section 9.1.

See Section 2.5 for guidance on using ctx arguments longer than 64 bytes.

2.5. Using ctx values longer than 64 bytes

The ctx parameter of ARKG-Derive-Public-Key and ARKG-Derive-Private-Key is limited to a length of at most 64 bytes. This is because this value needs to be communicated from the _subordinate party_ to the _delegating party_ to use the same argument value in both functions, therefore it is necessary in some contexts to limit the size of this parameter in order to limit the size of overall protocol messages.

If applications require ctx values longer than 64 bytes, implementors MAY use techniques such as that described in Section 5.3.3 of [RFC9380]. Precise procedure definitions are left as an application-specific implementation detail.

3. Generic ARKG instantiations

This section defines generic formulae for instantiating the individual ARKG parameters, which can be used to define concrete ARKG instantiations.

3.1. Using elliptic curve addition for key blinding

Instantiations of ARKG whose output keys are elliptic curve keys can use elliptic curve addition as the key blinding scheme BL [CCS_FGKLMN20] [Wilson2023]. This section defines a general formula for such instantiations of BL.

This formula has the following parameters:

* crv: An elliptic curve.

* hash-to-crv-suite: A hash-to-curve suite [RFC9380] suitable for hashing to the scalar field of crv.

* DST_ext: A domain separation tag.

Then the BL parameter of ARKG may be instantiated as follows:

* G is the generator of the prime order subgroup of crv.

* N is the order of G.

* The function hash_to_field is defined in Section 5 of [RFC9380].

BL-Derive-Key-Pair(ikm) -> (pk, sk)

```
DST_bl_sk = 'ARKG-BL-EC-KG.' || DST_ext
sk = hash_to_field(ikm, 1) with the parameters:
    DST: DST_bl_sk
    F: GF(N), the scalar field
       of the prime order subgroup of crv
    p: N
    m: 1
    L: The L defined in hash-to-crv-suite
    expand_message: The expand_message function
                    defined in hash-to-crv-suite

pk = sk * G
```

BL-PRF(ikm_tau, ctx) -> tau

```
DST_tau = 'ARKG-BL-EC.' || DST_ext || ctx
tau = hash_to_field(ikm_tau, 1) with the parameters:
    DST: DST_tau
    F: GF(N), the scalar field
       of the prime order subgroup of crv
    p: N
    m: 1
    L: The L defined in hash-to-crv-suite
    expand_message: The expand_message function
                    defined in hash-to-crv-suite
```

BL-Blind-Public-Key(pk, tau) -> pk_tau

```
pk_tau = pk + tau * G
```

BL-Blind-Private-Key(sk, tau) -> sk_tau

```
sk_tau_tmp = sk + tau
If sk_tau_tmp = 0, abort with an error.
sk_tau = sk_tau_tmp
```

3.2. Using HMAC to adapt a KEM without ciphertext integrity

Not all key encapsulation mechanisms guarantee ciphertext integrity, meaning that a valid KEM ciphertext can be created only with knowledge of the KEM public key. This section defines a general formula for adapting any KEM to guarantee ciphertext integrity by prepending a MAC to the KEM ciphertext.

For example, ECDH does not guarantee ciphertext integrity - any elliptic curve point is a valid ECDH ciphertext and can be successfully decapsulated using any elliptic curve private scalar.

This formula has the following parameters:

* Hash: A cryptographic hash function.

* DST_ext: A domain separation parameter.

* Sub-Kem: A key encapsulation mechanism as described for the KEM parameter in Section 2.1, except Sub-Kem MAY ignore the ctx parameter and MAY not guarantee ciphertext integrity. Sub-Kem defines the functions Sub-Kem-Derive-Key-Pair, Sub-Kem-Encaps and Sub-Kem-Decaps.

The KEM parameter of ARKG may be instantiated using Sub-Kem, HMAC [RFC2104] and HKDF [RFC5869] as follows:

* L is the output length of Hash in octets.

* LEFT(X, n) is the first n bytes of the byte array X.

* DROP_LEFT(X, n) is the byte array X without the first n bytes.

We truncate the HMAC output to 128 bits (16 octets) because as described in Section 9.1, ARKG needs ciphertext integrity only to ensure correctness, not for security. Extendable-output functions used as the Hash parameter SHOULD still be instantiated with an output length appropriate for the desired security level, in order to not leak information about the Sub-Kem shared secret key.

```
KEM-Derive-Key-Pair(ikm) -> (pk, sk)

    (pk, sk) = Sub-Kem-Derive-Key-Pair(ikm)


KEM-Encaps(pk, ikm, ctx) -> (k, c)

    ctx_sub = 'ARKG-KEM-HMAC.' || DST_ext || ctx
```

```
    (k', c') = Sub-Kem-Encaps(pk, ikm, ctx_sub)

    prk = HKDF-Extract with the arguments:
        Hash: Hash
        salt: not set
        IKM: k'

    info_mk = 'ARKG-KEM-HMAC-mac.' || DST_ext || ctx
    mk = HKDF-Expand with the arguments:
        Hash: Hash
        PRK: prk
        info: info_mk
        L: L
    t = HMAC-Hash-128(K=mk, text=c')

    info_k = 'ARKG-KEM-HMAC-shared.' || DST_ext || ctx
    k = HKDF-Expand with the arguments:
        Hash: Hash
        PRK: prk
        info: info_k
        L: The length of k' in octets.
    c = t || c'


KEM-Decaps(sk, c, ctx) -> k

    t = LEFT(c, 16)
    c' = DROP_LEFT(c, 16)
    ctx_sub = 'ARKG-KEM-HMAC.' || DST_ext || ctx
    k' = Sub-Kem-Decaps(sk, c', ctx_sub)

    prk = HKDF-Extract with the arguments:
        Hash: Hash
        salt: not set
        IKM: k'

    mk = HKDF-Expand with the arguments:
        Hash: Hash
        PRK: prk
        info: 'ARKG-KEM-HMAC-mac.' || DST_ext || ctx
        L: L

    t' = HMAC-Hash-128(K=mk, text=c')
    If t = t':
        k = HKDF-Expand with the arguments:
            Hash: Hash
            PRK: prk
            info: 'ARKG-KEM-HMAC-shared.' || DST_ext || ctx
```

```
            L: The length of k' in octets.
    Else:
        Abort with an error.
```

In concrete instances where Sub-Kem-Encaps and Sub-Kem-Decaps ignore the ctx parameter, implementations MAY eliminate the parameter and omit the computation of ctx_sub.

3.3. Using ECDH as the KEM

Instantiations of ARKG can use ECDH [RFC6090] as the key encapsulation mechanism KEM [CCS_FGKLMN20] [ASIACCS_SteWil24]. This section defines a general formula for such instantiations of KEM.

This formula has the following parameters:

* crv: an elliptic curve valid for use with ECDH [RFC6090].

* Hash: A cryptographic hash function.

* hash-to-crv-suite: A hash-to-curve suite [RFC9380] suitable for hashing to the scalar field of crv.

* DST_ext: A domain separation parameter.

The above parameters define the following intermediate value:

* DST_aug: 'ARKG-ECDH.' || DST_ext.

The KEM parameter of ARKG may be instantiated as described in section Section 3.2 with the parameters:

* Hash: Hash.

* DST_ext: DST_aug.

* Sub-Kem: The functions Sub-Kem-Derive-Key-Pair, Sub-Kem-Encaps and Sub-Kem-Decaps defined as follows:

  - Elliptic-Curve-Point-to-Octet-String and Octet-String-to-Elliptic-Curve-Point are the conversion routines defined in sections 2.3.3 and 2.3.4 of [SEC1], without point compression.

  - ECDH(pk, sk) represents the compact output of ECDH [RFC6090] using public key (curve point) pk and private key (exponent) sk.

  - G is the generator of the prime order subgroup of crv.

- N is the order of G.

```
Sub-Kem-Derive-Key-Pair(ikm) -> (pk, sk)

    DST_kem_sk = 'ARKG-KEM-ECDH-KG.' || DST_aug
    sk = hash_to_field(ikm, 1) with the parameters:
        DST: DST_kem_sk
        F: GF(N), the scalar field
          of the prime order subgroup of crv
        p: N
        m: 1
        L: The L defined in hash-to-crv-suite
        expand_message: The expand_message function
                        defined in hash-to-crv-suite

    pk = sk * G


Sub-Kem-Encaps(pk, ikm, ctx) -> (k, c)

    (pk', sk') = Sub-Kem-Derive-Key-Pair(ikm)

    k = ECDH(pk, sk')
    c = Elliptic-Curve-Point-to-Octet-String(pk')


Sub-Kem-Decaps(sk, c, ctx) -> k

    pk' = Octet-String-to-Elliptic-Curve-Point(c)
    k = ECDH(pk', sk)
```

Note: This instance intentionally ignores the ctx parameter of Sub-Kem-Encaps and Sub-Kem-Decaps.

3.4. Using X25519 or X448 as the KEM

Instantiations of ARKG can use X25519 or X448 [RFC7748] as the key encapsulation mechanism KEM. This section defines a general formula for such instantiations of KEM.

This formula has the following parameters:

* DH-Function: the function X25519 or the function X448 [RFC7748].

* DST_ext: A domain separation parameter.

The KEM parameter of ARKG may be instantiated as described in section Section 3.2 with the parameters:

* Hash: SHA-512 [FIPS 180-4] if DH-Function is X25519, or SHAKE256 [FIPS 202] with output length 64 octets if DH-Function is X448.

* DST_ext: 'ARKG-ECDHX.' || DST_ext.

* Sub-Kem: The functions Sub-Kem-Derive-Key-Pair, Sub-Kem-Encaps and Sub-Kem-Decaps defined as follows:

  - G is the octet string h'0900000000000000 0000000000000000 0000000000000000 0000000000000000' if DH-Function is X25519, or the octet string h'0500000000000000 0000000000000000 0000000000000000 0000000000000000 0000000000000000 0000000000000000' if DH-Function is X448.

    These are the little-endian encodings of the integers 9 and 5, which is the u-coordinate of the generator point of the respective curve group.

```
Sub-Kem-Derive-Key-Pair(ikm) -> (pk, sk)

    sk = ikm
    pk = DH-Function(sk, G)


Sub-Kem-Encaps(pk, ikm, ctx) -> (k, c)

    (pk', sk') = Sub-Kem-Derive-Key-Pair(ikm)

    k = DH-Function(sk', pk)
    c = pk'


Sub-Kem-Decaps(sk, c, ctx) -> k

    k = DH-Function(sk, c)
```

Note: This instance intentionally ignores the ctx parameter of Sub-Kem-Encaps and Sub-Kem-Decaps.

3.5. Using the same key for both key blinding and KEM

When an ARKG instance uses the same type of key for both the key blinding and the KEM - for example, if elliptic curve arithmetic is used for key blinding as described in Section 3.1 and ECDH is used as the KEM as described in Section 3.3 [CCS_FGKLMN20] - then the two keys MAY be the same key. Representations of such an ARKG seed MAY allow for omitting the second copy of the constituent key, but such representations MUST clearly identify that the single constituent key

is to be used both as the key blinding key and the KEM key.

4. Concrete ARKG instantiations

This section defines an initial set of concrete ARKG instantiations.

TODO: IANA registry? COSE/JOSE?

4.1. ARKG-P256

The identifier ARKG-P256 represents the following ARKG instance:

* BL: Elliptic curve addition as described in Section 3.1 with the parameters:
  - crv: The NIST curve secp256r1 [SEC2].
  - hash-to-crv-suite: P256_XMD:SHA-256_SSWU_RO_ [RFC9380].
  - DST_ext: 'ARKG-P256'.
* KEM: ECDH as described in Section 3.3 with the parameters:
  - crv: The NIST curve secp256r1 [SEC2].
  - Hash: SHA-256 [FIPS 180-4].
  - hash-to-crv-suite: P256_XMD:SHA-256_SSWU_RO_ [RFC9380].
  - DST_ext: 'ARKG-P256'.

Each ikm_bl, ikm_kem and ikm input to the procedures in this ARKG instance SHOULD contain at least 256 bits of entropy.

4.2. ARKG-P384

The identifier ARKG-P384 represents the following ARKG instance:

* BL: Elliptic curve addition as described in Section 3.1 with the parameters:
  - crv: The NIST curve secp384r1 [SEC2].
  - hash-to-crv-suite: P384_XMD:SHA-384_SSWU_RO_ [RFC9380].
  - DST_ext: 'ARKG-P384'.
* KEM: ECDH as described in Section 3.3 with the parameters:

- crv: The NIST curve secp384r1 [SEC2].
- Hash: SHA-384 [FIPS 180-4].
- hash-to-crv-suite: P384_XMD:SHA-384_SSWU_RO_ [RFC9380].
- DST_ext: 'ARKG-P384'.

Each ikm_bl, ikm_kem and ikm input to the procedures in this ARKG instance SHOULD contain at least 384 bits of entropy.

4.3. ARKG-P521

The identifier ARKG-P521 represents the following ARKG instance:

* BL: Elliptic curve addition as described in Section 3.1 with the parameters:
  - crv: The NIST curve secp521r1 [SEC2].
  - hash-to-crv-suite: P521_XMD:SHA-512_SSWU_RO_ [RFC9380].
  - DST_ext: 'ARKG-P521'.
* KEM: ECDH as described in Section 3.3 with the parameters:
  - crv: The NIST curve secp521r1 [SEC2].
  - Hash: SHA-512 [FIPS 180-4].
  - hash-to-crv-suite: P521_XMD:SHA-512_SSWU_RO_ [RFC9380].
  - DST_ext: 'ARKG-P521'.

Each ikm_bl, ikm_kem and ikm input to the procedures in this ARKG instance SHOULD contain at least 512 bits of entropy.

4.4. ARKG-P256k

The identifier ARKG-P256k represents the following ARKG instance:

* BL: Elliptic curve addition as described in Section 3.1 with the parameters:
  - crv: The SECG curve secp256k1 [SEC2].
  - hash-to-crv-suite: secp256k1_XMD:SHA-256_SSWU_RO_ [RFC9380].

- DST_ext: ’ARKG-P256k’.

* KEM: ECDH as described in Section 3.3 with the parameters:

  - crv: The SECG curve secp256k1 [SEC2].

  - Hash: SHA-256 [FIPS 180-4].

  - hash-to-crv-suite: secp256k1_XMD:SHA-256_SSWU_RO_ [RFC9380].

  - DST_ext: ’ARKG-P256k’.

Each ikm_bl, ikm_kem and ikm input to the procedures in this ARKG instance SHOULD contain at least 256 bits of entropy.

5. COSE bindings

This section proposes additions to COSE [RFC9052] to support ARKG use cases. These consist of a new key type to represent ARKG public seeds, algorithm identifiers for signing using an ARKG-derived private key, and new COSE_Sign_Args [I-D.lundberg-cose-split-algs] algorithm parameters for ARKG.

5.1. COSE key type: ARKG public seed

An ARKG public seed is represented as a COSE_Key structure [RFC9052] with kty value TBD (placeholder value -65537). Table 1 defines key type parameters pkbl (-1) and pkkem (-2) for the BL and KEM public key, respectively, as well as key type parameter dkalg (-3), representing the algorithm that derived public and private keys are to be used with.

Name	Label	Value type	Required?	Description
pkbl	-1	COSE_Key	Required	BL key of ARKG public seed
pkkem	-2	COSE_Key	Required	KEM key of ARKG public seed
dkalg	-3	int / tstr	Optional	alg parameter of public and private keys derived from this ARKG public seed

Table 1: COSE key type parameters for the ARKG-pub key type.

When dkalg (-3) is present in an ARKG public seed, the alg (3) parameter of public keys derived using ARKG-Derive-Public-Key with that seed SHOULD be set to the dkalg (-3) value of the seed.

The alg (3) parameter, when present, identifies the ARKG instance this public seed is to be used with. An initial set of COSE algorithm identifiers for this purpose is defined in Section 5.2.

The following CDDL [RFC8610] example represents an ARKG-P256 public seed restricted to generating derived keys for use with the ESP256 [RFC9864] signature algorithm:

```
{
  1: -65537,   ; kty: ARKG-pub (placeholder value)
               ; kid: Opaque identifier
  2: h'60b6dfddd31659598ae5de49acb220d8
       704949e84d484b68344340e2565337d2',
  3: -65700,   ; alg: ARKG-P256 (placeholder value)

  -1: {        ; BL public key
    1: 2,      ; kty: EC2
    -1: 1,     ; crv: P256
    -2: h'69380FC1C3B09652134FEEFBA61776F9
          7AF875CE46CA20252C4165102966EBC5',
    -3: h'8B515831462CCB0BD55CBA04BFD50DA6
          3FAF18BD845433622DAF97C06A10D0F1',
  },

  -2: {        ; KEM public key
    1: 2,      ; kty: EC2
    -1: 1,     ; crv: P256
    -2: h'5C099BEC31FAA581D14E208250D3FFDA
          9EC7F543043008BC84967A8D875B5D78',
    -3: h'539D57429FCB1C138DA29010A155DCA1
          4566A8F55AC2F1780810C49D4ED72D58',
  },

  -3: -9       ; Derived key algorithm: ESP256
}
```

The following is the same example encoded as CBOR:

```
h'a6013a000100000258206 0b6dfddd31659598ae5de49acb220d8704949e84d48
  4b68344340e2565337d2033a000100a320a401022001215820693 80fc1c3b096
  52134feefba61776f97af875ce46ca20252c4165102966ebc52258208b515831
  462ccb0bd55cba04bfd50da63faf18bd845433622daf97c06a10d0f121a40102
  20012158205c099bec31faa581d14e208250d3ffda9ec7f543043008bc84967a
  8d875b5d78225820539d57429fcb1c138da29010a155dca14566a8f55ac2f178
  0810c49d4ed72d582228'
```

5.2. COSE algorithms

This section defines COSE algorithm identifiers [RFC9052] for ARKG instances, and for signature algorithms combined with using a signing private key derived using ARKG.

Table 2 defines algorithm identifiers to represent ARKG instances.

Name	Value	Description
ARKG-P256	TBD (placeholder -65700)	The ARKG instance ARKG-P256 defined in Section 4.1.
ARKG-P384	TBD (placeholder -65701)	The ARKG instance ARKG-P384 defined in Section 4.2.
ARKG-P521	TBD (placeholder -65702)	The ARKG instance ARKG-P521 defined in Section 4.3.
ARKG-P256k	TBD (placeholder -65703)	The ARKG instance ARKG-P256k defined in Section 4.4.

Table 2: COSE algorithm identifiers for ARKG instances.

Table 3 defines algorithm identifiers to represent signing algorithms. These MAY be used to negotiate algorithm selection between a _digester_ and _signer_ as described in Section 2 of [I-D.lundberg-cose-split-algs], and in key representations exchanged between such _digesters_ and _signers_, but SHOULD NOT appear in COSE structures consumed by signature verifiers. COSE structures consumed by signature verifiers SHOULD instead use the corresponding algorithm identifier listed in the "verification algorithm" column.

Name	Value	Verification algorithm	Description
ESP256-ARKG	TBD	-9 (ESP256)	ESP256 [RFC9864] using private key derived by ARKG-P256 (Section 4.1).
ESP256-split-ARKG	TBD (placeholder -65539)	-9 (ESP256)	ESP256-split [I-D.lundberg-cose-split-algs] using private key derived by ARKG-P256 (Section 4.1).
ESP384-ARKG	TBD	-51 (ESP384)	ESP384 [RFC9864] using private key derived by ARKG-P384 (Section 4.2).
ESP384-split-ARKG	TBD	-51 (ESP384)	ESP384-split [I-D.lundberg-cose-split-algs] using private key derived by ARKG-P384 (Section 4.2).
ESP512-ARKG	TBD	-52 (ESP512)	ESP512 [RFC9864] using private key derived by ARKG-P521 (Section 4.3).
ESP512-split-ARKG	TBD	-52 (ESP512)	ESP512-split [I-D.lundberg-cose-split-algs] using private key derived by ARKG-P521 (Section 4.3).
ES256K-ARKG	TBD	-47 (ES256K)	ES256K [RFC8812] using private key derived by ARKG-P256k (Section 4.4).

Table 3: COSE algorithms for signing with an ARKG-derived key.

5.3. COSE signing arguments

This section defines ARKG-specific parameters for the COSE_Sign_Args structure [I-D.lundberg-cose-split-algs]. These consist of the parameters -1 and -2 respectively for the kh and ctx parameters of ARKG-Derive-Private-Key. Table 4 defines these algorithm parameters for COSE_Sign_args. kh and ctx are both REQUIRED for all the relevant alg values.

Name	Label	Type	Required?	Algorithm	Description
kh	-1	bstr	Required	ESP256-ARKG, ESP256-split-ARKG, ESP384-ARKG, ESP384-split-ARKG, ESP512-ARKG, ESP512-split-ARKG, ES256K-ARKG	kh argument to ARKG-Derive-Private-Key.
ctx	-2	bstr	Required	ESP256-ARKG, ESP256-split-ARKG, ESP384-ARKG, ESP384-split-ARKG, ESP512-ARKG, ESP512-split-ARKG, ES256K-ARKG	ctx argument to ARKG-Derive-Private-Key.

Table 4: Algorithm parameters for COSE_Sign_Args.

The following CDDL example conveys the kh and ctx arguments for signing data using the ESP256-split algorithm [I-D.lundberg-cose-split-algs] and a key derived using ARKG-P256:

```
{
  3: -65539,   ; alg: ESP256-split with ARKG-P256 (placeholder value)

               ; ARKG-P256 key handle
               ; (HMAC-SHA-256-128 followed by
                  SEC1 uncompressed ECDH public key)
  -1: h'27987995f184a44cfa548d104b0a461d
        0487fc739dbcdabc293ac5469221da91b220e04c681074ec4692a76ffacb9043de
          c2847ea9060fd42da267f66852e63589f0c00dc88f290d660c65a65a50c86361',

               ; info argument to ARKG-Derive-Private-Key
  -2: 'ARKG-P256.test vectors',
}
```

The following is the same example encoded as CBOR:

```
h'a3033a000100022058512798795f184a44cfa548d104b0a461d0487fc739dbc
  dabc293ac5469221da91b220e04c681074ec4692a76ffacb9043dec2847ea906
  0fd42da267f66852e63589f0c00dc88f290d660c65a65a50c86361215641524b
  472d503235362e7465737420766563746f7273'
```

6. Security Considerations

TODO

7. Privacy Considerations

TODO

8. IANA Considerations

8.1. COSE Key Types Registrations

This section registers the following values in the IANA "COSE Key Types" registry [IANA.cose].

* Name: ARKG-pub
  - Value: TBD (Placeholder -65537)
  - Description: ARKG public seed
  - Capabilities: [kty(-65537), pk_bl, pk_kem]
  - Reference: Section 5.1 of this document

8.2. COSE Key Type Parameters Registrations

This section registers the following values in the IANA "COSE Key Type Parameters" registry [IANA.cose].

* Key Type: TBD (ARKG-pub, placeholder -65537)
  - Name: pk_bl
  - Label: -1
  - CBOR Type: COSE_Key
  - Description: ARKG key blinding public key
  - Reference: Section 5.1 of this document
* Key Type: TBD (ARKG-pub, placeholder -65537)

- Name: pk_kem
- Label: -2
- CBOR Type: COSE_Key
- Description: ARKG key encapsulation public key
- Reference: Section 5.1 of this document

8.3. COSE Algorithms Registrations

This section registers the following values in the IANA "COSE Algorithms" registry [IANA.cose].

* Name: ARKG-P256
  - Value: TBD (placeholder -65700)
  - Description: ARKG using ECDH and additive blinding on secp256r1
  - Reference: Section 5.2 of this document
  - Recommended: TBD
* Name: ARKG-P384
  - Value: TBD (placeholder -65701)
  - Description: ARKG using ECDH and additive blinding on secp384r1
  - Reference: Section 5.2 of this document
  - Recommended: TBD
* Name: ARKG-P521
  - Value: TBD (placeholder -65702)
  - Description: ARKG using ECDH and additive blinding on secp521r1
  - Reference: Section 5.2 of this document
  - Recommended: TBD
* Name: ARKG-P256k
  - Value: TBD (placeholder -65703)

- Description: ARKG using ECDH and additive blinding on secp256k1
- Reference: Section 5.2 of this document
- Recommended: TBD

* Name: ESP256-ARKG
  - Value: TBD
  - Description: ESP256 using private key derived by ARKG-P256
  - Reference: [RFC9864], Section 5.2 of this document
  - Recommended: TBD
* Name: ESP256-split-ARKG
  - Value: TBD (placeholder -65539)
  - Description: ESP256-split using private key derived by ARKG-P256
  - Reference: [I-D.lundberg-cose-split-algs], Section 5.2 of this document
  - Recommended: TBD
* Name: ESP384-ARKG
  - Value: TBD
  - Description: ESP384 using private key derived by ARKG-P384
  - Reference: [RFC9864], Section 5.2 of this document
  - Recommended: TBD
* Name: ESP384-split-ARKG
  - Value: TBD
  - Description: ESP384-split using private key derived by ARKG-P384
  - Reference: [I-D.lundberg-cose-split-algs], Section 5.2 of this document

- Recommended: TBD

* Name: ESP512-ARKG

  - Value: TBD

  - Description: ESP512 using private key derived by ARKG-P521

  - Reference: [RFC9864], Section 5.2 of this document

  - Recommended: TBD

* Name: ESP512-split-ARKG

  - Value: TBD

  - Description: ESP512-split using private key derived by ARKG-P521

  - Reference: [I-D.lundberg-cose-split-algs], Section 5.2 of this document

  - Recommended: TBD

* Name: ESP256K-ARKG

  - Value: TBD

  - Description: ESP256K using private key derived by ARKG-P256k

  - Reference: [RFC8812], Section 5.2 of this document

  - Recommended: TBD

8.4. COSE Signing Arguments Algorithm Parameters Registrations

This section registers the following values in the IANA "COSE Signing Arguments Algorithm Parameters" registry [I-D.lundberg-cose-split-algs] (TODO):

* Name: kh

  - Label: -1

  - Type: bstr

  - Required: yes

- Algorithm: ESP256-ARKG, ESP256-split-ARKG, ESP384-ARKG, ESP384-split-ARKG, ESP512-ARKG, ESP512-split-ARKG, ES256K-ARKG
- Description: kh argument to ARKG-Derive-Private-Key.
- Capabilities: [alg(-65539, TBD)]
- Change Controller: IETF
- Reference: Section 5.3 of this document

* Name: ctx
  - Label: -2
  - Type: bstr
  - Required: yes
  - Algorithm: ESP256-ARKG, ESP256-split-ARKG, ESP384-ARKG, ESP384-split-ARKG, ESP512-ARKG, ESP512-split-ARKG, ES256K-ARKG
  - Description: ctx argument to ARKG-Derive-Private-Key.
  - Capabilities: [alg(-65539, TBD)]
  - Change Controller: IETF
  - Reference: Section 5.3 of this document

9. Design rationale

9.1. Using a MAC

The ARKG construction by Stebila et al. [ASIACCS_SteWil24] omits the MAC and instead encodes application context in the PRF labels, arguing that this leads to invalid keys/signatures in cases that would have a bad MAC. We choose to keep the MAC from the construction by Frymann et al. [CCS_FGKLMN20], but allow it to be omitted in case the chosen KEM already guarantees ciphertext integrity.

The reason for this is to ensure that the delegating party can distinguish key handles that belong to its ARKG seed. For example, this is important for applications using the W3C Web Authentication API [WebAuthn], which do not know beforehand which authenticators are connected and available. Instead, authentication requests may include references to several eligible authenticators, and the one to

use is chosen opportunistically by the WebAuthn client depending on which are available at the time. Consider using ARKG in such a scenario to sign some data with a derived private key: a user may have several authenticators and thus several ARKG seeds, so the signing request might include several well-formed ARKG key handles, but only one of them belongs to the ARKG seed of the authenticator that is currently connected. Without an integrity check, choosing the wrong key handle might cause the ARKG-Derive-Private-Key procedure to silently derive the wrong key instead of returning an explicit error, which would in turn lead to an invalid signature or similar final output. This would make it difficult or impossible to diagnose the root cause of the issue and present actionable user feedback. For this reason, we require the KEM to guarantee ciphertext integrity so that ARKG-Derive-Private-Key can fail early if the key handle belongs to a different ARKG seed.

It is straightforward to see that adding the MAC to the construction by Wilson does not weaken the security properties defined by Frymann et al. [CCS_FGKLMN20]: the construction by Frymann et al. can be reduced to the ARKG construction in this document by instantiating BL as described in Section 3.1 and KEM as described in Section 3.3. The use of hash_to_field in Section 3.1 corresponds to the KDF_1 parameter in [CCS_FGKLMN20], and the use of HMAC and HKDF in Section 3.2 corresponds to the MAC and KDF_2 parameters in [CCS_FGKLMN20]. Hence if one can break PK-unlinkability or SK-security of the ARKG construction in this document, one can also break the same property of the construction by Frymann et al.

10. Implementation Status

This section is to be removed from the specification by the RFC Editor before publication as an RFC.

There are currently two known implementations using features defined by this specification:

* wwWallet (https://github.com/wwWallet), an EU Digital Identity pilot project. wwWallet was entered into the "EUDI Wallet Prototypes" competition held by SprinD GmbH (https://www.sprind.org/en/actions/challenges/eudi-wallet-prototypes), and a branch of the wallet was submitted in the competition. The competition entry implements ARKG for efficiently generating single-use hardware-bound holder binding keys.

  The implementation (https://github.com/gunet/funke-s3a-wallet-frontend/blob/stage-3/src/services/keystore.ts) uses the COSE_Key_Ref data structure defined in version 01 of

[I-D.lundberg-cose-split-algs] in order to send ARKG inputs to a WebAuthn authenticator, and uses the placeholder value for ESP256-split-ARKG defined in Section 5.2 to negotiate creation and usage of ARKG-derived keys for signing operations. Work to update the implementation to instead use COSE_Sign_Args as defined in version 05 of [I-D.lundberg-cose-split-algs] is ongoing.

* Yubico (https://www.yubico.com/), a hardware security key vendor, has produced limited-availability prototypes of their YubiKey product with an ARKG implementation interoperable with wwWallet. The YubiKey implementation uses the COSE_Sign_Args data structure defined in version 05 of [I-D.lundberg-cose-split-algs] to receive ARKG inputs from a WebAuthn Relying Party, and uses the placeholder value for ESP256-split-ARKG defined in Section 5.2 to negotiate creation and usage of ARKG-derived keys for signing operations.

Table 5 summarizes implementation status for individual features.

Feature	Implementations
ARKG-P256	wwWallet, Yubico
ARKG-P384	-
ARKG-P521	-
ARKG-P256k	-
ESP256-ARKG	-
ESP256-split-ARKG	wwWallet, Yubico
ESP384-ARKG	-
ESP384-split-ARKG	-
ESP512-ARKG	-
ESP512-split-ARKG	-
ES256K-ARKG	-
COSE_Sign_Args	wwWallet, Yubico

Table 5: Implementation status of individual features.

10.1. Related Internet-Drafts

Parts of this specification depend upon definitions from [I-D.lundberg-cose-split-algs]:

* The algorithm identifiers ESP256-split-ARKG, ESP384-split-ARKG and ESP512-split-ARKG defined in Section 5.2 depend respectively on the algorithm identifiers ESP256-split, ESP384-split and ESP512-split defined in [I-D.lundberg-cose-split-algs].

* The ARKG-specific COSE_Sign_Args parameter definitions in Section 5.3 depend on [I-D.lundberg-cose-split-algs] for the definition of the COSE_Sign_Args structure.

10.2. Future Work

Hierarchical Deterministic Keys (HDK) [I-D.dijkhuis-cfrg-hdkeys] is a possible application of ARKG which has identified a limitation in the present construction, as discussed in HDK GitHub issue #94 (https://github.com/sander/hierarchical-deterministic-keys/issues/94). ARKG can be used recursively - for example, ARKG-P256 can be used to derive P-256 keys that are themselves used as ARKG seeds - but then requires one invocation of ARKG-Derive-Private-Key per layer of recursion in order to derive higher-layer private keys. This can be an issue if the base ARKG private seed is hardware-bound, since it would require multiple calls to the secure hardware device, especially if each of those calls requires a user gesture for authorization. Therefore a modified ARKG construction has been proposed, along with a draft for a potential security proof (https://github.com/Yubico/arkg-rfc/blob/pqarkg-h/pqarkg-h-security/pqarkg-h.pdf), which enables multiple layers of recursive ARKG to be condensed into a single invocation of ARKG-Derive-Private-Key. We would have liked to use this modified construction in this specification, but have not been able to get the potential proof appropriately peer reviewed in a timely manner, so prototypes have moved forward with the present construction. The modified construction may be revisited in the future if there is demand for applications, in which case new algorithm identifiers and such can be defined for the modified construction.

11. References

11.1. Normative References

[I-D.lundberg-cose-split-algs]
Lundberg, E. and M. B. Jones, "Split signing algorithms for COSE", Work in Progress, Internet-Draft, draft-lundberg-cose-two-party-signing-algs-05, 23 February 2026, <https://datatracker.ietf.org/doc/html/draft-lundberg-cose-two-party-signing-algs-05>.

[IANA.cose]
IANA, "CBOR Object Signing and Encryption (COSE)", <https://www.iana.org/assignments/cose>.

[RFC2104] Krawczyk, H., Bellare, M., and R. Canetti, "HMAC: Keyed-Hashing for Message Authentication", RFC 2104, DOI 10.17487/RFC2104, February 1997, <https://www.rfc-editor.org/rfc/rfc2104>.

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

[RFC4949] Shirey, R., "Internet Security Glossary, Version 2", FYI 36, RFC 4949, DOI 10.17487/RFC4949, August 2007, <https://www.rfc-editor.org/rfc/rfc4949>.

[RFC5869] Krawczyk, H. and P. Eronen, "HMAC-based Extract-and-Expand Key Derivation Function (HKDF)", RFC 5869, DOI 10.17487/RFC5869, May 2010, <https://www.rfc-editor.org/rfc/rfc5869>.

[RFC6090] McGrew, D., Igoe, K., and M. Salter, "Fundamental Elliptic Curve Cryptography Algorithms", RFC 6090, DOI 10.17487/RFC6090, February 2011, <https://www.rfc-editor.org/rfc/rfc6090>.

[RFC7748] Langley, A., Hamburg, M., and S. Turner, "Elliptic Curves for Security", RFC 7748, DOI 10.17487/RFC7748, January 2016, <https://www.rfc-editor.org/rfc/rfc7748>.

[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, <https://www.rfc-editor.org/rfc/rfc8017>.

[RFC8032] Josefsson, S. and I. Liusvaara, "Edwards-Curve Digital Signature Algorithm (EdDSA)", RFC 8032, DOI 10.17487/RFC8032, January 2017, <https://www.rfc-editor.org/rfc/rfc8032>.

[RFC8174] Leiba, B., "Ambiguity of Uppercase vs Lowercase in RFC 2119 Key Words", BCP 14, RFC 8174, DOI 10.17487/RFC8174, May 2017, <https://www.rfc-editor.org/rfc/rfc8174>.

[RFC8610] Birkholz, H., Vigano, C., and C. Bormann, "Concise Data Definition Language (CDDL): A Notational Convention to Express Concise Binary Object Representation (CBOR) and JSON Data Structures", RFC 8610, DOI 10.17487/RFC8610, June 2019, <https://www.rfc-editor.org/rfc/rfc8610>.

[RFC8812] Jones, M., "CBOR Object Signing and Encryption (COSE) and JSON Object Signing and Encryption (JOSE) Registrations for Web Authentication (WebAuthn) Algorithms", RFC 8812, DOI 10.17487/RFC8812, August 2020, <https://www.rfc-editor.org/rfc/rfc8812>.

[RFC9052] Schaad, J., "CBOR Object Signing and Encryption (COSE): Structures and Process", STD 96, RFC 9052, DOI 10.17487/RFC9052, August 2022, <https://www.rfc-editor.org/rfc/rfc9052>.

[RFC9380] Faz-Hernandez, A., Scott, S., Sullivan, N., Wahby, R. S., and C. A. Wood, "Hashing to Elliptic Curves", RFC 9380, DOI 10.17487/RFC9380, August 2023, <https://www.rfc-editor.org/rfc/rfc9380>.

[RFC9864] Jones, M.B. and O. Steele, "Fully-Specified Algorithms for JSON Object Signing and Encryption (JOSE) and CBOR Object Signing and Encryption (COSE)", RFC 9864, DOI 10.17487/RFC9864, October 2025, <https://www.rfc-editor.org/rfc/rfc9864>.

[SEC1] Certicom Research, "SEC 1: Elliptic Curve Cryptography", 2009, <http://www.secg.org/sec1-v2.pdf>.

[SEC2] Certicom Research, "SEC 2: Recommended Elliptic Curve Domain Parameters", 2010, <http://www.secg.org/sec2-v2.pdf>.

11.2. Informative References

[AC_BreCleFis24] Brendel, J., Clermont, S., and M. Fischlin, "Post-Quantum Asynchronous Remote Key Generation for FIDO2 Account Recovery. ASIACRYPT ’24", 2023, <https://eprint.iacr.org/2023/1275>.

[ASIACCS_SteWil24] Stebila, D. and S. M. Wilson, "Quantum-Safe Account Recovery for WebAuthn. ASIACCS ’24", 2023, <https://eprint.iacr.org/2024/678>.

[BIP32] Wuille, P., "BIP 32 Hierarchical Deterministic Wallets", 2012, <https://github.com/bitcoin/bips/blob/master/bip-0032.mediawiki>.

[CCS_FGKLMN20] Frymann, N., Gardham, D., Kiefer, F., Lundberg, E., Manulis, M., and D. Nilsson, "Asynchronous Remote Key Generation: An Analysis of Yubico’s Proposal for W3C WebAuthn. CCS ’20: Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security", 2020, <https://eprint.iacr.org/2020/1004>.

[EUROSP_FryGarMan23]
Frymann, N., Gardham, D., and M. Manulis, "Asynchronous Remote Key Generation for Post-Quantum Cryptosystems from Lattices. 2023 IEEE 8th European Symposium on Security and Privacy", 2023, <https://eprint.iacr.org/2023/419>.

[I-D.dijkhuis-cfrg-hdkeys]
Dijkhuis, S., "Hierarchical Deterministic Keys", Work in Progress, Internet-Draft, draft-dijkhuis-cfrg-hdkeys-06, 19 January 2025, <https://datatracker.ietf.org/doc/html/draft-dijkhuis-cfrg-hdkeys-06>.

[Shoup] Shoup, V., "A Proposal for an ISO Standard for Public Key Encryption (version 2.0)", 2001, <https://www.shoup.net/papers/iso-2.pdf>.

[WebAuthn-Recovery]
Lundberg, E. and D. Nilsson, "WebAuthn recovery extension: Asynchronous delegated key generation without shared secrets. GitHub", 2019, <https://github.com/Yubico/webauthn-recovery-extension>.

[Wilson2023]
Wilson, S. M., "Post-Quantum Account Recovery for Passwordless Authentication. Master’s thesis", 2023, <http://hdl.handle.net/10012/19316>.

Appendix A. Test Vectors

This section lists test vectors for validating implementations.

Test vectors are listed in CDDL [RFC8610] syntax using variable names defined in Section 2 and Section 3. Elliptic curve points are encoded using the Elliptic-Curve-Point-to-Octet-String procedure defined in section 2.3.3 of [SEC1], without point compression.

A.1. ARKG-P256

```
   ; Inputs:
   ctx         = 'ARKG-P256.test vectors'
   ikm_bl      = h'000102030405060708090a0b0c0d0e0f101112131415161718191a1b1c1d1e1f'
   ikm_kem     = h'202122232425262728292a2b2c2d2e2f303132333435363738393a3b3c3d3e3f'
   ikm         = h'404142434445464748494a4b4c4d4e4f505152535455565758595a5b5c5d5e5f'

   ; Derive-Seed:
   DST_bl_sk   = h'41524b472d424c2d45432d4b472e41524b472d50323536'
   DST_kem_sk  = h'41524b472d4b454d2d454344482d4b472e41524b472d454344482e41524b472d503235
36'
   pk_bl       = h'046d3bdf31d0db48988f16d47048fdd24123cd286e42d0512daa9f726b4ecf18df
                    65ed42169c69675f936ff7de5f9bd93adbc8ea73036b16e8d90adbfabdaddba7'
   pk_kem      = h'04c38bbdd7286196733fa177e43b73cfd3d6d72cd11cc0bb2c9236cf85a42dcff5
                    dfa339c1e07dfcdfda8d7be2a5a3c7382991f387dfe332b1dd8da6e0622cfb35'
   sk_bl       = 0xd959500a78ccf850ce46c80a8c5043c9a2e33844232b3829df37d05b3069f455
   sk_kem      = 0x74e0a4cd81ca2d24246ff75bfd6d4fb7f9dfc938372627feb2c2348f8b1493b5

   ; Derive-Public-Key:
   ctx_bl      = h'41524b472d4465726976652d4b65792d424c2e1641524b472d503235362e7465737420
766563746f7273'
   ctx_kem     = h'41524b472d4465726976652d4b65792d4b454d2e1641524b472d503235362e74657374
20766563746f7273'
   ctx_sub     = h'41524b472d4b454d2d484d41432e41524b472d454344482e41524b472d503235364152
4b472d4465726976652d4b65792d4b454d2e1641524b472d503235362e7465737420766563746f7273'
   DST_kem_sk  = h'41524b472d4b454d2d454344482d4b472e41524b472d454344482e41524b472d503235
36'
   k_prime     = h'fa027ebc49603a2a41052479f6e9f6d046175df2f00cecb403f53ffcd1cc698f'
   c_prime     = h'0487fc739dbcdabc293ac5469221da91b220e04c681074ec4692a76ffacb9043dec284
7ea9060fd42da267f66852e63589f0c00dc88f290d660c65a65a50c86361'
   info_mk     = h'41524b472d4b454d2d484d41432d6d61632e41524b472d454344482e41524b472d5032
353641524b472d4465726976652d4b65792d4b454d2e1641524b472d503235362e7465737420766563746f727
3'
   mk          = h'796c615d19ca0044df0a22d64ba8d5367dca18da32b871a3e255db0af7eb53c9'
   t           = h'27987995f184a44cfa548d104b0a461d'
   info_k      = h'41524b472d4b454d2d484d41432d7368617265642e41524b472d454344482e41524b47
2d5032353641524b472d4465726976652d4b65792d4b454d2e1641524b472d503235362e74657374207665637
46f7273'
   k           = h'cf5e8ddbb8078a6a0144d4412f22f89407ecee30ec128ce07836af9fc51c05d0'
   c           = h'27987995f184a44cfa548d104b0a461d0487fc739dbcdabc293ac5469221da91b220e0
4c681074ec4692a76ffacb9043dec2847ea9060fd42da267f66852e63589f0c00dc88f290d660c65a65a50c86
361'
   ikm_tau     = h'cf5e8ddbb8078a6a0144d4412f22f89407ecee30ec128ce07836af9fc51c05d0'
   DST_tau     = h'41524b472d424c2d45432e41524b472d5032353641524b472d4465726976652d4b6579
2d424c2e1641524b472d503235362e7465737420766563746f7273'
   tau         = 0x9e042fde2e12c1f4002054a8feac60088cc893b4838423c26a20af686c8c16e3
   pk_prime    = h'04572a111ce5cfd2a67d56a0f7c684184b16ccd212490dc9c5b579df749647d107
                    dac2a1b197cc10d2376559ad6df6bc107318d5cfb90def9f4a1f5347e086c2cd'
   kh          = h'27987995f184a44cfa548d104b0a461d0487fc739dbcdabc293ac5469221da91b220e0
4c681074ec4692a76ffacb9043dec2847ea9060fd42da267f66852e63589f0c00dc88f290d660c65a65a50c86
361'

   ; Derive-Private-Key:
   sk_prime    = 0x775d7fe9a6dfba43ce671cb38afca3d272c4d14aff97bd67559eb500a092e5e7
```

```
   ; Inputs:
   ctx         = 'ARKG-P256.test vectors'
   ikm_bl      = h'000102030405060708090a0b0c0d0e0f101112131415161718191a1b1c1d1e1f'
   ikm_kem     = h'202122232425262728292a2b2c2d2e2f303132333435363738393a3b3c3d3e3f'
   ikm         = h'a0a1a2a3a4a5a6a7a8a9aaabacadaeafb0b1b2b3b4b5b6b7b8b9babbbcbdbebf'

   ; Derive-Seed:
   DST_bl_sk   = h'41524b472d424c2d45432d4b472e41524b472d50323536'
   DST_kem_sk  = h'41524b472d4b454d2d454344482d4b472e41524b472d454344482e41524b472d503235
36'
   pk_bl       = h'046d3bdf31d0db48988f16d47048fdd24123cd286e42d0512daa9f726b4ecf18df
                    65ed42169c69675f936ff7de5f9bd93adbc8ea73036b16e8d90adbfabdaddba7'
   pk_kem      = h'04c38bbdd7286196733fa177e43b73cfd3d6d72cd11cc0bb2c9236cf85a42dcff5
                    dfa339c1e07dfcdfda8d7be2a5a3c7382991f387dfe332b1dd8da6e0622cfb35'
   sk_bl       = 0xd959500a78ccf850ce46c80a8c5043c9a2e33844232b3829df37d05b3069f455
   sk_kem      = 0x74e0a4cd81ca2d24246ff75bfd6d4fb7f9dfc938372627feb2c2348f8b1493b5

   ; Derive-Public-Key:
   ctx_bl      = h'41524b472d4465726976652d4b65792d424c2e1641524b472d503235362e7465737420
766563746f7273'
   ctx_kem     = h'41524b472d4465726976652d4b65792d4b454d2e1641524b472d503235362e74657374
20766563746f7273'
   ctx_sub     = h'41524b472d4b454d2d484d41432e41524b472d454344482e41524b472d503235364152
4b472d4465726976652d4b65792d4b454d2e1641524b472d503235362e7465737420766563746f7273'
   DST_kem_sk  = h'41524b472d4b454d2d454344482d4b472e41524b472d454344482e41524b472d503235
36'
   k_prime     = h'38c79546fc4a144ae2068ff0b515fc9af032b8255a78a829e71be47676a63117'
   c_prime     = h'0457fd1e438280c127dd55a6138d1baf0a35e3e9671f7e42d8345f47374afa83247a07
8fa2196cd69497aed59ef92c05cb6b03d306ec24f2f4ff2db09cd95d1b11'
   info_mk     = h'41524b472d4b454d2d484d41432d6d61632e41524b472d454344482e41524b472d5032
353641524b472d4465726976652d4b65792d4b454d2e1641524b472d503235362e7465737420766563746f727
3'
   mk          = h'0806abac4c1d205c3a8826cd178fbf7f91741268e3ca73634035efd76085d2a9'
   t           = h'b7507a82771776fbac41a18d94e19a7e'
   info_k      = h'41524b472d4b454d2d484d41432d7368617265642e41524b472d454344482e41524b47
2d5032353641524b472d4465726976652d4b65792d4b454d2e1641524b472d503235362e74657374207665637
46f7273'
   k           = h'dcdd95c742ddf25b8a95f3d76326cb3593b7860bb3e04c5e5b25cc15ce1e5c84'
   c           = h'b7507a82771776fbac41a18d94e19a7e0457fd1e438280c127dd55a6138d1baf0a35e3
e9671f7e42d8345f47374afa83247a078fa2196cd69497aed59ef92c05cb6b03d306ec24f2f4ff2db09cd95d1
b11'
   ikm_tau     = h'dcdd95c742ddf25b8a95f3d76326cb3593b7860bb3e04c5e5b25cc15ce1e5c84'
   DST_tau     = h'41524b472d424c2d45432e41524b472d5032353641524b472d4465726976652d4b6579
2d424c2e1641524b472d503235362e7465737420766563746f7273'
   tau         = 0x88cf9464b041a52cf2b837281afc67302ec9cb32da1fe515381b79c0d0c92322
   pk_prime    = h'04ea7d962c9f44ffe8b18f1058a471f394ef81b674948eefc1865b5c021cf858f5
                    77f9632b84220e4a1444a20b9430b86731c37e4dcb285eda38d76bf758918d86'
   kh          = h'b7507a82771776fbac41a18d94e19a7e0457fd1e438280c127dd55a6138d1baf0a35e3
e9671f7e42d8345f47374afa83247a078fa2196cd69497aed59ef92c05cb6b03d306ec24f2f4ff2db09cd95d1
b11'

   ; Derive-Private-Key:
   sk_prime    = 0x6228e470290e9d7cc0feff32a74caafa14c608c956337eba23997f5904cff226
```

```
   ; Inputs:
   ctx         = 'ARKG-P256.test vectors.0'
   ikm_bl      = h'000102030405060708090a0b0c0d0e0f101112131415161718191a1b1c1d1e1f'
   ikm_kem     = h'202122232425262728292a2b2c2d2e2f303132333435363738393a3b3c3d3e3f'
   ikm         = h'404142434445464748494a4b4c4d4e4f505152535455565758595a5b5c5d5e5f'

   ; Derive-Seed:
   DST_bl_sk   = h'41524b472d424c2d45432d4b472e41524b472d50323536'
   DST_kem_sk  = h'41524b472d4b454d2d454344482d4b472e41524b472d454344482e41524b472d503235
36'
   pk_bl       = h'046d3bdf31d0db48988f16d47048fdd24123cd286e42d0512daa9f726b4ecf18df
                    65ed42169c69675f936ff7de5f9bd93adbc8ea73036b16e8d90adbfabdaddba7'
   pk_kem      = h'04c38bbdd7286196733fa177e43b73cfd3d6d72cd11cc0bb2c9236cf85a42dcff5
                    dfa339c1e07dfcdfda8d7be2a5a3c7382991f387dfe332b1dd8da6e0622cfb35'
   sk_bl       = 0xd959500a78ccf850ce46c80a8c5043c9a2e33844232b3829df37d05b3069f455
   sk_kem      = 0x74e0a4cd81ca2d24246ff75bfd6d4fb7f9dfc938372627feb2c2348f8b1493b5

   ; Derive-Public-Key:
   ctx_bl      = h'41524b472d4465726976652d4b65792d424c2e1841524b472d503235362e7465737420
766563746f72732e30'
   ctx_kem     = h'41524b472d4465726976652d4b65792d4b454d2e1841524b472d503235362e74657374
20766563746f72732e30'
   ctx_sub     = h'41524b472d4b454d2d484d41432e41524b472d454344482e41524b472d503235364152
4b472d4465726976652d4b65792d4b454d2e1841524b472d503235362e7465737420766563746f72732e30'
   DST_kem_sk  = h'41524b472d4b454d2d454344482d4b472e41524b472d454344482e41524b472d503235
36'
   k_prime     = h'fa027ebc49603a2a41052479f6e9f6d046175df2f00cecb403f53ffcd1cc698f'
   c_prime     = h'0487fc739dbcdabc293ac5469221da91b220e04c681074ec4692a76ffacb9043dec284
7ea9060fd42da267f66852e63589f0c00dc88f290d660c65a65a50c86361'
   info_mk     = h'41524b472d4b454d2d484d41432d6d61632e41524b472d454344482e41524b472d5032
353641524b472d4465726976652d4b65792d4b454d2e1841524b472d503235362e7465737420766563746f727
32e30'
   mk          = h'd342e45f224a7278f11cf1468922c8879f4529125181d4159e4bf9ee69842f04'
   t           = h'81c4e65b552e52350b49864b98b87d51'
   info_k      = h'41524b472d4b454d2d484d41432d7368617265642e41524b472d454344482e41524b47
2d5032353641524b472d4465726976652d4b65792d4b454d2e1841524b472d503235362e74657374207665637
46f72732e30'
   k           = h'cde7e271f8da72e5fd2557de362420ddb170dce520362131670eb1080823a113'
   c           = h'81c4e65b552e52350b49864b98b87d510487fc739dbcdabc293ac5469221da91b220e0
4c681074ec4692a76ffacb9043dec2847ea9060fd42da267f66852e63589f0c00dc88f290d660c65a65a50c86
361'
   ikm_tau     = h'cde7e271f8da72e5fd2557de362420ddb170dce520362131670eb1080823a113'
   DST_tau     = h'41524b472d424c2d45432e41524b472d5032353641524b472d4465726976652d4b6579
2d424c2e1841524b472d503235362e7465737420766563746f72732e30'
   tau         = 0x513ea417b6cdc3536178fa81da36b4e5ecdc142c2d46a52e05257f21794e3789
   pk_prime    = h'04b79b65d6bbb419ff97006a1bd52e3f4ad53042173992423e06e52987a037cb61
                    dd82b126b162e4e7e8dc5c9fd86e82769d402a1968c7c547ef53ae4f96e10b0e'
   kh          = h'81c4e65b552e52350b49864b98b87d510487fc739dbcdabc293ac5469221da91b220e0
4c681074ec4692a76ffacb9043dec2847ea9060fd42da267f66852e63589f0c00dc88f290d660c65a65a50c86
361'

   ; Derive-Private-Key:
   sk_prime    = 0x2a97f4232f9abba32fbfc28c6686f8afd2d851c2a95a3ed2f0a384b9ad55068d
```

A.2. Other instances

TODO

Acknowledgements

ARKG was first proposed under this name by Frymann et al. [CCS_FGKLMN20], who analyzed a proposed extension to W3C Web Authentication by Lundberg and Nilsson [WebAuthn-Recovery], which was in turn inspired by a similar construction by Wuille [BIP32] used to create privacy-preserving Bitcoin addresses. Frymann et al. [CCS_FGKLMN20] generalized the constructions by Lundberg, Nilsson and Wuille from elliptic curves to any discrete logarithm (DL) problem, and also proved the security of arbitrary asymmetric protocols composed with ARKG. Further generalizations to include quantum-resistant instantiations were developed independently by Brendel et al. [AC_BreCleFis24], Frymann et al. [EUROSP_FryGarMan23] and Wilson [Wilson2023].

This document adopts the construction proposed by Wilson [Wilson2023], modified by the inclusion of a MAC in the key handles as done in the original construction by Frymann et al. [CCS_FGKLMN20]. The construction by Wilson [Wilson2023] was later refined by Stebila et al. [ASIACCS_SteWil24], but this revision replaced the "key blinding scheme" component with a "key-blinding signature scheme" component which is not one-for-one compatible with the construction in the present revision of this specification.

The authors would like to thank all of these authors for their research and development work that led to the creation of this document.

Document History

-10

* Fixed tau misspelled as tau’ in body of BL-Blind-Private-Key in section "Using elliptic curve addition for key blinding".

* Fixed definitions and references misspelling ESP512 as ESP521.

* Minor editorial clarifications.

* Updated informative references to research papers and changed citation style for the same.

* Made "Acknowledgements" and "Document History" sections un-numbered.

* Added Implementation Status section.

-09

* Fixed hash_to_field argument ikm_tau misnamed as tau in section "Using elliptic curve addition for key blinding".

* Updated to match draft -02 of [I-D.lundberg-cose-split-algs].

  - COSE algorithm identifier definitions for ARKG instances moved from section "COSE key type: ARKG public seed" to new section "COSE algorithms".

  - Added COSE algorithm identifier definitions for signature algorithms with key derived using ARKG.

  - COSE key type Ref-ARKG-Derived deleted in favour of new COSE_Sign_Args algorithm parameters.

  - Section "COSE key reference type: ARKG derived private key" replaced with "COSE signing arguments".

  - Added section "COSE Signing Arguments Algorithm Parameters Registrations".

-08

* Fixed incorrectly swapped ikm_bl and ikm_kem arguments in ARKG-Derive-Seed definition.

* Extracted parameter function BL-PRF and modified signatures of BL-Blind-Public-Key and BL-Blind-Private-Key accordingly. This is an editorial refactorization; overall operation of concrete ARKG instances is unchanged.

* Removed three redundant sets of ARKG-P256 test vectors.

* Added intermediate values to ARKG-P256 test vectors.

* Changed second set of ARKG-P256 test vectors to use a 32-byte ikm instead of h’00’.

* Clarified in sections "Using HMAC to adapt a KEM without ciphertext integrity", "Using ECDH as the KEM" and "Using X25519 or X448 as the KEM" that ctx_sub is intentionally ignored in those instances.

-07

* Fixed hash_to_field DST in Sub-Kem-Derive-Key-Pair in section "Using ECDH as the KEM" to agree with test vectors.

-06

* Changed DST construction in section "Using ECDH as the KEM" to include the "ARKG-ECDH." prefix everywhere in the formula. Previously the prefix was added in the argument to the "Using HMAC to adapt a KEM without ciphertext integrity" formula but not in the Sub-Kem functions defined in "Using ECDH as the KEM".

-05

* Deleted concrete instances ARKG-curve25519ADD-X25519, ARKG-curve448ADD-X448, ARKG-edwards25519ADD-X25519 and ARKG-edwards448ADD-X448 since implementations with a non-prime order generator, including EdDSA, are incompatible with the additive blinding scheme defined in section "Using elliptic curve addition for key blinding".

* Remodeled procedures to be fully deterministic:

  - BL-Generate-Keypair() replaced with BL-Derive-Key-Pair(ikm).

  - KEM-Generate-Keypair() replaced with KEM-Derive-Key-Pair(ikm).

  - ARKG-Generate-Seed() replaced with ARKG-Derive-Seed(ikm_bl, ikm_kem).

  - Parameter ikm added to ARKG-Derive-Public-Key.

  - Instance parameter hash-to-crv-suite added to generic formula "Using ECDH as the KEM", affecting concrete instances ARKG-P256ADD-ECDH, ARKG-P384ADD-ECDH, ARKG-P521ADD-ECDH and ARKG-P256kADD-ECDH.

  - Section "Deterministic key generation" deleted.

* Flipped order of (pk_bl, pk_kem) and (sk_bl, sk_kem) parameter and return value tuples for consistent ordering between BL and KEM throughout document.

* info parameter renamed to ctx.

* ctx length limited to at most 64 bytes.

* Encoding of ctx in ARKG-Derive-Public-Key and ARKG-Derive-Private-Key now embeds the length of ctx.

* Renamed concrete instances and corresponding DST_ext values:

- ARKG-P256ADD-ECDH to ARKG-P256
- ARKG-P384ADD-ECDH to ARKG-P384
- ARKG-P521ADD-ECDH to ARKG-P521
- ARKG-P256kADD-ECDH to ARKG-P256k

* Added ARKG-P256 test vectors.

-04

* Extracted COSE_Key_Ref definition and COSE algorithm registrations to draft-lundberg-cose-two-party-signing-algs.
* Redefined alg (3) parameter and added dkalg (-3) in ARKG-pub COSE_Key.
* Defined alg (3) and inst (-3) parameters of Ref-ARKG-derived COSE key type.

-03

* Renamed section "Using HMAC to adapt a KEM without {integrity protection => ciphertext integrity}".
* Fixed info argument to HMAC in section "Using HMAC to adapt a KEM without ciphertext integrity".
* Added reference to Shoup for definition of key encapsulation mechanism.
* Added CDDL definition of COSE_Key_Ref.
* Editorial fixes to references.
* Renamed proposed COSE Key Types.

-02

* Rewritten introduction.
* Renamed ARKG-Derive-Secret-Key to ARKG-Derive-Private-Key.
* Overhauled EC instantiations to use hash_to_field and account for non-prime order curve key generation.
* Eliminated top-level MAC and KDF instance parameters.

* Added info parameter to instance parameter functions.

* Added requirement of KEM ciphertext integrity and generic formula for augmenting any KEM using HMAC.

* Added curve/edwards25519/448 instances.

* Added proposal for COSE bindings and key reference types.

-01

* Editorial fixes to formatting and references

-00

* Initial Version

Contributors

Dain Nilsson
Yubico

Peter Altmann
Agency for Digital Government
Sweden

Michael B. Jones
Self-Issued Consulting
United States
URI: https://self-issued.info/

Sander Dijkhuis
Cleverbase
Netherlands

Authors' Addresses

Emil Lundberg (editor)
Yubico
G辰vlegatan 22
Stockholm
Sweden
Email: emil@emlun.se

John Bradley
Yubico
Email: ve7jtb@ve7jtb.com