The Asynchronous Remote Key Generation (ARKG) algorithm

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

This Internet-Draft is submitted in full conformance with the provisions of BCP 78 and BCP 79.

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF). Note that other groups may also distribute working documents as Internet-Drafts. The list of current Internet-Drafts is at https://datatracker.ietf.org/drafts/current/.
Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

This Internet-Draft will expire on 2 September 2024.

Copyright Notice

Copyright (c) 2024 IETF Trust and the persons identified as the document authors. All rights reserved.

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

Table of Contents

1. Introduction
   1.1. Requirements Language
   1.2. The Asynchronous Remote Key Generation (ARKG) algorithm
       1.2.1. Notation
       1.2.2. Instance parameters
       1.2.3. The function ARKG-Generate-Seed
       1.2.4. The function ARKG-Derive-Public-Key
       1.2.5. The function ARKG-Derive-Secret-Key
   1.3. Generic ARKG instantiations
       1.3.1. Using elliptic curve arithmetic for key blinding
       1.3.2. Using ECDH as the KEM
       1.3.3. Using both elliptic curve arithmetic for key blinding and ECDH as the KEM
       1.3.4. Using HMAC as the MAC
       1.3.5. Using HKDF as the KDF
   1.4. Concrete ARKG instantiations
       1.4.1. ARKG-P256-ECDH-P256-HMAC-SHA256-HKDF-SHA256
   1.5. COSE bindings
2. Security Considerations
3. Privacy Considerations
4. IANA Considerations
5. Design rationale
   5.1. Using a MAC
   5.2. Implementation Status
6. References
1. Introduction

Asymmetric cryptography, also called public key cryptography, is a fundamental component of much of modern information security. However, even the flexibility of asymmetric cryptosystems is not always enough for all applications. For the sake of privacy and forward secrecy it may be necessary to frequently generate new keys, but it is not always feasible for the holder of the private keys to be available whenever a new key pair is needed. For example, this is often the case when using a hardware security device to hold private keys, where the device may be detached or locked at the time a new key pair is needed.

The Asynchronous Remote Key Generation (ARKG) algorithm enables the holder of private keys to delegate generation of public keys without giving access to the corresponding private keys. This enables a public key consumer to autonomously generate public keys whenever one is needed, while the private key holder can later derive the corresponding private key using a "key handle" generated along with the public key.

The algorithm consists of three procedures: (1) the delegating party generates a seed pair and emits the public seed to a subordinate party, (2) the subordinate party uses the public seed to generate a public key and a key handle on behalf of the delegating party, and (3) the delegating party uses the key handle and the private seed to derive the private key corresponding to the public key generated by procedure (2). Procedure (1) is performed once, and procedures (2) and (3) may be repeated any number of times with the same seed pair. The required cryptographic primitives are a public key blinding scheme, a key encapsulation mechanism (KEM), a key derivation function (KDF) and a message authentication code (MAC) scheme. Both conventional primitives and quantum-resistant alternatives exist that meet these requirements. [Wilson]

Some motivating use cases of ARKG include:

*Efficient single-use signing keys. The European Union has proposed a digital identity system which, in order to protect users' privacy, needs a unique key pair for each authentication
signature. In online usage the system could relatively easily create a key on demand, submit it to a certification authority to have a single-use certificate issued for that key, and then submit that certificate with an authentication signature to a third party to access a service.

However, the proposed system also includes offline use cases: A user might for example need to use the system in a location with poor or no internet connectivity to present a digital driver's license or authorize a payment. For this, the system may need to pre-emptively generate a large amount of single-use certificates to be used offline.

One candidate implementation under evaluation to provide signing and key management for this system is the W3C Web Authentication API [WebAuthn] (WebAuthn), which requires a user gesture whenever a WebAuthn operation is invoked. A WebAuthn-based implementation of the proposed digital identity system could use ARKG to pre-emptively generate key pairs for offline use without the need to prompt for a user gesture for each key pair generated.

*Enhanced forward secrecy for encrypted messaging. For example, section 8.5.4 of RFC 9052 defines COSE representations for encrypted messages and 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." Applications could work around this limitation by exchanging a large number of keys in advance, but that number limits how many messages can be sent before another such exchange is needed. This also requires the sender to allocate storage space for the keys, which may be challenging to support in constrained hardware.

ARKG could enable the sender to generate ephemeral recipient public keys on demand. This may enhance forward secrecy if the sender keeps the ARKG public seed secret, since each recipient key pair is used to encrypt only one message.

*Generating additional public keys as backup keys. For example, the W3C Web Authentication API [WebAuthn] (WebAuthn) generates a new key pair for each account on each web site. This makes it difficult for users to set up a backup authenticator, because each time a key pair is created for the primary authenticator, another key pair also needs to be created for the backup authenticator, which may be stored in a safe but inconvenient location.

ARKG could enable the primary authenticator to also generate a public key for a paired backup authenticator whenever it
generates a key pair for itself, allowing the user to set up the pairing once and then leave the backup authenticator in safe storage until the primary authenticator is lost.

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. 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.

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

1.2.1. Notation

The following notation is used throughout this document:

*The symbol || represents octet string concatenation.

*When literal text strings are to be interpreted as octet strings, they are encoded using UTF-8.

*Elliptic curve operations are written in multiplicative notation: *
  * denotes point multiplication, i.e., the curve group operation;
  ^ denotes point exponentiation, i.e., repeated point multiplication; and + denotes scalar addition modulo the curve order.

*Random(min_inc, max_exc) represents a cryptographically secure random integer greater than or equal to min_inc and strictly less than max_exc.
1.2.2. Instance parameters

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

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

- Function BL-Generate-Keypair() \(\rightarrow\) (pk, sk): Generate a blinding key pair.
  
  No input.
  
  Output consists of a blinding public key pk and a blinding secret key sk.

- Function BL-Blind-Public-Key(pk, \(\tau\)) \(\rightarrow\) 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-Secret-Key(sk, \(\tau\)) \(\rightarrow\) sk\(_\tau\):
  
  Deterministically compute a blinded secret key.

  Input consists of a blinding secret key sk and a blinding factor \(\tau\).
  
  Output consists of the blinded secret key sk\(_\tau\).

- Integer L\(_\text{bl}\): The length of the blinding factor \(\tau\) in octets.

  pk and pk\(_\tau\) are opaque octet strings of arbitrary length. \(\tau\) is an opaque octet string of length L\(_\text{bl}\). The representations of sk, sk\(_\tau\) and L\(_\text{bl}\) are an undefined implementation detail.

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

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

- KEM-Generate-Keypair() \(\rightarrow\) (pk, sk): Generate a key encapsulation key pair.
  
  No input.
  
  Output consists of public key pk and secret key sk.

- KEM-Encaps(pk) \(\rightarrow\) (k, c): Generate a key encapsulation.
Input consists of an encapsulation public key pk.

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

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

Input consists of encapsulation secret key sk and encapsulation ciphertext c.

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

pk, k and c are opaque octet strings. The representation of sk is an undefined implementation detail.

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

*MAC: A message authentication code (MAC) scheme, consisting of:

-Function MAC-Tag(k, m) -> t: Generate a message authentication tag for a given message using a given key.

Input consists of the shared MAC key k and the message m.

Output consists of the MAC tag t.

-Function MAC-Verify(k, m, t) -> { 0, 1 }: Verify a message authentication tag.

Input consists of the shared MAC key k, the message m and the MAC tag t.

Output is 1 if and only if MAC-Tag(k, m) = t.

-Integer L_mac: The length of the MAC key k in octets.

k is an opaque octet string of length L_mac. m and t are opaque octet strings of arbitrary length. The representation of L_mac is an undefined implementation detail.

See [Frymann2020] for definitions of security properties required of the message authentication code scheme MAC.

*KDF: A variable-length key derivation function with the signature: KDF(info, ikm, L) -> okm

Input consists of a domain separation parameter info, input key material ikm and output length L.
Output consists of output key material okm of length L in octets.

info and ikm are opaque octet strings of arbitrary length. okm is an opaque octet string of length L. L is an integer with undefined representation.

See [Frymann2020] for definitions of security properties required of the key derivation function KDF.

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

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-Secret-Key output ECDSA keys, then the ARKG instance will also output ECDSA keys.

Instantiations **MUST** satisfy the following compatibility criteria:

*The output shared secret k of KEM-Encaps and KEM-Decaps is a valid input key material ikm of KDF.

*Output key material okm of length L_bl of KDF is a valid input blinding factor tau of BL- Blind- Public-Key and BL- Blind- Secret-Key.

It is permissible for some KDF outputs to not be valid blinding factors, as long as this happens with negligible probability - see section [Design Rationale: Using a MAC].

*Output key material okm of length L_mac of KDF is a valid input MAC key k of MAC- Tag(k, m) and MAC- Verify(k, m, t).

It is permissible for some KDF outputs to not be valid MAC keys, as long as this happens with negligible probability - see section [Design Rationale: Using a MAC].

We denote a concrete ARKG instance by the pattern ARKG- BL- KEM- MAC- KDF, substituting the chosen instantiation for the BL, KEM, MAC and KDF parts. 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.

### 1.2.3. The function ARKG- Generate- Seed

This function is performed by the delegating party. The delegating party generates 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 generate public keys on behalf of the delegating party.

``` ARKG-Generate-Seed() -> (pk, sk) Options: BL The key blinding scheme chosen for the ARKG instantiation. KEM The key encapsulation mechanism chosen for the ARKG instantiation.

Inputs: None

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

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

\[(pk_{kem}, sk_{kem}) = KEM-Generate-Keypair() \quad (pk_{bl}, sk_{bl}) = BL-Generate-Keypair() \quad pk = (pk_{kem}, pk_{bl}) \quad sk = (sk_{kem}, sk_{bl})\]```

1.2.4. The function ARKG-Derive-Public-Key

This function is performed by the subordinate party, which holds the ARKG public seed pk = (pk_{kem}, pk_{bl}). 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, in order to generate any number of public keys.

``` ARKG-Derive-Public-Key((pk_{kem}, pk_{bl}), info) -> (pk', kh) Options: BL The key blinding scheme chosen for the ARKG instantiation. KEM The key encapsulation mechanism chosen for the ARKG instantiation. MAC The MAC scheme chosen for the ARKG instantiation. KDF The key derivation function chosen for the ARKG instantiation. L_bl The length in octets of the blinding factor tau of the key blinding scheme BL. L_mac The length in octets of the MAC key of the MAC scheme MAC.```
If this procedure aborts due to an error, for example because KDF returns an invalid tau or mk, the procedure can safely be retried with the same arguments.

1.2.5. The function ARKG-Derive-Secret-Key

This function is performed by the delegating party, which holds the ARKG private seed \((sk_kem, sk_bl)\). The resulting secret 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 secret keys any number of times.

\[
\text{ARKG-Derive-Secret-Key}((sk_kem, sk_bl), kh, info) \rightarrow sk'
\]

Options: 
- **BL** The key blinding scheme chosen for the ARKG instantiation. 
- **KEM** The key encapsulation mechanism chosen for the ARKG instantiation. 
- **MAC** The MAC scheme chosen for the ARKG instantiation. 
- **KDF** The key derivation function chosen for the ARKG instantiation. 
- **L_bl** The length in octets of the blinding factor \(tau\) of the key blinding scheme BL. 
- **L_mac** The length in octets of the MAC key of the MAC scheme MAC.

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

\[
\begin{align*}
(k, c) &= \text{KEM-Encaps}(pk_kem) \\
tau &= \text{KDF}(\text{"arkg-blind" || 0x00 || info, k, L_bl}) \\
mk &= \text{KDF}(\text{"arkg-mac" || 0x00 || info, k, L_mac}) \\
tag &= \text{MAC-Tag}(mk, c || info) \\

pk' &= \text{BL-Blind-Public-Key}(pk_bl, tau) \\
kh &= (c, tag)
\end{align*}
\]

If this procedure aborts due to an error, for example because KDF returns an invalid \(tau\) or \(mk\), the procedure can safely be retried with the same arguments.
Inputs:

- `sk_kem`: A key encapsulation secret key.
- `sk_bl`: A key blinding secret key.
- `kh`: A key handle output from ARKG-Derive-Public-Key.
- `info`: Optional context and application specific information (can be a zero-length string).

Output:

- `sk'`: A blinded secret key.

The output `sk'` is calculated as follows:

\[
(c, \text{tag}) = kh \\
k = \text{KEM-Decaps}(sk_{kem}, c) \\
mk = \text{KDF}(\text{"arkg-mac" || 0x00 || info, k, L_mac})
\]

If \(\text{MAC-Verify}(mk, c || info, \text{tag}) = 0\):

Abort with an error.

\[
tau = \text{KDF}(\text{"arkg-blind" || 0x00 || info, k, L_bl}) \\
sk' = \text{BL-Blind-Secret-Key}(sk_{bl}, tau)
\]

Errors in this procedure are typically unrecoverable. For example, KDF might return an invalid tau or mk, or the tag may be invalid. 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 a correct and honest implementation of ARKG-Derive-Secret-Key. See also section [Design Rationale: Using a MAC].

1.3. Generic ARKG instantiations

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

TODO: IANA registry? COSE/JOSE?

1.3.1. Using elliptic curve arithmetic for key blinding

Instantiations of ARKG whose output keys are elliptic curve keys can use elliptic curve arithmetic as the key blinding scheme BL. Frymann2020 [Wilson] This section defines a general formula for such instantiations of BL.
Let \text{crv} be an elliptic curve. Then the BL parameter of ARKG may be instantiated as follows:

*Elliptic curve points are encoded to and from octet strings using the procedures defined in sections 2.3.3 and 2.3.4 of SEC 1 [SEC1].

*Elliptic curve scalar values are encoded to and from octet strings using the procedures defined in sections 2.3.7 and 2.3.8 of SEC 1 [SEC1].

*N is the order of \text{crv}.

*G is the order of \text{crv}.

``` `BL-Generate-Keypair() -> (pk, sk)`

\text{sk} = \text{Random}(1, \ N)
\text{pk\_tmp} = \text{G}^{\text{sk}}
If \text{pk\_tmp} equals the point at infinity, abort with an error.
\text{pk} = \text{pk\_tmp}

\text{TODO: Also reject G?}

`BL-Blind-Public-Key(pk, \tau) -> pk\_tau`

If \tau = 0 or \tau >= N, abort with an error.
\text{pk\_tau\_tmp} = \text{pk} * (\text{G}^{\tau})
If \text{pk\_tau\_tmp} equals the point at infinity, abort with an error.
\text{pk\_tau} = \text{pk\_tau\_tmp}

\text{TODO: Also reject G?}

`BL-Blind-Secret-Key(sk, \tau) -> sk\_tau`

If \tau = 0 or \tau >= N, abort with an error.
\text{sk\_tau\_tmp} = \text{sk} + \tau
If \text{sk\_tau\_tmp} = 0, abort with an error.
\text{sk\_tau} = \text{sk\_tau\_tmp}

\text{TODO: Also reject 1?  
```

1.3.2. Using ECDH as the KEM

Instantiations of ARKG can use ECDH [RFC6090] as the key encapsulation mechanism. This section defines a general formula for such instantiations of KEM.
Let $crv$ be an elliptic curve used for ECDH. Then the KEM parameter of ARKG may be instantiated as follows:

*Elliptic curve points are encoded to and from octet strings using the procedures defined in sections 2.3.3 and 2.3.4 of [SEC1].

*Elliptic curve coordinate field elements are encoded to and from octet strings using the procedures defined in sections 2.3.5 and 2.3.6 of [SEC1].

*Elliptic curve scalar values are encoded to and from octet strings using the procedures defined in sections 2.3.7 and 2.3.8 of [SEC1].

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

*N is the order of $crv$.

*G is the generator of $crv$.

```plaintext
KEM-Generate-Keypair() -> (pk, sk)
KEM-Encaps(pk) -> (k, c) (pk', sk') = KEM-Generate-Keypair()
KEM-Decaps(sk, c) -> k
```

1.3.3. Using both elliptic curve arithmetic for key blinding and ECDH as the KEM

If elliptic curve arithmetic is used for key blinding and ECDH is used as the KEM, as described in the previous sections, then both of them MAY use the same curve or MAY use different curves. If both use the same curve, then it is also possible to use the same public key as both the key blinding public key and the KEM public key. [Frymann2020]
1.3.4. Using HMAC as the MAC

Let Hash be a cryptographic hash function. Then the MAC parameter of ARKG may be instantiated using HMAC [RFC2104] as follows:

```
MAC-Tag(k, m) -> t
```

\[ t = \text{HMAC-Hash}(K=k, \text{text}=m) \]

```
MAC-Verify(k, m, t) -> \{0, 1\}
```

\[ t' = \text{HMAC-Hash}(K=k, \text{text}=m) \]

If \( t = t' \):
   return 1
Else:
   return 0

1.3.5. Using HKDF as the KDF

Let Hash be a cryptographic hash function. Then the KDF parameter of ARKG may be instantiated using HKDF [RFC5869] as follows:

```
KDF(info, ikm, L) -> okm
```

\[ \text{PRK} = \text{HKDF-Extract} \text{ with the arguments:} \]
\[ \text{Hash: Hash} \]
\[ \text{salt: not set} \]
\[ \text{IKM: ikm} \]

\[ \text{okm} = \text{HKDF-Expand} \text{ with the arguments:} \]
\[ \text{Hash: Hash} \]
\[ \text{PRK: PRK} \]
\[ \text{info: info} \]
\[ \text{L: L} \]

1.4. Concrete ARKG instantiations

This section defines an initial set of concrete ARKG instantiations.

TODO: IANA registry? COSE/JOSE?
1.4.1. ARKG-P256-ECDH-P256-HMAC-SHA256-HKDF-SHA256

The identifier ARKG-P256-ECDH-P256-HMAC-SHA256-HKDF-SHA256 represents the following ARKG instantiation:

*BL: Elliptic curve arithmetic as described in section [Using elliptic curve arithmetic for key blinding] with the parameter:

  -crv: The NIST curve secp256r1 [SEC2].

*KEM: ECDH [RFC6090] as described in section [Using ECDH as the KEM] with the parameter:

  -crv: The NIST curve secp256r1 [SEC2].

*MAC: HMAC as described in section [Using HMAC as the MAC] with the parameter:

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

*KDF: HKDF as described in section [Using HKDF as the KDF] with the parameter:

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

*L_bl: 32

*L_mac: 32

1.4.1.1. ARKG-P384-ECDH-P384-HMAC-SHA384-HKDF-SHA384

The identifier ARKG-P384-ECDH-P384-HMAC-SHA384-HKDF-SHA384 represents the following ARKG instantiation:

*BL: Elliptic curve arithmetic as described in section [Using elliptic curve arithmetic for key blinding] with the parameter:

  -crv: The NIST curve secp384r1 [SEC2].

*KEM: ECDH [RFC6090] as described in section [Using ECDH as the KEM] with the parameter:

  -crv: The NIST curve secp384r1 [SEC2].

*MAC: HMAC as described in section [Using HMAC as the MAC] with the parameter:

  -Hash: SHA-384 [FIPS 180-4].
*KDF: HKDF as described in section [Using HKDF as the KDF] with the parameter:
  - Hash: SHA-384 [FIPS 180-4].

*L_bl: 48

*L_mac: 48

1.4.1.2. ARKG-P521-ECDH-P521-HMAC-SHA512-HKDF-SHA512

The identifier ARKG-P521-ECDH-P521-HMAC-SHA512-HKDF-SHA512 represents the following ARKG instantiation:

*BL: Elliptic curve arithmetic as described in section [Using elliptic curve arithmetic for key blinding] with the parameter:
  - crv: The NIST curve secp521r1 [SEC2].

*KEM: ECDH [RFC6090] as described in section [Using ECDH as the KEM] with the parameter:
  - crv: The NIST curve secp521r1 [SEC2].

*MAC: HMAC as described in section [Using HMAC as the MAC] with the parameter:
  - Hash: SHA-512 [FIPS 180-4].

*KDF: HKDF as described in section [Using HKDF as the KDF] with the parameter:
  - Hash: SHA-512 [FIPS 180-4].

*L_bl: 64

*L_mac: 64

1.4.1.3. ARKG-P256k-ECDH-P256k-HMAC-SHA256-HKDF-SHA256

The identifier ARKG-P256k-ECDH-P256k-HMAC-SHA256-HKDF-SHA256 represents the following ARKG instantiation:

*BL: Elliptic curve arithmetic as described in section [Using elliptic curve arithmetic for key blinding] with the parameter:
  - crv: The SECG curve secp256k1 [SEC2].
*KEM: ECDH [RFC6090] as described in section [Using ECDH as the KEM] with the parameter:

- crv: The SECG curve secp256k1 [SEC2].

*MAC: HMAC as described in section [Using HMAC as the MAC] with the parameter:

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

*KDF: HKDF as described in section [Using HKDF as the KDF] with the parameter:

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

*L_bl: 32

*L_mac: 32

1.4.1.4. ARKG-Ed25519-X25519-HMAC-SHA256-HKDF-SHA256

The identifier ARKG-Ed25519-X25519-HMAC-SHA256-HKDF-SHA256 represents the following ARKG instantiation:

*BL: Elliptic curve arithmetic as described in section [Using elliptic curve arithmetic for key blinding] with the parameter:

- crv: The curve Ed25519 [REF?].

*KEM: ECDH [RFC6090] as described in section [Using ECDH as the KEM] with the parameter:

- crv: The curve X25519 [REF?].

*MAC: HMAC as described in section [Using HMAC as the MAC] with the parameter:

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

*KDF: HKDF as described in section [Using HKDF as the KDF] with the parameter:

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

*L_bl: 32

*L_mac: 32
1.4.1.5. ARKG-X25519-X25519-HMAC-SHA256-HKDF-SHA256

The identifier ARKG-X25519-X25519-HMAC-SHA256-HKDF-SHA256 represents the following ARKG instantiation:

*BL: Elliptic curve arithmetic as described in section [Using elliptic curve arithmetic for key blinding] with the parameter:

- crv: The curve X25519 [REF?].

*KEM: ECDH [RFC6090] as described in section [Using ECDH as the KEM] with the parameter:

- crv: The curve X25519 [REF?].

*MAC: HMAC as described in section [Using HMAC as the MAC] with the parameter:

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

*KDF: HKDF as described in section [Using HKDF as the KDF] with the parameter:

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

*L_bl: 32

*L_mac: 32

1.5. COSE bindings

TODO?: Define COSE representations for interoperability:
- ARKG public seed (for interoperability between different implementers of ARKG-Generate-Seed and ARKG-Derive-Public-Key)
- ARKG key handle (for interoperability between different implementers of ARKG-Derive-Public-Key and ARKG-Derive-Secret-Key)

2. Security Considerations

TODO

3. Privacy Considerations

TODO

4. IANA Considerations

TODO
5. Design rationale

5.1. Using a MAC

The ARKG construction by Wilson \[Wilson\] omits the MAC and instead encodes application context in the PRF labels, arguing 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. \[Frymann2020\] for two purposes.

The first is so that the delegating party can distinguish between key handles addressed to it and those addressed to other delegating parties. We anticipate use cases where a private key usage request may contain key handles for several delegating parties eligible to fulfill the request, and the delegate party to be used can be chosen opportunistically depending on which are available at the time. Without the MAC, choosing the wrong key handle would cause the ARKG-Derive-Secret-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. The MAC also allows ARKG key handles to be transmitted via heterogeneous data channels, possibly including a mix of ARKG key handles and similar values used for other algorithms.

The second purpose is so that the delegating party can be assured that no errors should happen during the execution of ARKG-Derive-Secret-Key, such as out-of-range or invalid key values. For example, key generation in ARKG-Derive-Public-Key might be done by randomly testing candidates \[NIST.SP.800-56Ar3\] and retrying ARKG-Derive-Public-Key until a valid candidate is found. A MAC enables ARKG-Derive-Secret-Key to assume that the first candidate from a given pseudo-random seed will be successful, and otherwise return an explicit error rejecting the key handle as invalid. ARKG-Derive-Public-Key is likely to run on powerful general-purpose hardware, such as a laptop, smartphone or server, while ARKG-Derive-Secret-Key might run on more constrained hardware such as a cryptographic smart card, which benefits greatly from such optimizations.

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. \[Frymann2020\]: the construction by Frymann et al. can be reduced to the ARKG construction in this document by instantiating KEM as group exponentiation and instantiating BL as group multiplication to blind public keys and modular integer addition to blind secret keys. The MAC and KDF parameters correspond trivially to the MAC and KDF parameters in \[Frymann2020\], where $KDF_1(k) = KDF(k, l_1)$ and $KDF_2(k) = KDF(k, l_2)$ with fixed labels $l_1$ and $l_2$. 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.

5.2. Implementation Status

TODO

6. References

TODO

TODO: Ask authors for canonical reference addresses

7. References

7.1. Normative References


7.2. Informative References


Appendix A. Acknowledgements

ARKG was first proposed under this name by Frymann et al. [Frymann2020], 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 Pieter Wuille [BIP32] used to create privacy-preserving Bitcoin addresses. Frymann et al. [Frymann2020] 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 Wilson [Wilson], Clermont [Clermont] and Frymann et al. [Frymann2023].

This document adopts the construction proposed by Wilson [Wilson], modified by the inclusion of a MAC in the key handles as done in the original construction by Frymann et al. [Frymann2020].

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

Appendix B. Test Vectors

TODO

Appendix C. Document History

-00 Initial Version

Contributors

Dain Nilsson
Yubico

Authors' Addresses

Emil Lundberg (editor)
Yubico
Kungsgatan 44
Stockholm
Sweden

Email: emil@emlun.se

John Bradley
Yubico

Email: ve7jtb@ve7jtb.com